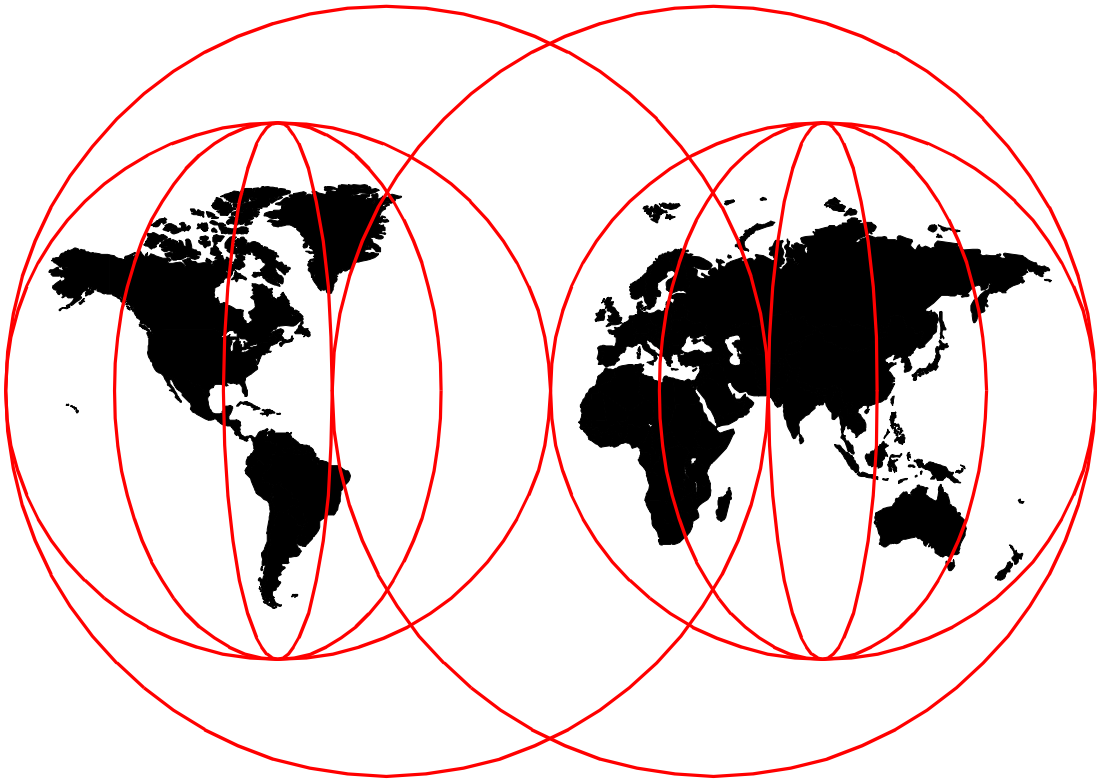
Building AS/400 Applications with Java

Bob Maatta, Leonardo Llames, Jennifer Maynard, Mohammad Omar Nishtar



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-2163-02

International Technical Support Organization

SG24-2163-02

Building AS/400 Applications with Java

June 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special Notices” on page 417.

Third Edition (June 1999)

This edition applies to Version 4, Release Number 2 and later of OS/400

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

Contents

Figures

Tables

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 redbook is intended for customers and service providers who need to install the AS/400 Developer Kit for Java, and for application developers who want to develop Java applications on the AS/400 system. By reading this redbook, you gain a fast start on your way to using Java with the AS/400 system.

This redbook explains how you can use Java and the AS/400 system to build server applications and client/server applications for the new network computing paradigm. It provides many practical programming examples with detailed explanations on how they work. It also describes how to modernize legacy RPG applications in a practical and evolutionary way using client and server Java examples (available for you to download from the IBM ITSO redbook Internet site). Plus, you get valuable tips on how to improve Java performance.

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 writes extensively and teaches IBM classes worldwide on all areas of AS/400 client/server and application development. Before joining the ITSO in 1995, he worked in the U.S. AS/400 National Technical Support Center as a Consulting Market Support Specialist. He has over 20 years of experience in the computer field and has worked with all aspects of personal computers since 1983. He is a Sun Certified Java Programmer and a Sun Certified Java Developer.

Leonardo Llames is a Consulting I/T Specialist at the IBM AS/400 Systems Center in Rochester, MN. He has been working in the areas of performance analysis, application and database design, and client-server technologies for IBM customers for the past 10 years and currently works in object-oriented technology and Java. Prior to his assignment to Rochester, he was an application developer for IBM Business Partners.

Jennifer Maynard is a software engineer in the IBM Rochester User Technologies area. Jennifer is a team leader in the Application Enablement and Development department, where she designs and writes online Java product information. She holds a BS degree in Scientific and Technical Communication from Michigan Technological University.

Mohammad Omar Nishtar is an Information Technology Specialist with IBM Global Services in Pakistan. He has been with IBM since 1987. His areas of expertise include AS/400 application development using Java, RPG, and Cobol, AS/400 communications and project management. He has taught numerous courses on AS/400 application development.

The authors of the previous editions of this redbook were:

Simon Coulter, FlyByNight Software
Jim Fair, IBM Rochester
Hal Frye, IBM Boulder
Pierre Goudet, IBM France
Leonardo Llames, IBM Rochester
Bob Maatta, ITSO Rochester
Brian Skaarup, EDB Gruppen Systems A/S
Daniel Stucki, DV Bern AG
Neil Willis, ITSO Rochester

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

Susan Gantner
Barbara Morris
IBM Toronto Laboratory

Jerry Wille
Blair Wyman
IBM Rochester Laboratory

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:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 433, to the fax number shown on the form.
- Use the online evaluation form found at: `http://www.redbooks.ibm.com/`
- Send your comments in an internet note to: `redbook@us.ibm.com`

Chapter 1. Introduction

This redbook covers Java on the AS/400 system by focusing on four main areas:

- An overview of Java on the AS/400 system
- Installing Java on the AS/400 system
- Building AS/400 applications with Java
- Performance considerations and analysis of AS/400 Java applications

First, we provide an overview of Java and its key components. Then, we cover how Java is implemented on the AS/400 system and provide details of AS/400 specific considerations. Next, we show you how to install Java on the AS/400 system and how to set up the Java environment.

Then, we cover building AS/400 applications using Java. In this section, we cover migrating an existing RPG Order Entry application to Java. In the migration scenario, we first build a client-based Java graphical user interface (GUI) program that interfaces with the existing AS/400 RPG application. We then migrate the RPG application to Java and use the Java remote method invocation (RMI) feature to allow the client Java program to interface with the AS/400 server Java program.

Finally, we do a performance measurement and analysis of the application to identify problem areas and improve response time.

The migration of an existing RPG Order Entry application to Java scenario is shown in the following diagrams. We start with an existing RPG Order Entry application. For more details about this application, see Chapter 5, “Overview of the Order Entry Application” on page 87.

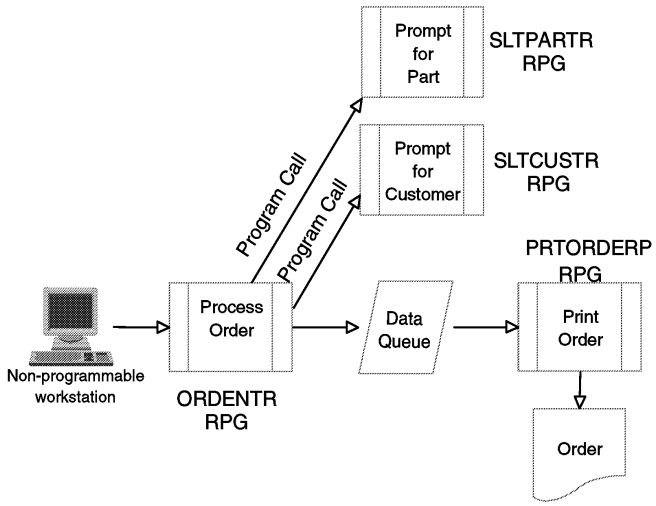


Figure 1. RPG Order Entry Application

In the second version of the application, we develop a Java client GUI. We modify the AS/400 RPG application to allow it to interface with the new Java client program, but still function through a 5250 interface. For more details about this scenario, see Chapter 6, “Migrating the User Interface to Java Client” on page 101.

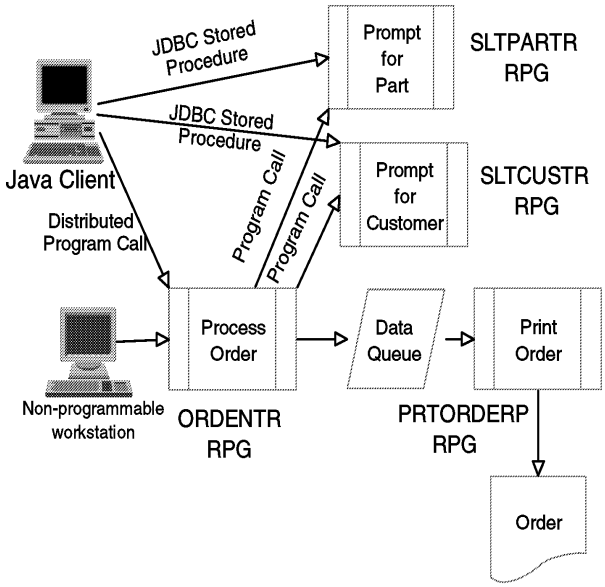


Figure 2. Java Client Order Entry Application

Next, we change the RPG Order Entry program to a Java program on AS/400. We modify the client Java program to allow it to interface with the new Java server program through the RMI. For details about this scenario, see Chapter 7, “Moving the Server Application to Java” on page 133.

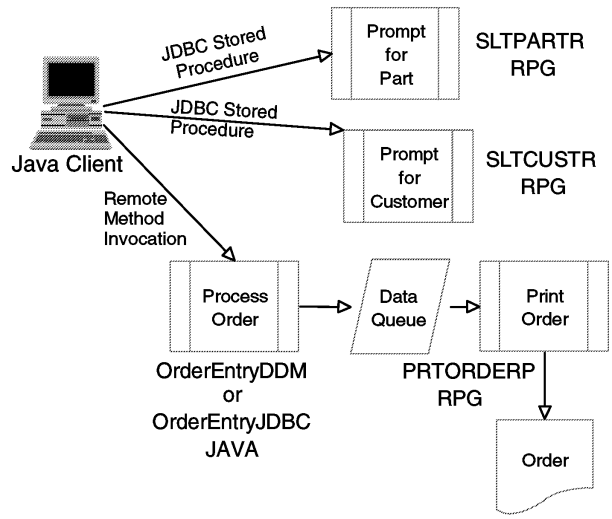


Figure 3. Java Client/Java Server Order Entry Application

In Chapter 10, “Applying Object Oriented Technology to the Application” on page 251, we do a object-oriented analysis and implementation of the application.

In our new design, we separate the application into three distinct layers:

- The *Views layer* represents the end user interface.
- The *Business Objects layer* represents the actual business logic that we implement to satisfy the application requirement. We also refer to this layer as the Domain layer.
- The *Database Access layer* represents how we actually access the data stored in the AS/400 databases. Our goal is to make each layer independent of the layer that it interfaces to. For example, we may decide to change the end user interface for our application or we may decide to change the way we access the Customer database from a JDBC interface to a JDBC Stored Procedure interface. We want to make these changes without having to change anything in the layers to which they interface.

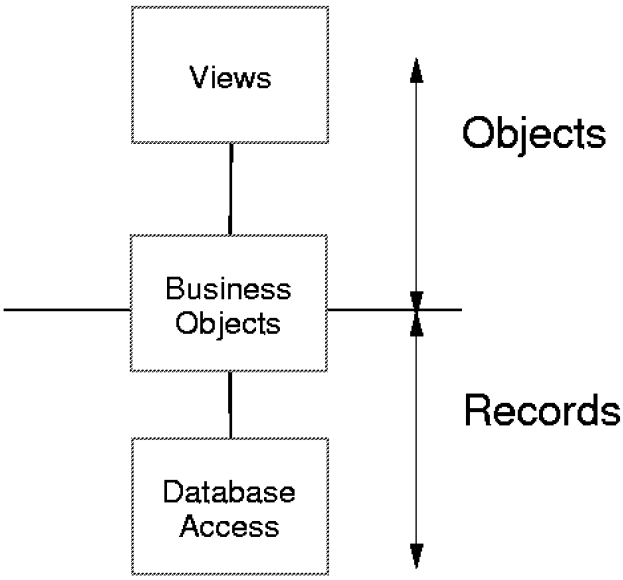


Figure 4. Object-Oriented Design Approach

Java on the AS/400 system results in a different environment from what most AS/400 developers previously used. From a work management point of view, there are several new concepts with which an application developer should be familiar. We cover these concepts in Chapter 13, “Java Performance and Work Management Overview” on page 323. Performance measurement and analysis of AS/400 Java applications is important in identifying and correcting performance problems. In Chapter 14, “Java Application Design and Host Performance Analysis” on page 355, we provide an in-depth analysis and solution of a Java performance problem.

We also provide information on several complementary topics. These include the VisualAge for Java — AS/400 unique support, debugging Java applications on the AS/400 system and designing GUIs with Java.

To maximize the benefits from this book, we assume that you have these certain skills:

- General AS/400 operational skills and Windows 95/98/NT skills to install Java on the AS/400 system and on your workstation
- Proficiency with a Web browser
- Proficiency in Java programming, object-oriented design techniques, and AS/400 application development methods, as well as DB2/400 skills if you plan on developing Java based applications

Note: This book is not designed to teach basic Java skills. See Appendix C, “Related Publications” on page 419, for a list of books that are freely available and translated into many languages.

Chapter 2. Java Overview and AS/400 Implementation

In this chapter, we discuss the Java platform architecture. First, we look at how Sun Microsystems, Inc. defines the Java platform and implements it in their Java Development Kit (JDK). Then, we present the Java implementation on the AS/400 system and explain some of the AS/400-specific aspects of this implementation.

2.1 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 yet 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 capability of developing 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

2.1.1 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is the center piece of the Java platform. It is the "engine" of Java. The JVM is responsible for running Java applications or applets 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

2.1.1.1 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 applications or applets 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.

2.1.1.2 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.

2.1.1.3 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.

2.1.1.4 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.

2.1.1.5 Java Native Interface (JNI)

You can view the Java Native Interface (JNI) 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.

2.1.1.6 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 Section 2.1.3, “Java Utilities” on page 10.

2.1.2 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 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.

2.1.2.1 Core Library

The classes included in the Core library are grouped into several packages. Currently, the Core library consists of these packages:

- **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 (RMI) 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 are building 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) incorporates several new features that further enhance a developer's ability to deliver scalable, commercial, 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.

2.1.2.2 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 a 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 JavaTelephony.
- **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 to name a few.
- **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.

2.1.3 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
- rmic

- rmiregistry
- serialver
- native2ascii

2.1.3.1 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[]){
    . . .
    }
}
```

2.1.3.2 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.

2.1.3.3 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.

2.1.3.4 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.

2.1.3.5 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, the use of this tool may violate the license agreement for the class that you are disassembling.

2.1.3.6 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.

2.1.3.7 The jar Tool

The **jar** tool combines several Java class files into a single Java ARchive (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.

2.1.3.8 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, and also 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.

2.1.3.9 The appletviewer Tool

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

2.1.3.10 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.

2.1.3.11 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.

2.1.3.12 The serialver Command

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

2.1.3.13 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. The documentation is provided as an HTML file that can be viewed with any browser. The documentation is stored in the DOCS subdirectory of the JDK1.1.4 directory and is called index.html.

2.2 Java on the AS/400 System

Starting with V4R2 of OS/400, the AS/400 system implements the Java platform. This implementation fully complies with JDK 1.1.4 (V4R2), JDK 1.1.6 (V4R3), and JDK 1.1.7 (V4R4) as defined by Sun Microsystems, Inc. As in any other Java platform implementation, the AS/400 system also provides these components:

- Java Virtual Machine (JVM)
- Java APIs
- Java Utilities

The AS/400 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
- DB2/400 JDBC driver
- Multi-process design point

Figure 5 on page 15 in the following section shows a high-level diagram of the AS/400 Java implementation.

2.2.1 AS/400 Java Virtual Machine

On the AS/400 system, the JVM is implemented within SLIC, below the Technology Independent Machine Interface (TIMI), and 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.

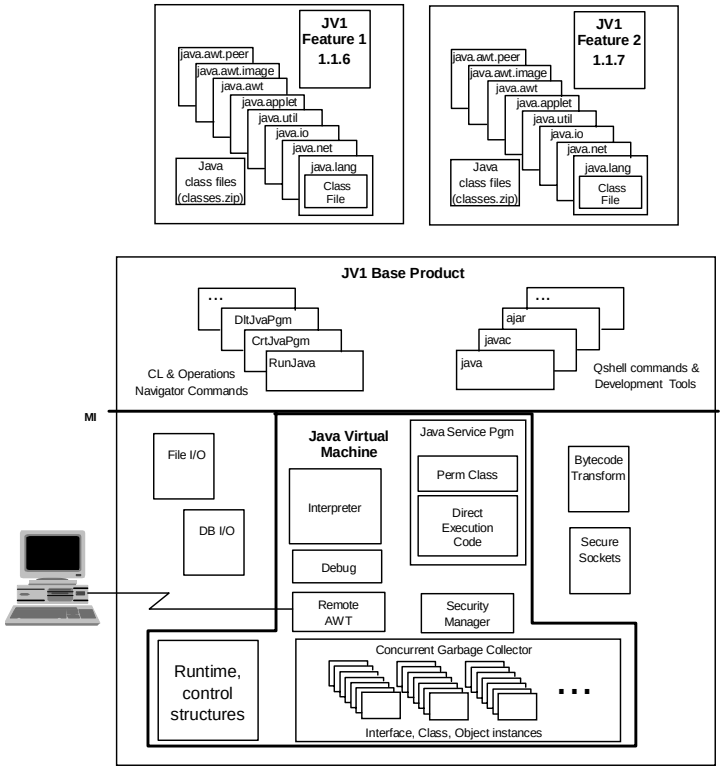


Figure 5. AS/400 Java Virtual Machine and AS/400 Developer Kit for Java

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, in 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 AS/400 system, see Chapter 9, “Java Native Interface” on page 205.

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 APIs Reference* 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.

2.2.2 Java APIs and AS/400

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 AS/400 system. You need to install the AS/400 Developer Kit for Java (5769-JV1) LPP by using the standard OS/400 Install Licensed Program procedures. Chapter 3, “Installation” on page 33, contains all of the details on how to install this LPP on your system.

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 AS/400 (5769-TC1) LPP product. You must install this no charge LPP on your system before using Java on your AS/400 system.

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

The *java.awt* APIs use the Remote AWT support of the AS/400 JVM to route any GUI operation to a workstation. This is because there are no GUI-capable devices on the AS/400 system. See Section 2.3, “AS/400 Specific Implementation” on page 20, for more details on Remote AWT.

2.2.3 Java Utilities and AS/400

Most Java utilities are supported on the AS/400 system. They are run from within the Qshell Interpreter.

The Qshell Interpreter is an option of OS/400 that you must install on your AS/400 system to run any Java utility on your system. Refer to Chapter 3, “Installation” on

page 33, for more details on how to install this option on your AS/400 system. Also, see Section 2.3, “AS/400 Specific Implementation” on page 20, for more details on the 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

With the following exceptions, **qsh** supports the syntax and options of the Java utilities, except the ajar tool, as they are described in the standard JDK documentation.

2.2.3.1 The java Command

The **java** command runs Java programs that are associated with the specified Java class. The AS/400 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 AS/400 implementation provides the capability to debug Java applications using the standard OS/400 debugging tools. Use the equivalent Run Java (RUNJVA) or JAVA OS/400 command to debug a Java program. The VisualAge for Java AS/400 feature provides a cooperative debugger that is capable of debugging AS/400 Server Java applications from a workstation. This is the currently available Code/400 Cooperative debugger that has been enhanced to support Java.
- **-noasyncgc**: The AS/400 implementation uses a highly scalable garbage collector instead of the standard JDK garbage collector. You can tune the AS/400 garbage collector using the new special options.
- **-noclassgc**: As previously mentioned, the AS/400 garbage collector uses its own options. The standard **java** command options do not apply.
- **-help**: To get help on how to use the **java** command, use the RUNJVA 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**: Sends output profiling information to a specified file or to the java.prof file in the current directory. The AS/400 system has its own performance analysis tools.

- **-ss**: Stack size does not apply to the AS/400 Java runtime environment, because of the AS/400 architecture. There is no such thing as maximum stack size in the AS/400 environment.
- **-oss**: Again, there is no such thing as maximum stack size in the AS/400 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 the RUNJVA or JAVA OS/400 command, tracing may not be available.
- **-verify**: The Java bytecodes are verified, and then translated into AS/400 RISC PowerPC machine instructions when you run the CRTJVAPGM command, or the RUNJVA or JAVA OS/400 command. All of the standard **java** utility verify options do not apply to the AS/400 Java runtime environment.
- **-verifyremote**: See the preceding -verify option.
- **-noverify**: See the preceding -verify option.

The AS/400 version of the **java** command supports these specific AS/400 options:

- **-secure**: This option tells the AS/400 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. Whichever class file 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 AS/400 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 AS/400 garbage collector collection priority. The lower the number, the lower the priority. 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 AS/400 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 Section 2.3.1, “The OS/400 Java Commands” on page 20, for more details on OS/400 Java-related commands.

2.2.3.2 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:
-classpath: The -classpath option does not override the default classpath. Instead, it is appended to the system default classpath. The -classpath option does override the CLASSPATH environment variable.

2.2.3.3 The javah Command

The **javah** command on AS/400 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 AS/400 system. The JNI-style C language header file is always created in the current working directory. The header file is created as an AS/400 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 AS/400 system. Use the Copy from Stream File (CPYFRMSTMF) OS/400 command to do so.
- **-stubs**: If you specify this option, it is ignored by the AS/400 implementation.
- **-trace**: If you specify this option, it is ignored by the AS/400 implementation.
- **-v**: Verbose. This option is not supported.

2.2.3.4 The javap Command

The **javap** command on AS/400 ignores these options:

- **-b**: This option is for backward compatibility with previous releases of the JDK. As the initial implementation of the JDK on the AS/400 system is version 1.1.4 of the JDK, it did not make sense to support this option on the AS/400 system.
- **-p**: On AS/400, -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 AS/400 system.

2.2.3.5 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

2.2.3.6 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 AS/400 system 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 AS/400 system using remote AWT.

2.3 AS/400 Specific Implementation

In this section, we explain some aspects of the Java implementation that are specific to the AS/400 system, such as:

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

2.3.1 The OS/400 Java Commands

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

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

2.3.1.1 The CRTJVAPGM Command

The AS/400 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 RUNJVA command or JAVA command.

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 AS/400 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 RUNJVA 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 AS/400. 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 AS/400 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 REPLACE parameter allows you to specify whether an existing AS/400 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 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 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.

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

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

Figure 6 shows the OS/400 prompt for the CRTJVAPGM command.

```
                        Create Java Program (CRTJVAPGM)

Type choices, press Enter.

Class file or JAR file . . . . .

Optimization . . . . . .   10            10, *INTERPRET, 20, 30, 40
Replace program  . . . . . .   *YES       *YES, *NO
Enable performance collection  .   *NONE  *NONE, *ENTRYEXIT, *FULL

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

Figure 6. Create Java Program (CRTJVAPGM) Command

2.3.1.2 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, and the LICOPT parameter. You can use these parameters just as you would with the CRTJVAPGM command. See Section 2.3.1.1, “The CRTJVAPGM Command” on page 20, 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 parameter, ENBPFRCOL parameter, and LICOPT parameter.

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.

2.3.1.3 DLTJVAPGM Command

The Delete Java Program (DLTJVAPGM) command deletes an AS/400 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.

2.3.1.4 DSPJVAPGM Command

The Display Java Program (DSPJVAPGM) command displays information about the AS/400 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.

Figure 7 shows the output of the DSPJVAPGM command.

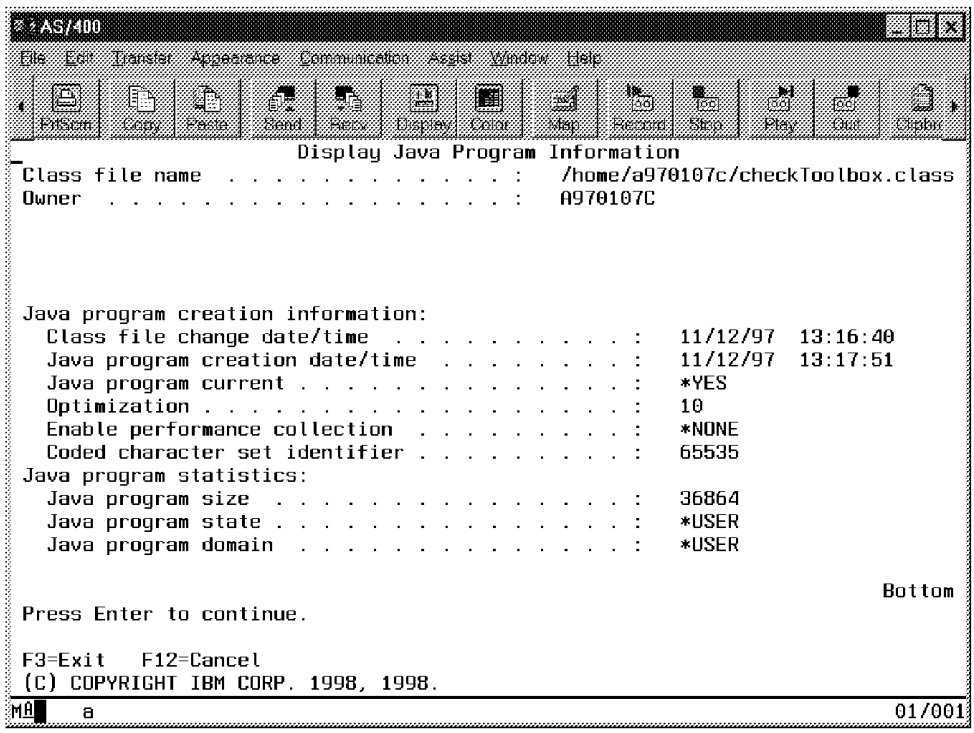


Figure 7. Display Java Program (DSPJVAPGM) Command

2.3.1.5 The RUNJVA (or JAVA) Command

The Run Java (RUNJVA) command or JAVA command runs the AS/400 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 3, “Installation” on page 33, 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 AS/400 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 Section 2.3.1.1, “The CRTJVAPGM Command” on page 20. 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 AS/400 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 AS/400 system.

- **GCPTY**: Specifies the AS/400 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 AS/400 system.

- **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 AS/400 system 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 8 on page 27 shows the RUNJVA command prompt.

```
                          Run Java Program (RUNJVA)

Type choices, press Enter.

Class  . . . . . .

Parameters . . . . . .   *NONE



              + for more values



Classpath  . . . . . .   *ENVVAR


                          Additional Parameters

Classpath security check level     *WARN         *WARN, *SECURE, *IGNORE
                                                                    More...
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
Parameter CLASS required.
```

Figure 8. Run Java Program (RUNJVA) Command

2.3.2 The Qshell Interpreter

The Qshell Interpreter, **qsh**, is a command interpreter for the AS/400 system 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.

2.3.2.1 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 9 on page 29 shows the shell terminal window.

Figure 9. 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 a 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**: This key 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 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, (at http://as400bks.rochester.ibm.com) for a complete description of the AS/400 Qshell Interpreter features and functions.

2.3.3 Remote AWT Support

The Remote Abstract Windowing Toolkit (AWT) is an implementation of the Java Abstract Windowing Toolkit (AWT). The AS/400 system does not have a native GUI device. Remote AWT allows Java applications, which have a graphical user interface, to run on the AS/400 system.

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 AS/400 system) 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 10 on page 31 shows a Remote AWT implementation.

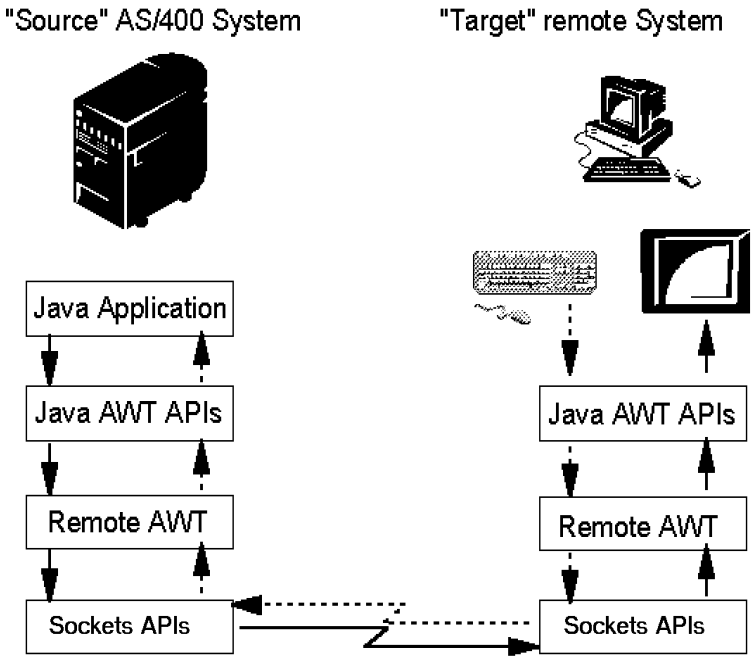


Figure 10. Remote AWT (V4R3 and Later)

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 AS/400 system without any modification, even though that application uses Java AWT support while the AS/400 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 AS/400 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 11 on page 32 shows the architecture of a standard client/server application.

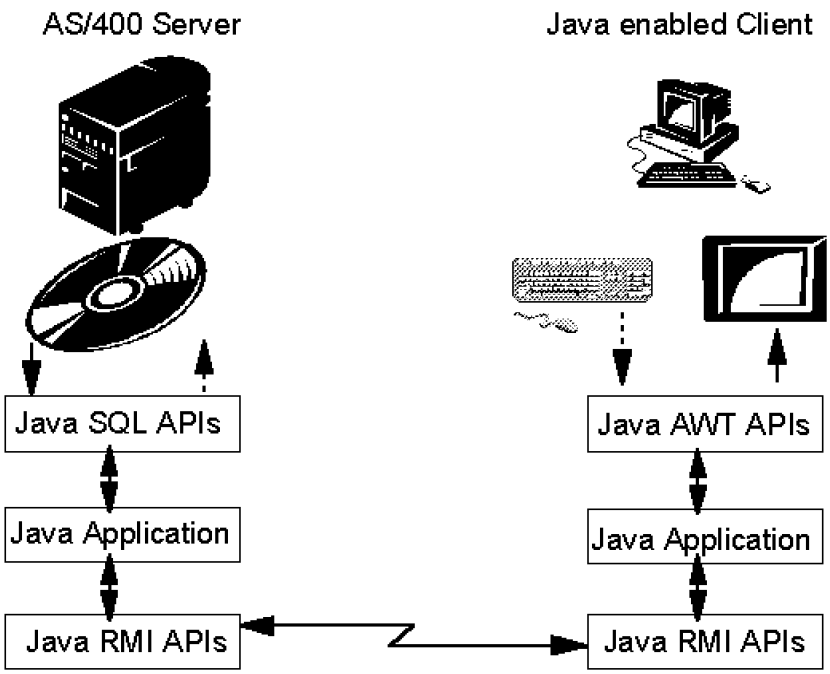


Figure 11. AS/400 Server and Java-Enabled Client

This is a much "cleaner" design where the 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 AS/400 system, have an industrial strength database management system (DBMS) that is designed to handle heavy data base operations in a secure multi-user environment. Take care when placing enterprise data on client workstations.

Chapter 3. Installation

This chapter provides the information that you need to successfully install various software components that are required to run Java on the AS/400 system. It covers:

- Installing Java support on the AS/400 system
- Installing the Sun Microsystems, Inc. Java Development Kit (JDK) on your workstation
- Installing the AS/400 Toolbox for Java on a workstation
- Setting up the runtime environment variables, such as the PATH and CLASSPATH directives
- Configuring and running Remote AWT

In this book, we use a PC-based workstation that runs Microsoft Windows 95 Operating System as an example. You can use any JVM-enabled workstation instead of a Windows 95 system. This includes RS/6000 with AIX workstations, PCs running IBM OS/2 Warp Version 4 or Microsoft Windows NT, Apple Macintosh, or any JVM-capable network computer, such as the IBM Network Station.

3.1 Installing Java on the AS/400 System

To install Java on the AS/400 system, perform these steps:

1. Sign on to the system using the QSECOFR user profile. At the Sign On display, enter `QSECOFR` on the User prompt.
2. Enter the corresponding password on the Password prompt, as shown in Figure 12.
3. Press the **Enter** key to display the AS/400 Main Menu.

```
                              Sign On
                                          System  . . . . . :   AS400SYS
                                          Subsystem . . . . :   QCTL
                                          Display . . . . . :   DSP01

          User  . . . . . . . . . . . . . .   QSECOFR
          Password  . . . . . . . . . . . .
          Program/procedure . . . . . . . .
          Menu  . . . . . . . . . . . . . .
          Current library . . . . . . . . .









                                    (C) COPYRIGHT IBM CORP. 1980, 1998.
```

Figure 12. Sign On to Your AS/400 Using the QSECOFR User Profile

4. Enter the `GO LICPGM` command (as shown in Figure 13) on the AS/400 Main Menu command line. The system displays the Work with Licensed Programs menu.

```
MAIN                              AS/400 Main Menu
                                                          System:   AS400SYS
Select one of the following:

     1. User tasks
     2. Office tasks
     3. General system tasks
     4. Files, libraries, and folders
     5. Programming
     6. Communications
     7. Define or change the system
     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access/400 tasks

    90. Sign off

Selection or command
===> GO LICPGM

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
(C) COPYRIGHT IBM CORP. 1980, 1998.
```

Figure 13. AS/400 Main Menu

Now, you need to check to see if all of the required software is already installed on your AS/400 system. All AS/400 systems that are shipped from IBM that have OS/400 Version 4 Release 2 (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 as described in Section 3.2, “Manually Installing Java Support on the AS/400 System” on page 41.

3.1.1 Checking to See What Software Is Installed

To check what software is installed on your AS/400 system, perform the following series of steps:

1. Enter option `10` on the Work with Licensed Programs menu command line, as shown in Figure 14 on page 35. All of the LPPs that are currently installed on your system are displayed.

```
LICPGM                  Work with Licensed Programs
                                                        System:   AS400SYS
Select one of the following:

  Manual Install
     1. Install all

  Preparation
     5. Prepare for install

  Licensed Programs
    10. Display installed licensed programs
    11. Install licensed programs
    12. Delete licensed programs
    13. Save licensed programs


                                                                    More...
Selection or command
===> 10

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F16=AS/400 Main menu
```

Figure 14. Work with Licensed Program Menu

2. 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 your 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 we are looking for OS/400 options, select the Product Option information, as shown in Figure 15 on page 36.

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769SS1           OS/400 - Library QGPL
5769SS1           OS/400 - Library QUSRSYS
5769SS1    *BASE  Operating System/400
5769SS1    1      OS/400 - Extended Base Support
5769SS1    2      OS/400 - Online Information
5769SS1    3      OS/400 - Extended Base Directory Support
5769SS1    8      OS/400 - AFP Compatibility Fonts
5769SS1    9      OS/400 - *PRV CL Compiler Support
5769SS1    12     OS/400 - Host Servers
5769SS1    13     OS/400 - System Openness Includes
5769SS1    14     OS/400 - GDDM
5769SS1    15     OS/400 - Common Programming APIs Toolkit
5769SS1    16     OS/400 - Ultimedia System Facilities
5769SS1    18     OS/400 - Media and Storage Extensions
                                                                   More...
Press Enter to continue.

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

Figure 15. Display Licensed Programs with Product Options

If you are planning to develop client/server Java applications on the AS/400 system, you may require all of the LPPs and operating system options, as shown in Table 1.

Table 1. Licensed Programs and Product Options

Licensed Program	Product Option	Description
5769-SS1	12	OS/400 - Host Servers
5769-SS1	30	OS/400 - Qshell Interpreter
5799-XEH		Qshell Utilities for AS/400
5769-JC1		AS/400 Toolbox for Java
5769-JV1		AS/400 Developer Kit for Java
5769-TC1		TCP/IP Connectivity Utilities for AS/400
5763-XD1		Client Access/400 Optimized for Windows
5769-XW1		Client Access/400 Windows Family Base

To run Java applications on the AS/400 system, you must have OS/400 V4R2.

Starting with V4R2, Java applications that run on the AS/400 system can also use the AS/400 Toolbox for Java (5769-JC1) classes to access AS/400 resources, such as data queues, print and spool support, run OS/400 commands, or call AS/400 programs. For this reason, you must install the OS/400 Host Servers (5769-SS1, option 12), TCP/IP Connectivity Utilities for AS/400 (5769-TC1), and AS/400 Toolbox for Java (5769-JC1) on your AS/400 system.

You can use the Java classes that are provided in the AS/400 Toolbox for Java (5769-JC1) to develop client-based applications and applets that access AS/400 resources.

AS/400 Toolbox for Java

The Licensed Product Product for the AS/400 Toolbox for Java has changed in V4R4 to 5769-JC1. This is also known as Modification 2 of the AS/400 Toolbox for Java. It can only be used with AS/400 systems at V4R2 or later for AS/400 Java applications or AS/400 Java client/server applications.

Prior to V4R4, the AS/400 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 AS/400 Toolbox for Java provides a set of Java classes that simplify the access of AS/400 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 AS/400 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 AS/400 system. This is why you must install the OS/400 Host Servers and TCP/IP Connectivity Utilities.

The AS/400 Developer Kit for Java (5769-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 V4R4, you have two JDK options:

- Install option 1 for JDK 1.1.6
- Install option 2 for JDK 1.1.7

The Qshell Interpreter (5769-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 AS/400 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. For V4R4, you do not need to install the Qshell Utilities because they are included with the Qshell Interpreter.

3. 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 16 on page 38 shows 0S/400 - Qshell Interpreter (5769-SS1, option 30).

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769SS1    25      OS/400 - NetWare Enhanced Integration
5769SS1    29      OS/400 - AS/400 Integration for NT
5769SS1    30      OS/400 - QShell Interpreter
5769SS1    31      OS/400 - Domain Name System
5769SS1    32      OS/400 - Directory Services
5769SS1    33      OS/400 - Private Address Space Environment
5769SS1    34      OS/400 - Digital Certificate Manager
5769SS1    35      OS/400 - Cryptographic Service Provider
5769AC3    *BASE   Crypto Access Provider 128-bit for AS/400
5769CB1    *BASE   ILE COBOL for AS/400
5769CB1    1       System/36 Compatible COBOL for AS/400
5769CB1    2       System/38 Compatible COBOL for AS/400
5769CB1    5       OPM COBOL for AS/400
5769CB1    6       *PRV ILE COBOL for AS/400
                                                                    More...
Press Enter to continue.

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

Figure 16. Display Installed Licensed Programs and Product Options

Browse through the various pages of the displayed LPPs and look for all the programs that are previously listed. Make sure that you find **AS/400 Toolbox for Java (5769-JC1)**, as shown in Figure 17.

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769CE1    *BASE   Client Encryption 40-bit
5769CE3    *BASE   Client Encryption 128-bit
5769CM1    *BASE   Communications Utilities for AS/400
5769CX2    *BASE   ILE C for AS/400
5716CX4    *BASE   Visualage C++ for OS/400 BE
5716CX4    1       Visualage C++ for OS/400
5769CX5    *BASE   VisualAge for C++ for AS/400 - Host Component
5769CX5    1       VisualAge for C++ for AS/400 - Windows Client
5648C05    *BASE   IBM Network Station Manager for AS/400
5769DG1    *BASE   IBM HTTP Server for AS/400
5769FW1    *BASE   IBM Firewall for AS/400
2YPTINT    *BASE   A product for Management Central compare and
1YPTINT    *BASE   A product for Management Central compare and
5769JC1    *BASE   AS/400 Toolbox for Java
                                                                    More...
Press Enter to continue.

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

Figure 17. Display Installed Licensed Programs (5769-JC1)

Look for **AS/400 Developer Kit for Java (5769-JV1)**. Figure 18 on page 39 shows that the *BASE AS/400 Developer Kit for Java is installed with both JDK 1.1.6 (option 1) and JDK 1.1.7 (option 2).

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769JS1    *BASE   Job Scheduler for AS/400
5769JV1    *BASE   AS/400 Developer Kit for Java
5769JV1    1       Java Developer Kit 1.1.6
5769JV1    2       Java Developer Kit 1.1.7
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
5769LNT    5       LotusScript Extension ToolKit
5769LNT    6       HiTest C API
5769LNT    7       Advanced Services
5769NC5    *BASE   NetQuestion for AS/400
5700NT1    *BASE   Native Tools (NATT) - IBM Internal Tools
                                                                      More...
Press Enter to continue.

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

Figure 18. Display Installed Licensed Programs (5769-JV1)

If AS/400 Toolbox for Java (5769-JC1) or AS/400 Developer Kit for Java (5769-JV1) are not installed, follow the procedures in Section 3.2, “Manually Installing Java Support on the AS/400 System” on page 41, to install them.

Continue paging down until you find the **TCP/IP Connectivity Utilities for AS/400 (5769-TC1)** LPP as shown in Figure 19.

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769RD1    1       OnDemand Spooled File Archive Feature
5769RD1    2       OnDemand Object Archive Feature
5769RD1    3       OnDemand Record Archive Feature
5769RD1    4       OnDemand AnyStore Feature
5769RG1    *BASE   ILE RPG for AS/400
5769RG1    1       S/36 compatible RPG II for AS/400
5769RG1    2       S/38 compatible RPG III for AS/400
5769RG1    5       RPG for AS/400
5769SA2    *BASE   IBM Integration Services for FSIOP
5769ST1    *BASE   DB2 Query Mgr and SQL DevKit for AS/400
5769TC1    *BASE   TCP/IP Connectivity Utilities for AS/400
5769WP1    *BASE   OfficeVision for AS/400
5769WP1    1       OfficeVision - Text Search Services
5763XD1    *BASE   Client Access/400 Optimized for Windows
                                                                      More...
Press Enter to continue.

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

Figure 19. Display Installed Licensed Programs (Host Servers)

If you plan on using PCs that run Microsoft Windows 95/NT as Java clients or as Java Remote AWT devices for your AS/400 applications, use Client Access

for Windows 95 to provide easy to use AS/400 connectivity features, such as accessing the integrated file system to store Java source code and Java bytecodes from your PC. In this case, look for the **Client Access/400 Optimized for Windows (5763-XD1)** LPP, as shown in Figure 20.

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769RD1    1       OnDemand Spooled File Archive Feature
5769RD1    2       OnDemand Object Archive Feature
5769RD1    3       OnDemand Record Archive Feature
5769RD1    4       OnDemand AnyStore Feature
5769RG1    *BASE   ILE RPG for AS/400
5769RG1    1       S/36 compatible RPG II for AS/400
5769RG1    2       S/38 compatible RPG III for AS/400
5769RG1    5       RPG for AS/400
5769SA2    *BASE   IBM Integration Services for FSIOP
5769ST1    *BASE   DB2 Query Mgr and SQL DevKit for AS/400
5769TC1    *BASE   TCP/IP Connectivity Utilities for AS/400
5769WP1    *BASE   OfficeVision for AS/400
5769WP1    1       OfficeVision - Text Search Services
5763XD1    *BASE   Client Access/400 Optimized for Windows
                                                                      More...
Press Enter to continue.

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

Figure 20. Display Installed Licensed Programs (5763-XD1)

Starting with Version 4 Release 1 (V4R1) of OS/400, the enhanced version of the Windows 3.1 client and the Windows 95/NT client are now part of the Client Access/400 Windows Family Base (5769-XW1). Make sure that this LPP is installed, as shown in Figure 21.

```
                    Display Installed Licensed Programs
                                                          System:   AS400SYS
Licensed  Product
Program   Option   Description
5769XE1    *BASE   Client Access/400 Express for Windows
5769XW1    *BASE   Client Access/400 Windows Family Base
5769XY1    *BASE   Client Access Base Family
5769XZ1    *BASE   OS/2 Warp Server for AS400 (WS400)
0TOOL00    *BASE   STT Tools (*BASE)
0TOOL00    1       STT tools (01) General
0TOOL00    2       STT Tools (02) IPL History Collection
0TOOL00    3       STT Tools (03) Performance Reporter
0TOOL00    4       STT Tools (04) Firewall/Security









                                                                      Bottom
Press Enter to continue.

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

Figure 21. Display Installed Licensed Programs (5769-XW1)

If all of the required LPPs and OS/400 options are installed on your system, you can skip the next section. Otherwise, go through the following steps to manually install any missing OS/400 options or LPPs on your system.

3.2 Manually Installing Java Support on the AS/400 System

In this section, we show you the steps necessary to install one or more of the software functions that run Java on your AS/400 system. First, we guide you step-by-step through the installation process. Then, we cover loading and applying the latest PTF Cumulative Package.

3.2.1 Installing OS/400 Options and Licensed Programs

To install OS/400 options and LPPs, perform these steps:

1. Return to the Work with Licensed Programs menu by using the **F3** or the **F12** key from the Display Installed Licensed Programs display.
2. Enter option `11` on the Work with Licensed Programs menu, as shown in Figure 22.

```
LICPGM                  Work with Licensed Programs
                                                        System:   AS400SYS
Select one of the following:

  Manual Install
     1. Install all

  Preparation
     5. Prepare for install

  Licensed Programs
    10. Display installed licensed programs
    11. Install licensed programs
    12. Delete licensed programs
    13. Save licensed programs


                                                                    More...
Selection or command
===> 11

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F16=AS/400 Main menu
```

Figure 22. Work with Licensed Programs Display

3. Press the **Enter** key. The Install Licensed Programs display appears as shown in Figure 23 on page 42.

```
                        Install Licensed Programs
                                                          System:   AS400SYS
Type options, press Enter.
  1=Install

       Licensed  Product
Option Program   Option   Description

       5769SS1           OS/400 - Library QGPL
       5769SS1           OS/400 - Library QUSRSYS
       5769SS1    1      OS/400 - Extended Base Support
       5769SS1    2      OS/400 - Online Information
       5769SS1    3      OS/400 - Extended Base Directory Support
       5769SS1    4      OS/400 - S/36 and S/38 Migration
       5769SS1    5      OS/400 - System/36 Environment
       5769SS1    6      OS/400 - System/38 Environment
       5769SS1    7      OS/400 - Example Tools Library
       5769SS1    8      OS/400 - AFP Compatibility Fonts
       5769SS1    9      OS/400 - *PRV CL Compiler Support
       5769SS1    11     OS/400 - S/36 Migration Assistant
                                                                      More...
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks

(C) COPYRIGHT IBM CORP. 1980, 1999.
```

Figure 23. Install Licensed Programs Display

4. On the appropriate line in the list, enter a `1` next to OS/400 - Host Servers (5769-SS1) if this option is not already installed on your system.

```
                        Install Licensed Programs
                                                          System:   AS400SYS
Type options, press Enter.
  1=Install

       Licensed  Product
Option Program   Option   Description

  1    5769SS1    12     OS/400 - Host Servers
       5769SS1    13     OS/400 - System Openness Includes
       5769SS1    14     OS/400 - GDDM
       5769SS1    15     OS/400 - Common Programming APIs Toolkit
       5769SS1    16     OS/400 - Ultimedia System Facilities
       5769SS1    17     OS/400 - PSF/400 Fax Support
       5769SS1    18     OS/400 - Media and Storage Extensions
       5769SS1    20     OS/400 - Advanced 36
       5769SS1    21     OS/400 - Extended NLS Support
       5769SS1    22     OS/400 - ObjectConnect
       5769SS1    23     OS/400 - OptiConnect
       5769SS1    25     OS/400 - NetWare Enhanced Integration
                                                                      More...
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks
```

Figure 24. Install Licensed Programs (OS/400 - Host Servers)

5. Use the **Page Down** or the **Scroll Up** key until you see OS/400 - Qshell Interpreter (5769-SS1), as shown in Figure 25 on page 43.

```
                          Install Licensed Programs
                                                              System:   AS400SYS
Type options, press Enter.
  1=Install

       Licensed  Product
Option Program   Option   Description

       5769SS1    26      OS/400 - DB2 Symmetric Multiprocessing
       5769SS1    27      OS/400 - DB2 Multisystem
       5769SS1    29      OS/400 - AS/400 Integration for NT
  1    5769SS1    30      OS/400 - QShell Interpreter
       5769SS1    31      OS/400 - Domain Name System
       5769SS1    32      OS/400 - Directory Services
       5769SS1    33      OS/400 - Private Address Space Environment
       5769SS1    34      OS/400 - Digital Certificate Manager
       5769SS1    35      OS/400 - Cryptographic Service Provider
       5769SS1    36      OS/400 - PSF/400 1-20 IPM Printer Support
       5769SS1    37      OS/400 - PSF/400 1-45 IPM Printer Support
       5769SS1    38      OS/400 - PSF/400 Any Speed Printer Support
                                                                        More...
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks
```

Figure 25. Install Licensed Programs (OS400 - Qshell Interpreter)

Enter a `1` to select the OS/400 - Qshell Interpreter (5769-SS1) option if it is not already installed on your system. This option provides a text-based command entry environment that allows you to use most Java standard commands from any AS/400 workstation the same as you would from a DOS command entry window on a PC. This is a required option because an OS/400 style command is not provided for every Java command.

Note: If you need to run any other Java command, such as **javadoc** to produce standard documentation from a Java program, or **jar** to package one or more Java classes together, or any other Java standard command, you must use OS/400 Java Qshell Interpreter support. *Do not forget to install it*.

In V4R4 of OS/400, the current Java CL commands are:

- **CRTJVAPGM command**: Compiles a Java program on the AS/400 system. This command converts Java bytecodes that are produced by any Java compiler into an executable AS/400 program. To run the CRTJVAPGM command, you must specify the name of a Java class file that contains Java bytecodes. You can produce Java bytecodes on any PC-based Java compiler and copy them to the AS/400 integrated file system. Or, you can run the **javac** command on the AS/400 system by using the OS/400 Java Qshell Interpreter support. Then, create the AS/400 executable program from an OS/400 command entry display.
- **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.
- **RUNJVA (or JAVA) command**: Runs a Java program on the AS/400 system. This command is equivalent to the standard **java** command. To run a Java program on the AS/400 system, you can choose to use the OS/400 RUNJVA command or the JAVA command from an OS/400 command entry display and benefit from the OS/400 style prompts and online help. Or, use

the standard Java style **java** command using the OS/400 Java Qshell Interpreter support.

- **DSPJVAPGM command**: Displays some details about the AS/400 executable Java program that was produced by the CRTJVAPGM command.
- **DLTJVAPGM command**: Deletes the AS/400 executable Java program and removes the link from the associated class file.

6. Use the **Page Down** or the **Scroll Up** key until you see the two LPPs that are required for running Java within the AS/400 system. These are AS/400 Toolbox for Java (5769-JC1) and AS/400 Developer Kit for Java (5769-JV1). Enter a `1` next to each one on these two LPPs to select them for installation, as shown in Figure 26. Do not forget to select which JDK option to install. You can install either JDK 1.1.6 (option 1), JDK 1.1.7 (option 2) or both.

   Note: If you install both JDK 1.1.6 and JDK 1.1.7, but do not specify which version you want to run, then JDK 1.1.7 is your default because it is the most recent JDK.

```
                        Install Licensed Programs
                                                          System:   AS400SYS
Type options, press Enter.
  1=Install

       Licensed  Product
Option Program   Option   Description

  1    5769JC1    *BASE   AS/400 Toolbox for Java
       5769JS1    *BASE   Job Scheduler for AS/400
  1    5769JV1    *BASE   AS/400 Developer Kit for Java
  1    5769JV1    1       Java Developer Kit 1.1.6
  1    5769JV1    2       Java Developer Kit 1.1.7
       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
       5769LNT    5       LotusScript Extension ToolKit
       5769LNT    6       HiTest C API
                                                                      More...
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks
```

Figure 26. Install Licensed Programs

The AS/400 Toolbox for Java (5769-JC1) provides a set of Java classes that are aimed at simplifying the access to AS/400 resources from a Java application. These classes on the AS/400 server allow Java applications on AS/400 to have access to data queues, printer and spooling functions, running OS/400 commands, or calling AS/400 programs. The AS/400 Toolbox for Java classes also provide record-level access to AS/400 files using DDM, reading from and writing to PC-like files stored on the integrated file system, and JDBC access to the DB2/400 database.

AS/400 Toolbox for Java

The Licensed Product Product for the AS/400 Toolbox for Java has changed in V4R4 to 5769-JC1. This is also known as Modification 2 of the AS/400 Toolbox for Java. It can only be used with AS/400 systems at V4R2 or later for AS/400 Java applications or AS/400 Java client/server applications.

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

You can also use the AS/400 Toolbox for Java classes on the client to develop Java applications or applets that access AS/400 resources. Refer to *Building AS/400 Client Server Applications with Java*, SG24-2152, for a complete detailed description and working examples on how to use the AS/400 Toolbox for Java classes to develop Java applications that access the AS/400 system.

The AS/400 Developer Kit for Java (5769-JV1) provides all of the Java classes as defined in the Java Development Kit and published by Sun Microsystems, Inc. This set of application programming interfaces (APIs) allows for running any Java compliant program on the AS/400 system. This LPP requires OS/400 Version 4 Release 2 (V4R2) or later.

7. Scroll down the list until you find **TCP/IP Connectivity Utilities for AS/400 (5769-TC1)**. Enter `1` next to this LPP to install if it is not already installed on your AS/400 system. This is shown in Figure 27.

```
                          Install Licensed Programs
                                                              System:   AS400SYS
Type options, press Enter.
  1=Install

        Licensed  Product
Option  Program   Option   Description

        5769RG1    5       RPG for AS/400
        5769RG1    6       *PRV ILE RPG for AS/400
        5769SA2    *BASE   IBM Integration Services for FSIOP
        5769ST1    *BASE   DB2 Query Mgr and SQL DevKit for AS/400
  1     5769TC1    *BASE   TCP/IP Connectivity Utilities for AS/400
        5769WP1    *BASE   OfficeVision for AS/400
        5769WP1    1       OfficeVision - Text Search Services
        5763XD1    *BASE   Client Access/400 Optimized for Windows
        5769XE1    *BASE   Client Access/400 Express for Windows
        5769XW1    *BASE   Client Access/400 Windows Family Base
        5769XY1    *BASE   Client Access Base Family
        5769XZ1    *BASE   OS/2 Warp Server for AS400 (WS400)
                                                                          Bottom
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks
```

Figure 27. Install Licensed Programs (TCP/IP Connectivity Utilities for AS/400)

If you plan on using PCs that run Microsoft Windows 95 or Windows NT Workstation as Java clients or as Java Remote AWT clients for your AS/400 server-based Java applications, you may consider installing Client Access/400 Optimized for Windows (5763-XD1) on your AS/400 system. To do this, scroll down the Install Licensed Programs display until you see the Client Access for Windows 95/NT LPP, as shown in Figure 28 on page 46.

```
                          Install Licensed Programs
                                                              System:   AS400SYS
Type options, press Enter.
  1=Install

       Licensed  Product
Option Program   Option   Description

       5769RG1    5       RPG for AS/400
       5769RG1    6       *PRV ILE RPG for AS/400
       5769SA2    *BASE   IBM Integration Services for FSIOP
       5769ST1    *BASE   DB2 Query Mgr and SQL DevKit for AS/400
       5769TC1    *BASE   TCP/IP Connectivity Utilities for AS/400
       5769WP1    *BASE   OfficeVision for AS/400
       5769WP1    1       OfficeVision - Text Search Services
  1    5763XD1    *BASE   Client Access/400 Optimized for Windows
       5769XE1    *BASE   Client Access/400 Express for Windows
       5769XW1    *BASE   Client Access/400 Windows Family Base
       5769XY1    *BASE   Client Access Base Family
       5769XZ1    *BASE   OS/2 Warp Server for AS400 (WS400)
                                                                        Bottom
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks
```

Figure 28. Install Licensed Programs (Client Access/400 Optimized for Windows)

Enter a `1` next to the Client Access/400 Optimized for Windows (5763-XD1) entry on the display if this LPP is not already installed on your AS/400 system. Then, scroll down the display by using the **Page Down** key or the **Scroll Up** key until you see Client Access/400 Windows Family Base (5769-XW1). Enter a `1` on the corresponding line to select this LPP for installation if it is not currently installed on your machine. Figure 29 shows this selection.

```
                          Install Licensed Programs
                                                              System:   AS400SYS
Type options, press Enter.
  1=Install

       Licensed  Product
Option Program   Option   Description

       5769RG1    5       RPG for AS/400
       5769RG1    6       *PRV ILE RPG for AS/400
       5769SA2    *BASE   IBM Integration Services for FSIOP
       5769ST1    *BASE   DB2 Query Mgr and SQL DevKit for AS/400
       5769TC1    *BASE   TCP/IP Connectivity Utilities for AS/400
       5769WP1    *BASE   OfficeVision for AS/400
       5769WP1    1       OfficeVision - Text Search Services
       5763XD1    *BASE   Client Access/400 Optimized for Windows
       5769XE1    *BASE   Client Access/400 Express for Windows
  1    5769XW1    *BASE   Client Access/400 Windows Family Base
       5769XY1    *BASE   Client Access Base Family
       5769XZ1    *BASE   OS/2 Warp Server for AS400 (WS400)
                                                                        Bottom
F3=Exit   F11=Display status/release   F12=Cancel   F19=Display trademarks
```

Figure 29. Install Licensed Programs (Client Access/400 Windows Family Base)

You have now completed selecting the required OS/400 options and LPPs. Press **Enter** to see the Confirm Install of Licensed Programs display shown in Figure 30.

```
                    Confirm Install of Licensed Programs
                                                          System:   AS400SYS
Press Enter to confirm your choices for 1=Install.
Press F12 to return to change your choices.

Licensed  Product
Program   Option   Description
5769SS1    12      OS/400 - Host Servers
5769SS1    30      OS/400 - QShell Interpreter
5769JC1    *BASE   AS/400 Toolbox for Java
5769JV1    *BASE   AS/400 Developer Kit for Java
5769JV1    1       Java Developer Kit 1.1.6
5769JV1    2       Java Developer Kit 1.1.7
5769TC1    *BASE   TCP/IP Connectivity Utilities for AS/400
5763XD1    *BASE   Client Access/400 Optimized for Windows
5769XW1    *BASE   Client Access/400 Windows Family Base




                                                                      Bottom
F11=Display status/release   F12=Cancel
```

Figure 30. Confirm Install of Licensed Programs Display

On this display, most of the OS/400 options and LPPs are at the V4R4 level. However, as we have previously mentioned, the AS/400 Toolbox for Java LPP can be installed on any AS/400 system that runs OS/400 V3R2, V3R7, V4R1, V4R2, or V4R3. This is depicted by the name of the LPP (5769-JC1) that conforms to OS/400 V3R2 standards and by the V3R2M0 information that is shown in the *Installed Release* column.

The Client Access/400 Optimized for Windows LPP is now at the V3R1M3 level. You can install this level on any AS/400 system that runs OS/400 V3R1, V3R2, V3R6, V3R7, V4R1, V4R2, or V4R3.

Note: *Be aware that some Operations Navigator functions may not be available on the CISC systems.* The Enhanced Client for Windows 3.1 became available as a member of the AS/400 Client Access Family for Windows with OS/400 V4R1.

Note: If one or more options or LPPs are not installed on your system, the corresponding Installed Release information is blank.

Press **Enter** to see the Install Options display shown in Figure 31 on page 48.

```
                              Install Options
                                                          System:   AS400SYS
Type choices, press Enter.

  Installation device  . . .   OPT01        Name

  Objects to install . . . .   1            1=Programs and language objects
                                            2=Programs
                                            3=Language objects

  Automatic IPL  . . . . . .   N            Y=Yes
                                            N=No









F3=Exit   F12=Cancel
```

Figure 31. PTF Install Options

Enter `OPT01` (or the name of the CD-ROM drive on your AS/400 system if it differs from OPT01) on the Installation device prompt and leave all other options as they are. Make sure that you insert the OS/400 Volume 1 in the CD-ROM drive. Press the **Enter** key. This starts the installation process on your system. If your distribution media is tape, use the name of the tape device for the Installation device (for example, TAP01).

3.2.2 Installing the Cumulative PTF Package

When the installation completes, you must install the latest cumulative PTF package to load and apply all of the required PTFs to the newly installed OS/400 options and LPPs.

To proceed with installing the PTFs, complete this process:

1. Use the **F3 (Exit)** key or the **F12 (Cancel)** key to return to the AS/400 Main Menu.
2. Enter `GO PTF` on the AS/400 Main Menu command line, as shown in Figure 32 on page 49.

```
MAIN                              AS/400 Main Menu
                                                          System:   AS400SYS
Select one of the following:

     1. User tasks
     2. Office tasks
     3. General system tasks
     4. Files, libraries, and folders
     5. Programming
     6. Communications
     7. Define or change the system
     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access/400 tasks

    90. Sign off

Selection or command
===> GO PTF

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 32. AS/400 Main Menu Display

3. Press the **Enter** key. The Program Temporary Fix menu is shown in Figure 33.

```
PTF                           Program Temporary Fix
                                                          System:   AS400SYS
Select one of the following:

     1. Load a program temporary fix
     2. Apply a program temporary fix
     3. Copy a program temporary fix
     4. Remove a program temporary fix
     5. Display a program temporary fix
     6. Order a program temporary fix
     7. Install a program temporary fix from a list
     8. Install program temporary fix package

    70. Related commands




Selection or command
===> 8

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F16=AS/400 Main menu
(C) COPYRIGHT IBM CORP. 1980, 1998.
```

Figure 33. Program Temporary Fix Display

4. On the Program Temporary Fix menu, enter option 8 (Install program temporary fix package). Then, press the **Enter** key to see the Install Options for Program Temporary Fixes display shown in Figure 34 on page 50.

```
                Install Options for Program Temporary Fixes
                                                        System:   AS400SYS
Type choices, press Enter.

  Device  . . . . . .  .  OPT01       Name, *SERVICE

  Automatic IPL . . . . . .  Y        Y=Yes
                                      N=No

  Restart type  . . . . . .  *SYS     *SYS, *FULL

  PTF type  . . . . . .  .  1         1=All PTFs
                                      2=HIPER PTFs and HIPER LIC fixes
                                        only
                                      3=HIPER LIC fixes only
                                      4=Refresh Licensed Internal Code

  Other options . . . . . .  N        Y=Yes
                                      N=No



F3=Exit   F12=Cancel
```

Figure 34. Install Options for Program Temporary Fixes

On this display, enter `OPT01` (or the name of your AS/400 system CD-ROM drive or tape device if it differs from OPT01). Keep the default values for all the other options that are shown in the previous figure. Make sure that you use the latest cumulative PTF package volume and place it in the CD-ROM drive (or tape device). Then, press the **Enter** key to start loading and applying the latest cumulative PTF package. If you cannot re-IPL the system at this time, you can change the Automatic IPL value to **N** for No.

Attention

You cannot use any Java-related functions before you power down the system and apply the required PTFs during the following IPL.

After you complete this step, you are ready to install the required Java support on your workstation.

3.3 Installing Java on Your Workstation

Now, you need to install a Java runtime on your workstation. In this redbook, we show you how to install Sun Microsystems, Inc. JDK on a PC that runs the Microsoft Windows 95 operating system.

You may choose to use any other workstation environment that is capable of running a JVM. Such environments include: RS/6000 with AIX workstations, PCs running IBM OS/2 Warp Version 4 or Microsoft Windows NT operating system, Apple Macintosh, or any JVM-enabled network computer, such as IBM Network Station.

This redbook does not provide guidance on how to install a Java environment on any of these systems, nor does it provide help on how to install a Web browser or to configure TCP/IP communications.

You must have at least one workstation enabled for Java in your configuration. Java is a programming language designed to develop applications that you can easily deploy in a network computing environment.

3.3.1 Downloading Sun Microsystems, Inc. JDK from the Internet

To install a Java environment on your workstation, you must obtain the required software. You can download the Java Development Kit free of charge from the Sun Microsystems, Inc. Internet site.

If you have several workstations on which you want to install the JDK, you may choose to obtain the required software on a CD-ROM. You can purchase a CD-ROM from Sun Microsystems, Inc. For more details on JDK 1.1, visit Sun Microsystems, Inc. Internet site at this URL (also shown in Figure 35): http://www.javasoft.com/products/jdk/1.1/

Figure 35. Sun Microsystems, Inc. JDK 1.1 Home Page

To download the JDK from the Internet, you must start your Web browser and go to the previous URL. In this redbook, we use Netscape for our examples.

Note: The JDK that we used on our Windows 95 PC was JDK 1.1.7B. In the future, there will be later releases of the JDK (for example, JDK 1.1.8). In the documentation that Sun Microsystems, Inc. provides, it states that these 1.1.n releases will be upward compatible. When a later release becomes available for your PC, it should work the same or similar to the JDK 1.1.7 examples that are shown here.

From this site, simply follow the instructions as they are listed on the HTML page (shown in Figure 35). For our example, click on **JDK 1.1 Win32 Release** to get to the Downloading the Java™ Development Kit Version 1.1.7B section. Then, follow their steps to download and unpack the JDK software, as shown in Figure 36.

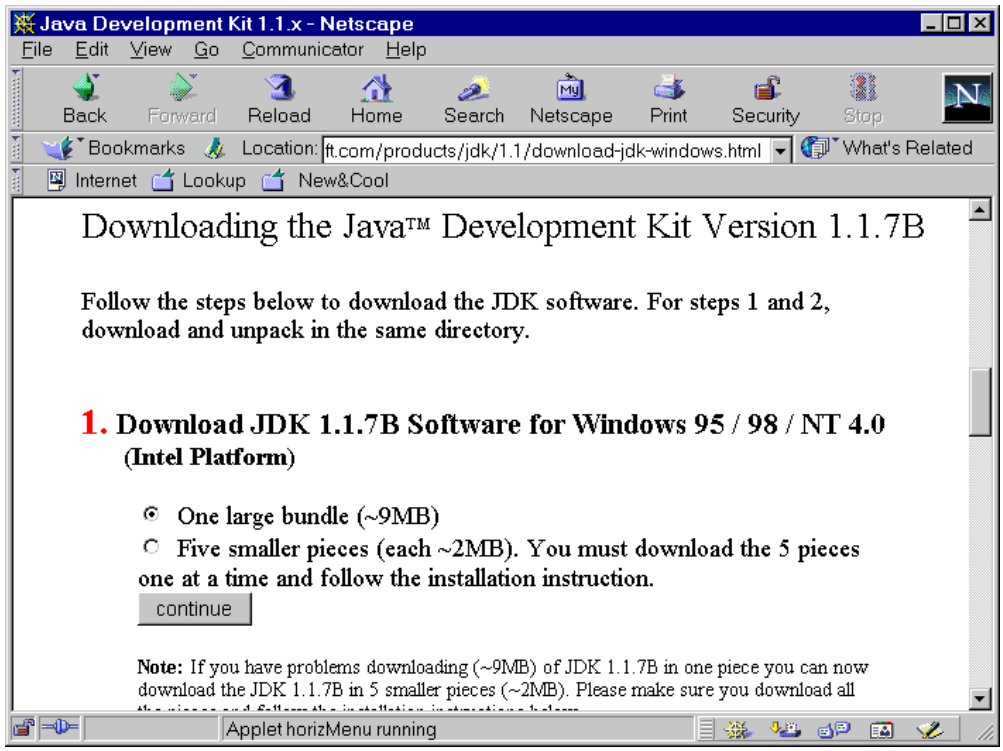


Figure 36. Downloading JDK 1.1.7B from the Internet

Note: Carefully read the information notice you receive relating to U.S. Export Control Terms and Conditions. You may not be eligible to download the JDK. This notice is shown in Figure 37 on page 53.

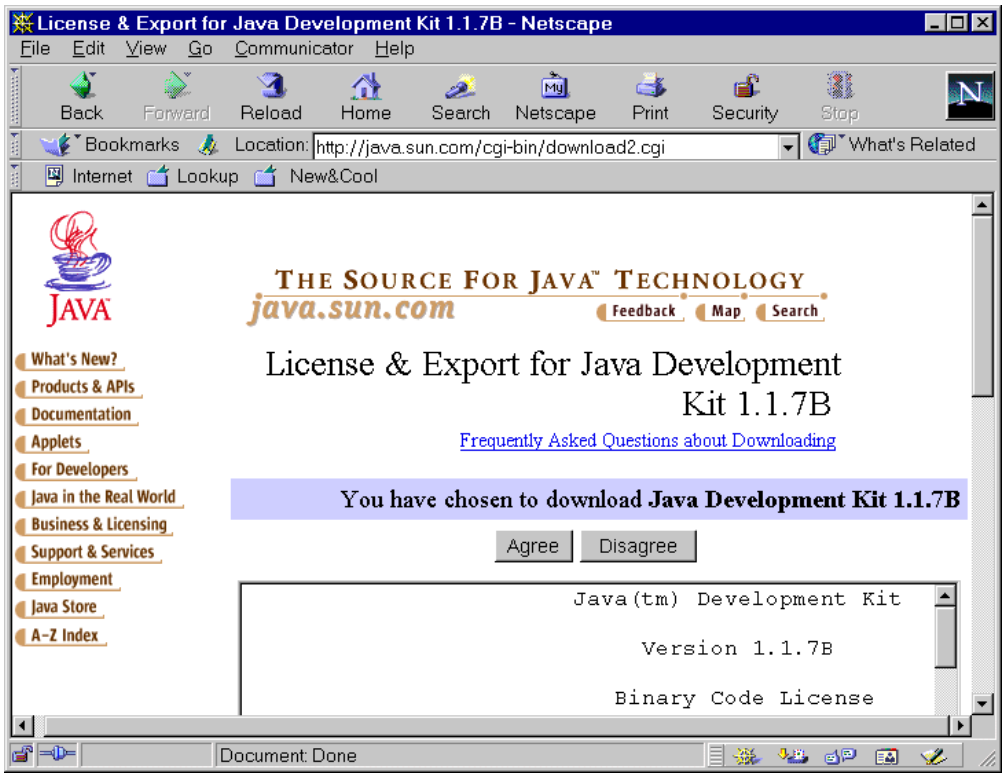


Figure 37. License and Export Conditions

If you are eligible to download the JDK and agree to the terms of the license agreement, click on the **Agree** push button. The JDK Download Page is shown in Figure 38 on page 54.

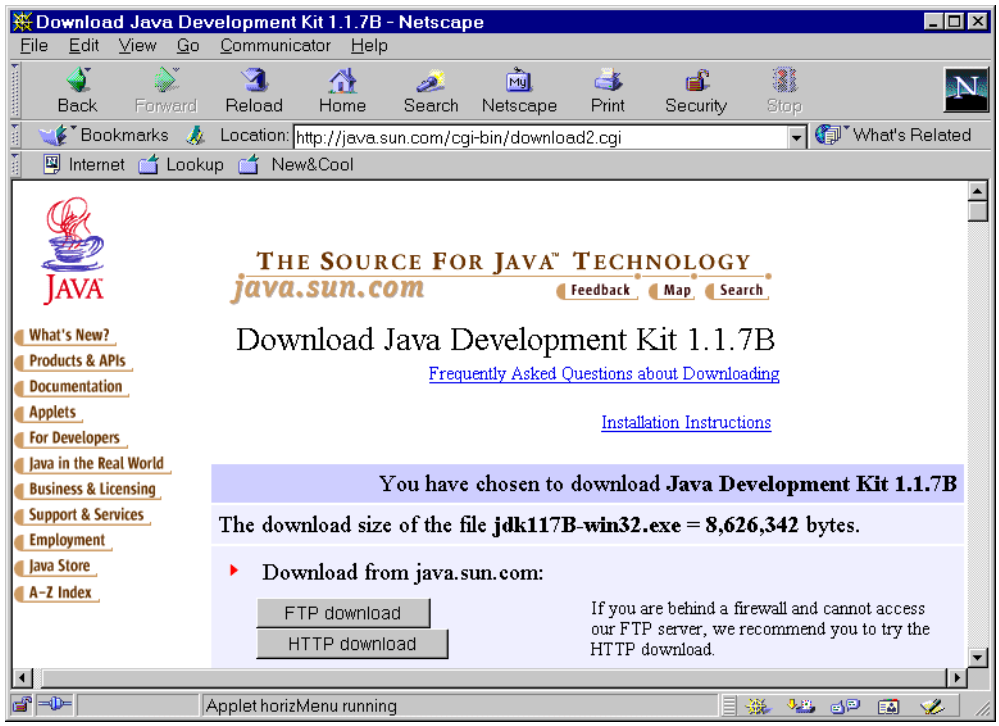


Figure 38. JDK Download Page (Windows Version)

Click on the **FTP download** or **HTTP download** push button to copy the installation program, named jdk117B-win32.exe, to the hard disk of your PC. You are prompted to specify a path name (directory) and file name for saving the file (see Figure 39). Specify a directory name of your choice and keep the proposed program name.

Usually temporary files are saved to the following directory tree:

```
C:\WINDOWS\TEMP\
```

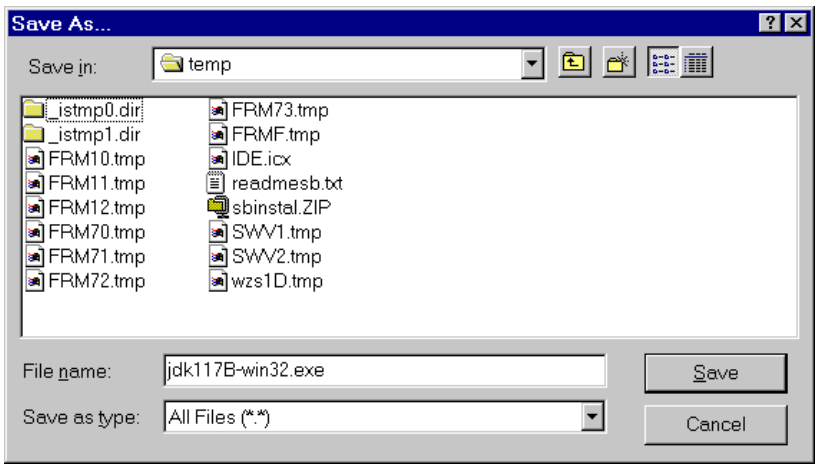


Figure 39. Directory Trees

Then, click on the **Save** push button to begin downloading the file to your PC. This takes several minutes, depending on the speed of your Internet connection link, because this file is 8.6 MB.

After copying the jdk117B-win32.exe, return to the Java Development Kit page at this location: http://www.javasoft.com/products/jdk/1.1/

You may use the **Back** push button from your browser toolbar to do so.

Again, simply follow the instructions as they are listed on the HTML page:

1. Click on the **latest JDK 1.1 documentation** link. This takes you to the JDK Documentation page.
2. Scroll down to download the HTML documentation, as shown in Figure 40.
3. Select **ZIP file for Windows** and press the **continue** push button.

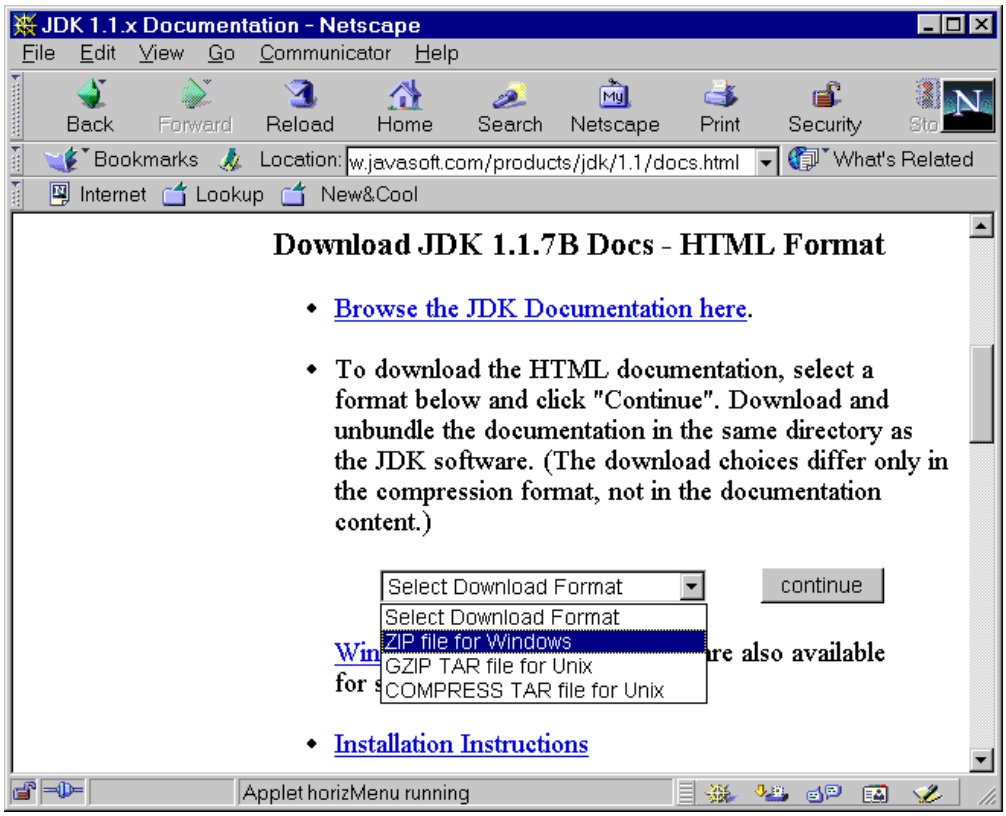


Figure 40. Downloading JDK 1.1 Documentation

This takes you to the Download Java Development Kit Documentation 1.1.7B page, as shown in Figure 41 on page 56.

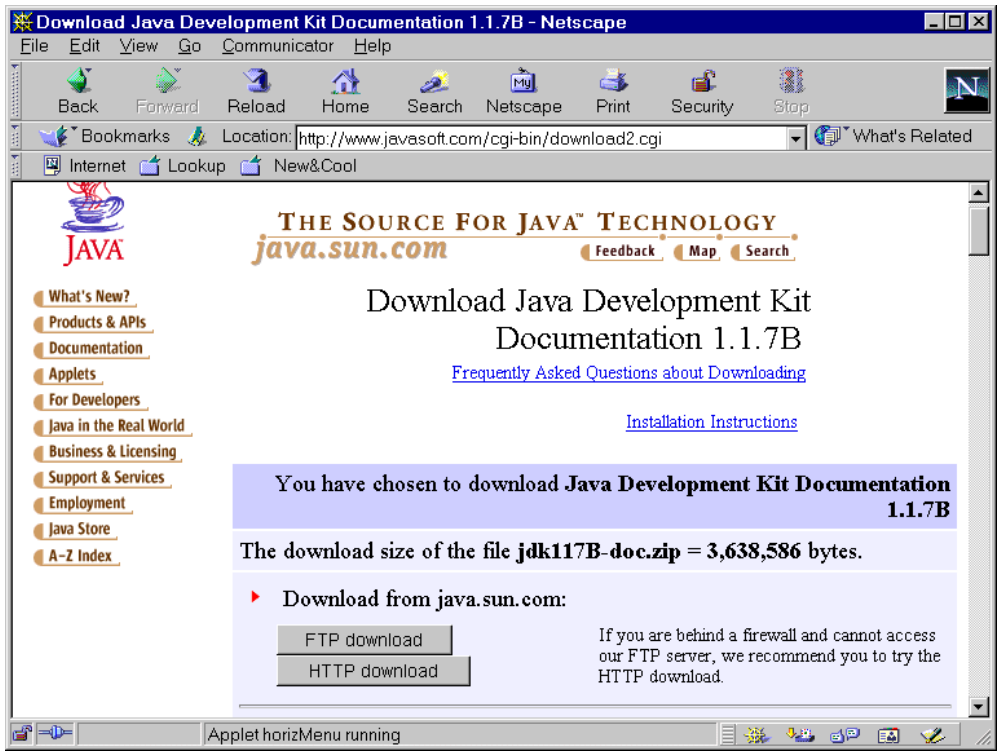


Figure 41. JDK Download Page (Documentation)

4. Click on the **FTP download** or **HTTP download** push button to copy the install program, named jdk117B-doc.zip, to the hard disk of your PC.

   You may receive a warning message: `Unknown File Type`. Ignore the message by clicking on the **Save File** push button.

5. Then, you are prompted to specify a path name (folder) and file name for saving the file, as shown in Figure 42.

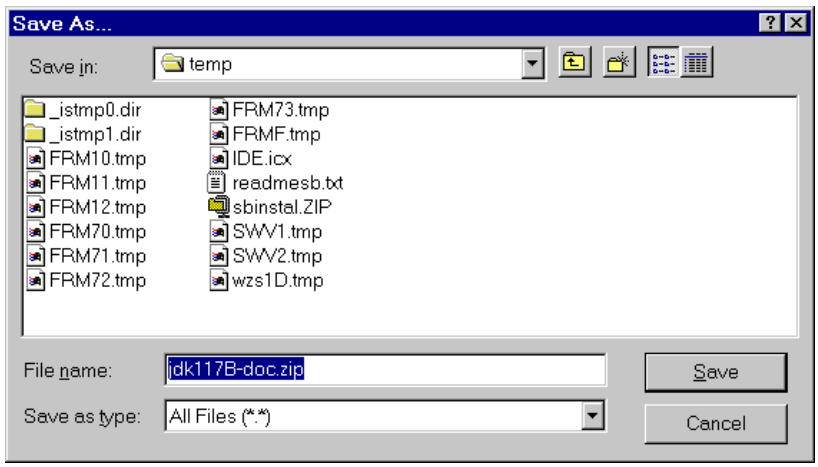


Figure 42. Downloading the zip File to the Correct Folder

6. Specify the folder name and the file name of your choice. Usually, temporary files are saved in the TEMP folder:

   `C:\WINDOWS\TEMP\`

7. Keep the proposed file name for further reference. Then, click on the **Save** push button to begin copying the documentation to your PC. This takes several minutes, depending on the speed of your Internet connection link, because this file is 3.6 MB.

You are now ready to begin installing the JDK and associated documentation on your PC. Start the Windows Explorer and find the two files that you just downloaded. If you used the proposed folder and file names, look for the program **C:\WINDOWS\TEMP\ jdk117B-win32.exe**. Double-click on it to run the installation program. This creates a folder called JDK1.1.7B on your disk and installs all the files and subfolders that compose the JDK.

Then, look for the documentation self extracting program. Again, if you used the proposed folder and file names, look for **C:\WINDOWS\TEMP\ jdk117B-doc.zip**. Double-click on it to start installing the documentation. Make sure you first install the JDK classes and then the documentation. The documentation self-extracting file automatically creates a subfolder named **docs** in the previously created JDK1.1.7B folder.

This completes installing the JDK and related documentation on your PC.

Important

The Java core classes of Sun Microsystems, Inc. JDK 1.1.7B are contained in a compressed (ZIP) file named classes.zip in JDK1.1.7B\LIB\.

Do not UNZIP this file! It must remain zipped for the compiler and interpreter to access the class files within it properly. This file contains all of the compiled class files for the JDK.

3.4 Setting Up the Environment

In this section, we guide you through the necessary steps to set up the proper Java environment both on your workstation and on the AS/400 system.

3.4.1 Setting Up the Environment on Your Workstation

To set up the Java environment on your workstation, you need to set the PATH and CLASSPATH environment variables. This example shows you how to set up your Java environment on a PC. However, you can run the JDK without modifying any system environment variables, such as PATH or CLASSPATH, or the autoexec.bat file.

Note: The path variable is a convenience to you and is not necessary to set.

If you choose not to modify these environment variables, you must specify the correct path every time you want to run any Java command, such as **java**, **javac**, or **javadoc** from a DOS session. For example, if you want to compile a Java program without setting the path, named myclass.java, you need to enter this command at the DOS prompt every time you run a program:

```
C:\WINDOWS> C:\JDK1.1.7\BIN\javac myclass.java
```

This can become tedious and error prone. Therefore, you usually want to set up the proper path information in your autoexec.bat file. Once you set up the path information in your autoexec.bat file, you can compile your program by simply entering this command at the DOS prompt:

```
C:\WINDOWS> javac myclass.java
```

Now, you need to set the PATH variable and CLASSPATH variable.

3.4.1.1 Setting Up the PATH Variable

To change the path information in your autoexec.bat file, complete these steps:

1. Start a text editor, such as WordPad or Notepad.
2. From the File menu, open **c:\autoexec.bat**.

   Look for the PATH statement.

   Note: The PATH statement is a series of directories that are separated by semicolons (;). Windows looks for programs in the PATH directories in order, from left to right. For example, in the following PATH statement, we added the Java directory at the end:

   ```
   SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\;C:\DOS;C:\JDK1.1.7B\BIN
   ```

   Note: The C:\WINDOWS commands are only in your path if your Java file is located in C:\WINDOWS.
3. Put the Java directory at the end of the path statement.
4. Choose **Save** from the File menu to update your autoexec.bat file.
5. Exit from the text editor.

Figure 43 shows the contents of a typical autoexec.bat file.

```
C:\WINDOWS\net start
@ECHO OFF

SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\DOS;%PATH%;C:\JDK1.1.7B\BIN
```

Figure 43. Editing the autoexec.bat Path Variable

Once you have added the proper path information to your autoexec.bat file, you must run it for the new path information to take effect. You should re-boot your system. However, it can run interactively (care must be exercised as to what else will run again) by entering these commands at the DOS prompt:

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

If you get an `Out of environment space` error while running the autoexec.bat file, you must increase the initial environment memory for your DOS session.

To increase the environment memory, perform these steps:

1. Click with the right mouse button somewhere on the MS-DOS Prompt window title (or to click with the left mouse button on the MS-DOS icon on the top left corner of the MS-DOS Prompt window) to show the pop-up menu.
2. On the pop-up menu, select **Properties** to open the MS-DOS Prompt Properties window.
3. Select the **Memory** tab (the one in the middle).
4. On the Memory tab, select **4096** from the drop-down list next to Initial environment as shown in Figure 44.
5. Click on the **Apply** push button to save your changes.
6. Click on the **OK** push button to close the MS-DOS Properties window.

Now, you can re-run the autoexec.bat file to set up the required JDK path information.

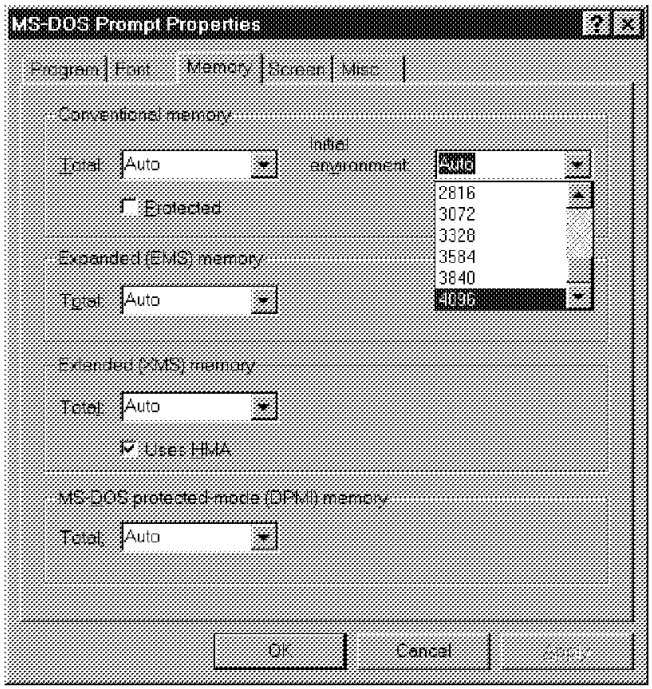


Figure 44. Changing the Initial Memory Environment for DOS

Then, test that the path was changed properly by entering the following command at the DOS prompt:

```
C:\WINDOWS> java
```

If the system does find the **java** command, you get a list of messages describing how to use this command, as shown in Figure 45 on page 60.

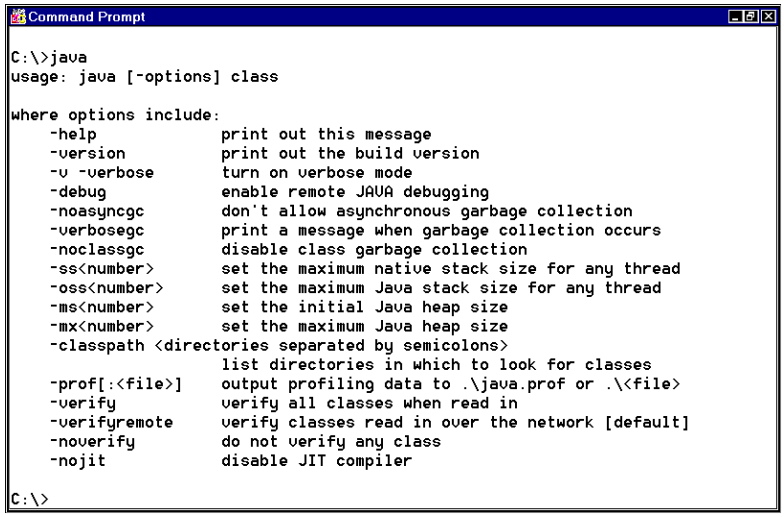


Figure 45. Testing the Path Variable for Java

You have now completed setting up the PATH environment variable.

3.4.1.2 Setting Up the CLASSPATH Variable

The JDK does not require you to change the CLASSPATH environment variable, because it can find the core Java classes as long as they are in the classes.zip file that is located in the LIB subdirectory. The LIB subdirectory is located in the same directory as the BIN subdirectory as indicated in the PATH statement. The JDK automatically appends this CLASSPATH data to what you have explicitly set in your autoexec.bat file:

```
.;[bin]\..\classes;[bin]\..\lib\classes.zip
```

In this example, [bin] is substituted by the absolute path to the JDK1.1.7B\BIN directory.

Therefore, if you keep the BIN and LIB directories at the same directory level (that is, if they have a common parent directory), the Java executables find the classes. You need to set the CLASSPATH only if you move the CLASSES.ZIP file or if you want to load additional class libraries, such as the one you develop or the AS/400 Toolbox for Java.

This completes setting up the Java environment on the workstation.

3.4.2 Setting Up the Environment on the AS/400 System

On the AS/400 system, 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 AS/400 Developer Kit for Java does not require that you have to change the CLASSPATH environment variable. The AS/400 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 LIB subdirectory. The LIB subdirectory is in the same directory as the BIN subdirectory. In V4R4, the classes.zip file replaces the java.zip and sun.zip files.

The AS/400 JVM automatically searches all the classes it needs to load in the LIB subdirectory that is located in the jdk directory on the AS/400 integrated file system.

Therefore, if you do not modify anything in the jdk directory or its associated subdirectories (as shipped with the 5769-JV1 LPP), the Java executables find the classes.

You need to setup the CLASSPATH environment variable if you want to load additional class libraries, such as the one that you develop yourself or the AS/400 Toolbox for Java.

The AS/400 Toolbox for Java classes are stored on the AS/400 integrated file system in two files, named jt400.zip and jt400.jar. These two files are located in this directory tree:

```
/QIBM/ProdData/HTTP/Public/jt400/lib/
```

To use the AS/400 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 AS/400 environment depends on whether you want to run **java** from the Qshell Interpreter, by using Command Language (CL) commands (for example, RUNJVA), or both. In this section, we provide details on how to do this.

3.4.2.1 Creating a Symbolic Link

In these examples, we use a symbolic link. This link is used to simplify setting up the AS/400 environment. We do this by entering the following command on an AS/400 command entry line:

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

An example of adding the symbolic link AS400JT is shown in Figure 46.

```
                              Add Link (ADDLNK)

Type choices, press Enter.

Object . . . . . .   '/QIBM/ProdData/HTTP/Public/jt400'

New link . . . . . .   AS400JT

Link type  . . . . . .   *SYMBOLIC     *SYMBOLIC, *HARD









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

Figure 46. Adding a Symbolic Link

Now, rather than entering /QIBM/ProdData/HTTP/Public/jt400 each time, we can use AS400JT instead.

3.4.3 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.

3.4.3.1 Setting the 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 47.

```
                          Run Java Program (RUNJVA)

 Type choices, press Enter.

 Class  . . . . . . . . . . . . .   MyClass

 Parameters . . . . . . . . . . .   *NONE



               + for more values



 Classpath  . . . . . . . . . . .   'QIBM/ProdData/HTTP/Public/jt400/lib/jt400.z
ip'




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

Figure 47. 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.

3.4.3.2 Setting the CLASSPATH Environment Variable

The CLASSPATH environment variable is available and you can use it to set the CLASSPATH variable. 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

This environment variable is only in effect for the session in which it is set. If you prefer using this option, you can set your AS/400 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 AS/400 Toolbox for Java classes to be found:

```
ADDENVVAR ENVVAR(CLASSPATH) VALUE('AS400JT/lib/jt400.zip:
AS400JT/utilities')
```

Now, when a Java CL command is used (for example, 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

3.4.4 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 AS/400 system. 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

3.4.4.1 Setting the 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/jt400.zip:AS400JT/utilities
```

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.

3.4.4.2 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/jt400.zip:AS400JT/utilities myjavapgm
```

3.4.4.3 Setup for Individual Users when Using Qshell Interpreter

When using the Qshell Interpreter, you can control the CLASSPATH information by individual user or system wide. In this section, we show 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 you want to access the AS/400 Toolbox for Java classes. First, you need to create the user home directory. You can do this by entering this command on the AS/400 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.

```
                    Create Directory (CRTDIR)

Type choices, press Enter.

Directory  . . . . . . . . . . .   '/home/myusrprf'

Public authority for data  . . .   *INDIR        Name, *INDIR, *RWX, *RW...
Public authority for object  . .   *INDIR        *INDIR, *NONE, *ALL...
               + for more values
Auditing value for objects . . .   *SYSVAL       *SYSVAL, *NONE, *USRPRF...

                                                                      Bottom
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys Parameter DIR required.
```

Figure 48. Creating an Integrated File System Directory on AS/400 System

This creates a subdirectory, named "myusrprf", in the **home** directory. Then, you need to create the .profile file for that user profile. You can do this by entering the this command on the AS/400 command entry line:

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

The EDTF command invokes a stream file editor that is provided in the QUSRTOOL library, which is similar to Source Entry Utility (SEU). See member TGPAESFI in file QUSRTOOL/QATTINFO for directions on installing this editor on your system.

Then, add this line in the .profile file for this user:

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

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 your 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 AS/400 Toolbox for Java classes and your own Java classes, as long as your classes are stored in your home directory. Figure 49 shows the EDTF window.

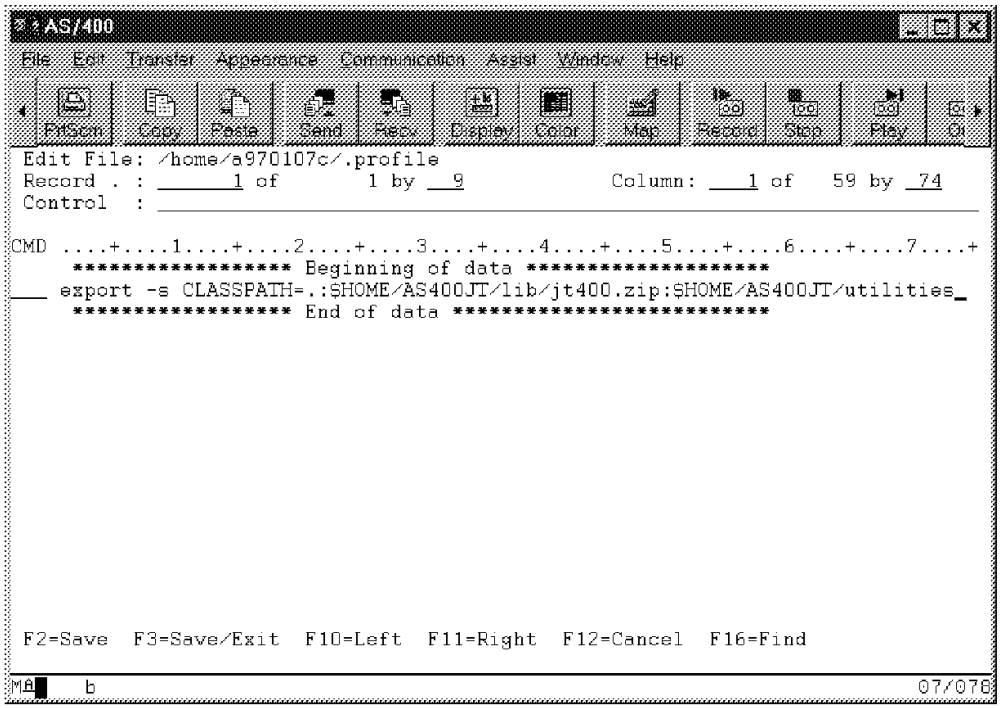


Figure 49. Using EDTF to Create .profile File

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 we use the $HOME parameter to control the path to the AS/400 Toolbox classes, we 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 AS/400 Toolbox for Java classes.

3.4.4.4 Setup for Entire AS/400 System Using Qshell Interpreter

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 AS/400 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/jt400.zip:/AS400JT/utilities
```

Similar to the individual user profile setup, we use a symbolic link, named AS400JT, to simplify the creation of the EXPORT directive and make it possible to access the AS/400 Toolbox for Java classes, the AS400ToolboxInstaller class, and your own classes through a single export statement. In this case, we do not use the user home directory to control the path to the AS/400 Toolbox classes, so we 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 OS/400 integrated file system command.

3.4.5 AS/400 CLASSPATH Recommendation

You can automate the process of setting the CLASSPATH environment variable by creating a CL program similar to Figure 50 and setting it as the Initial Program to Call (INLPGM) in your user profile.

```
MONMSG     MSGID(CPF0000)
             ADDENVVAR  ENVVAR(CLASSPATH) +
                VALUE('.:/+
                :AS400JT/lib/jt400.zip+
                :AS400JT/utilities
            ADDLIBLE   LIB(QJAVA)
             ADDLIBLE   LIB(APILIB) POSITION(*LAST)
```

Figure 50. CL Source for Initial Program

We use the library APILIB for our examples in this redbook. You should 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 to Call, it will run each time you sign on. This ensures that the CLASSPATH is set properly each time.

3.4.5.1 Testing the Java Environment on the AS/400 System

Now, you should create a simple "Hello World" Java application that runs on your AS/400 system to test that you have setup the Java environment on the AS/400 system 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. Logon to your AS/400 system, and press the **Enter** key to access the AS/400 Main Menu.

b. Go to a DOS prompt and make sure that you are in your specified AS/400 integrated file system directory. For this example, we use mydir as the directory name.

c. Enter `notepad HelloAS400World.java` to open Notepad and create a new Java source named "HelloAS400World.java."

d. Enter the source code shown in Figure 51.

Note: Use exact punctuation and capitalization. Both are critical to Java syntax.

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

Figure 51. HelloAS400World Program

e. Exit and save your work to your directory on the AS/400 system.

2. Compile the source by completing these tasks:

a. Enter `qsh` on the AS/400 Main Menu command line. Press **Enter**.

You are now in the Qshell environment, which is like a UNIX environment.

b. Enter `cd /mydir` to get to your directory. Press **Enter**.

c. Try using a few Qshell commands, such as: **ls** (list files, like **dir**), **env** (show environment, like **set**), **find Hello*.*** (similar to **ls**), **hostname**, and so on.

d. Enter the `javac` command without specifying any parameters. Press **Enter**. This shows you the syntax of the command.

e. Enter `javac HelloAS400World.java` on the QSH Command Entry command line to compile your program, as shown in Figure 52 on page 68. Press **Enter**.

```
                                QSH Command Entry

  $
> cd /mydir
  $
> javac
  use: javac [-g][-O][-debug][-depend][-nowarn][-verbose][-classpath path][-now
  rite][-deprecation][-d dir][-J<runtime flag>] file.java...
  $
> javac HelloAS400World.java
  $




===>



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

Figure 52. Compiling a Java Program Using the Qshell Interpreter

The **$** symbol signifies the end of the compile. If there are any errors in your source, they will be printed to the display. Edit your source file to fix any errors.

If you are successful and your source compiles without errors, a file called HellAS400World.class is created inside of your 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. Enter `java -help` on the QSH Command Entry display to see a list of valid parameters for the **java** command. Press **Enter**.
   b. Enter `java -version` to see which JDK level is loaded on your system. In this example, we are using JDK 1.1.7. Press **Enter**.
   c. Enter `java -classpath /mydir HelloAS400World` to run your application.
   d. This message should be displayed: `Hello World from AS/400!` See Figure 53 on page 69.

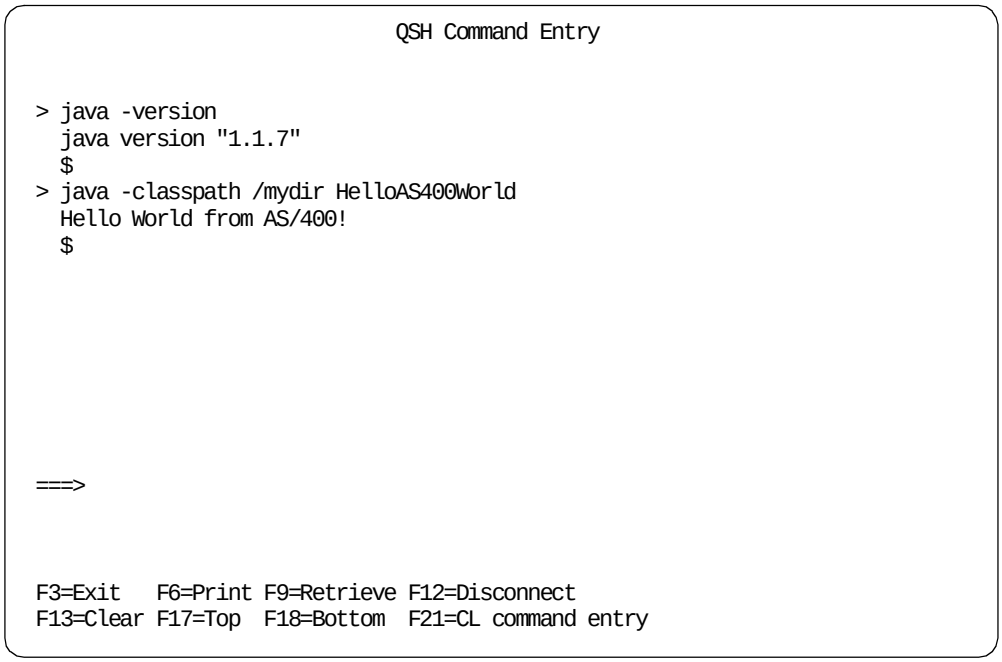


Figure 53. Running a Java Program Using the Qshell Interpreter

e. Try running it again, but this time enter: `java -verbose -classpath /mydir HelloAS400World`. Press **Enter**.

   This shows you 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 AS/400 JVM.

3.4.5.2 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.

In the next section, we show how to install the AS/400 Toolbox for Java on your workstation.

3.4.6 Installing the AS/400 Toolbox for Java on Your Workstation

In this section, we show how to install the AS/400 Toolbox for Java classes on your workstation. Although this is not mandatory, you may want to install the AS/400 Toolbox for Java on your workstation if you are interested in building applications that use Java on your workstation. For example, you may want to build a client/server application that uses Java on both the workstation and the AS/400 system. This set of Java APIs provides easy to use Java classes that allow you to access most AS/400 resources from Java applets or applications running on any Java-enabled workstation. The AS/400 Toolbox for Java provides the following support:

- Connection management and security (connect to the AS/400 system and change password)
- JDBC (relational database access APIs)
- Data queues
- AS/400 Program call
- OS/400 command execution
- Integrated file system stream file access
- Print and spool access
- Record level access (native DDM database access APIs)
- AS/400 messages
- Data conversion between Java data types and AS/400 data types

AS/400 Toolbox for Java (5769-JC1) is an AS/400 LPP and is shipped to you on the OS/400 LPP media (CD-ROMs). You install this LPP on the AS/400 system using standard installation procedures as described in Section 3.2, “Manually Installing Java Support on the AS/400 System” on page 41.

You can choose to use the AS/400 Toolbox for Java classes directly from the AS/400 integrated file system by setting the CLASSPATH variable or you can copy the AS/400 Toolbox for Java classes on your PC hard disk drive.

3.4.6.1 Setting Up the CLASSPATH Variable

To directly use the AS/400 Toolbox for Java classes, all you need to do is map a network drive on your PC to the AS/400 integrated file system and direct the Java JVM on your PC to this drive with an appropriate CLASSPATH directive.

The AS/400 Toolbox for Java classes are located on the AS/400 integrated file system in this directory structure:

```
/QIBM/ProdData/HTTP/Public/jt400/lib/
```

To use the AS/400 Toolbox for Java classes, you must tell the JVM where to look for these classes. When working on Windows 95, we do this by using a CLASSPATH directive that we 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. Put the AS/400 Toolbox for Java directory at the end of the SET CLASSPATH= statement.
6. Choose **Save** from the **File** menu to update your autoexec.bat file.
7. Exit from the text editor.

For example, on these SET statements, we set up the AS/400 Toolbox for Java directory structure:

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

First, we set our own environment variable, called AS400JT, with the value of the AS/400 Toolbox for Java main directory on the integrated file system. This assumes that you have mapped a network drive "S:" to your AS/400 system.

Then, we set the CLASSPATH environment variable to instruct the JVM that the AS/400 Toolbox for Java classes are contained in the jt400.zip file that is located in the LIB subdirectory. We also indicate how to find utilities, such as the AS400ToolboxInstaller.class file that is located in the utilities subdirectory. Setting up our own environment variable allows for easier changes later if we decide to download the AS/400 Toolbox for Java classes on the hard disk of the PC.

Figure 54 shows a typical autoexec.bat file that is edited with the Notepad utility.

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

C:\WINDOWS\net start

@ECHO OFF
SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:DOS;%PATH%;C:\JDK1.1.7.B\BIN
```

Figure 54. Setting Up the CLASSPATH Variable for Java

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

You should re-boot your system, but it can run interactively (take 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.

This is the easiest way to use the AS/400 Toolbox for Java classes, and it is 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 AS/400 LPP, changes to the AS/400 Toolbox for Java are distributed by means of PTFs. PTFs are applied to the AS/400 system, so using the classes directly from the AS/400 integrated file system ensures that you automatically get all of the changes after they are applied to the AS/400 system.

3.4.6.2 Copying the AS/400 Toolbox for Java Classes to Your PC

For performance reasons, you may prefer to copy the AS/400 Toolbox for Java classes on your PC hard disk drive. Copying the class files to your workstation allows you to serve the files from your workstation.

To copy the files from AS/400 to your workstation, perform these steps:

1. Decide which method you want to use to copy the files to your workstation. You can either use the AS400ToolboxInstaller class or manually copy either the zip or jar file. Perform these tasks:
   a. The AS/400 Toolbox for Java information fully documents the **AS400ToolboxInstaller** class. In the AS/400 Toolbox for Java information in the AS/400e Information Center, look under "Tips for programming" and then "Install and update."
   b. Find the file named jt400.zip. It should reside in the /QIBM/ProdData/HTTP/Public/jt400/lib directory. Copy jt400.zip from the AS/400 system to your workstation. You can do this in a variety of ways. The easiest way is to use Client Access/400 to map a network drive on your workstation to the AS/400 system. You can also use file transfer protocol (FTP) to send the file to your workstation. If you do this, make sure that you transfer the file in binary mode.
2. Update the CLASSPATH environment variable of your workstation by adding the location where you put the program files. For example, on a personal computer that is using Windows 95 operating system, if jt400.zip resides in C:\jt400\lib\jt400.zip, add ;C:\jt400\lib\jt400.zip to the CLASSPATH.

To update the CLASSPATH, you need to modify the autoexec.bat file on your PC to change the CLASSPATH environment variable, so the JVM on your PC loads the AS/400 Toolbox for Java classes from your PC hard disk, instead of loading them from the AS/400 integrated file system.

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

1. Use a PC text editor, such as Notepad or Wordpad.
2. On the File menu, open the **autoexec.bat** file.
3. Look for the **SET CLASSPATH=AS400JT** line. If there is no line, you need to add one.
4. Change the line to set our private environment variable to `c:\jt400`.

5. Save your changes.
6. Exit from the text editor.

Now, open an MS-DOS session and run the autoexec.bat file again for the change to be effective.

You are now ready to use Java and the AS/400 Toolbox for Java classes on your PC and on the AS/400 system.

3.5 Using Remote AWT Support on Your Workstation

The Remote Abstract Windowing Toolkit (AWT) 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 AS/400 system. The Remote AWT feature of the AS/400 Developer Kit for Java provides an easy way to test and run (on the AS/400 system) any Java application that uses the Java AWT.

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.

3.5.1 Setting Up the Remote AWT Environment

In this section, we show you how to use remote AWT with V4R4 and JDK 1.1.7 support. To setup the Remote AWT environment, make sure that you have downloaded and installed Sun Microsystems, Inc. Java Development Kit (JDK) 1.1. Then, setup access to Remote AWT by accessing the RAWTGui.zip file using either of these methods:

- Map a network drive on your workstation to your AS/400 system and set the proper CLASSPATH environment variable. The JVM that runs on your workstation should find the Remote AWT classes that are contained in the RAWTGui.zip file that is located in the /QIBM/ProdData/Java400/jdk117 directory in the integrated file system.

  Then, update your CLASSPATH statement for Remote AWT by adding the RAWTGui.zip file to your environment. To do this, enter the following statement on a line in your autoexec.bat file:

  ```
  SET RAWT=S:\QIBM\ProdData\Java400\jdk117\RAWTGui.zip
  ```

- If you do not want to run using a network drive, copy the RAWTGui.zip file to your workstation. You can use a network drive or FTP to do this. To do this, create a directory on your workstation (called rawt). Then, copy the Remote AWT RAWTGui.zip file from the AS/400 integrated file system to your workstation. You can do this by using a network drive and entering the DOS commands, shown in Figure 55 on page 74, at the DOS prompt.

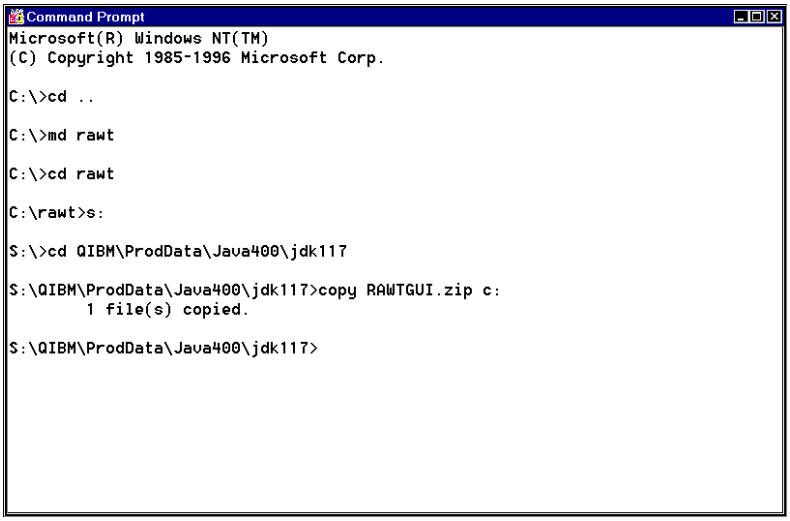


Figure 55. Downloading the Remote AWT Classes to Your Workstation

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

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

Now, you are ready to start Remote AWT.

3.5.2 Starting Remote AWT Support on Your Workstation

Now, you need to start the Remote AWT daemon on your workstation, so it is listening on a TCP/IP port for incoming Remote AWT requests that are sent by the AS/400 Java application.

We do this by starting a Java application named com.ibm.rawt.server.RAWTPCServer on your workstation. To do this, complete these tasks:

1. Open an MS-DOS prompt.
2. Enter `java com.ibm.rawt.server.RAWTPCServer` at the prompt. Press **Enter**.

As shown in Figure 56, the Welcome to Remote-AWT message window appears, which includes the Remote AWT version number and your workstation IP address.

Figure 56. Remote AWT Welcome

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 your workstation.

To create a .bat file, follow these steps:

1. Start an editor, such as WordPad or Notepad.
2. Type in 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 your file in the folder of your choice and give it a meaningful name such as rmtawt.bat.
4. Exit from the editor.

Now, you can start the Remote AWT daemon on your workstation. Enter `rmtawt` at the DOS prompt whenever you need to run a Java application on the AS/400 system that uses the Java AWT APIs. You may even want to include this in your autoexec.bat file to have the daemon ready on your workstation.

You do not have to restart Remote AWT on the remote display after your Java application program ends. The RAWTPCServer 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.

3.5.3 Running Remote AWT Support on the AS/400 System

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

1. Enter the Run Java (`RUNJVA`) 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. Your display should now appear as shown in Figure 57 on page 76.

```
                          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 57. 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: In 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 your application.

3.5.3.1 Multiple JDK Versions

If you have multiple JDK versions installed, follow step 1 through step 13 as described in Section 3.5.3, “Running Remote AWT Support on the AS/400 System” on page 75. Then, add this property to select your 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.

   Note: `x.x.x` is the JDK version (for example, `1.1.6`).

The command line should look like this:

```
java class (classname) prop((RmtAwtServer '1.1.11.11')
(os400.class.path.rawt 1)(java version 1.1.6))
```

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 AS/400 host table and your Domain Name Server and Windows workstation TCP/IP properties.

In each of the preceding examples, a small application named MyApp is used to test the set up. 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();
        }
}
```

Chapter 4. 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

4.1 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, such 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.

4.2 Java

Java brings a number of interesting things to the AS/400 system, 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 AS/400 system because they both share similar architectural principles, as shown in Figure 58.

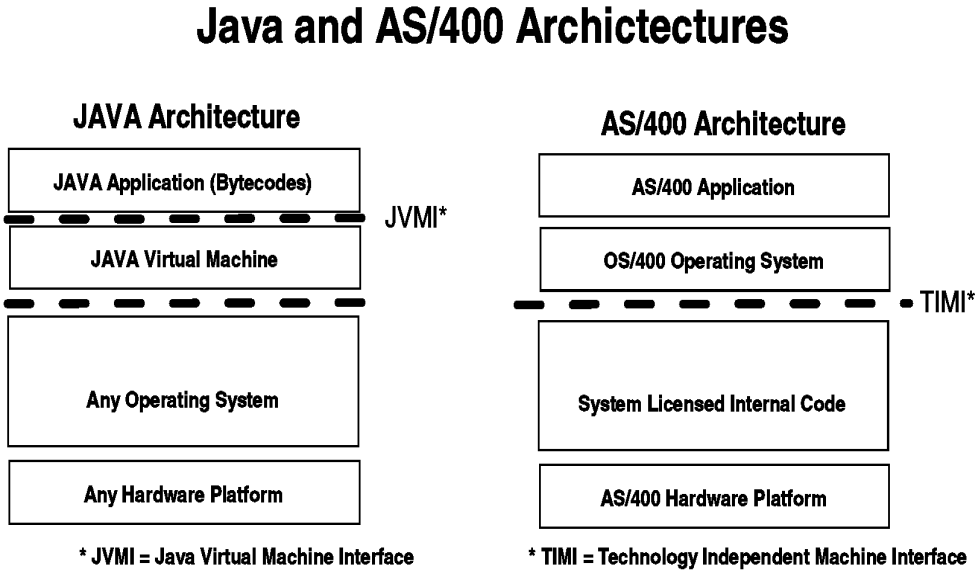


Figure 58. Java Architecture Compared to AS/400 Architecture

Figure 59 on page 82 shows a high-level view of how Java is implemented on AS/400 by taking advantage of the AS/400 architecture.

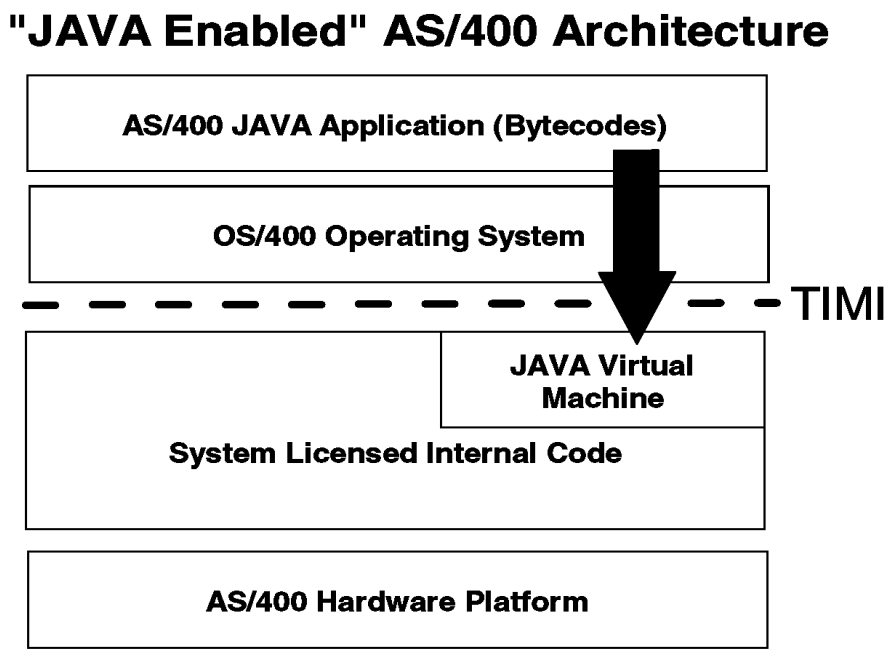


Figure 59. High-Level Schematic of AS/400 Java Implementation

4.2.1 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, 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 system that provided support for the runtime.

More information about the development of Java can be found in *The Java Language: A White Paper* 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**: An 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.

4.2.2 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.

Statement 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.

4.2.3 Object Creation

Every class in Java has a 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.

4.2.4 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. These 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;
```

4.2.5 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);
```

4.2.6 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.

4.2.7 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.

4.2.8 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. 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 and, therefore, can determine when an object is no longer needed.

4.2.9 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.

4.2.10 Overriding Methods

If a class defines a method with the same signature (name and arguments) as a method in its super class, the class is said to override the method. 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.

4.2.11 Compiling Java on the AS/400 System

Java programs on the AS/400 system are stored in the integrated file system. They run as Java bytecodes or direct execution programs.

Running a Java program from the bytecodes results in exactly the same form of execution as all other JVMs. However, the AS/400 system supports a direct execution mechanism where the Java bytecodes for a class are transformed into a service program that results in faster processing.

The default method of running a Java program causes a direct execution version of the program to be created if one does not already exist or if the service program and the class file are not synchronized. To run from the Java bytecodes, the program must be run in interpreted mode.

You can compile Java on the AS/400 system by using any of these commands:

- Use the `javac` command within the Qshell Interpreter on the AS/400 system 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 AS/400 command line to create a direct execution version of the .class file.
- Run the `java` command or Run Java (`RUNJVA`) 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, IBM Press (1998).

Chapter 5. Overview of the Order Entry Application

In this chapter, we cover 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 section introduces the application and specifies the database layout. In Chapter 6, “Migrating the User Interface to Java Client” on page 101, 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 7, “Moving the Server Application to Java” on page 133, we convert the server-based RPG application to Java, so both sides of the application are written in Java.

5.1 Overview of the Order Entry Application

This section provides an overview of the application and a description of how the application database is used.

5.1.1 The ABC Company

The ABC Company is a wholesale supplier with one warehouse and 10 sales districts. Each district serves 3000 customers (30000 total customers for the company). The warehouse maintains stock for the 100000 items sold by the Company.

Figure 60 illustrates the company structure (warehouse, district, and customer).

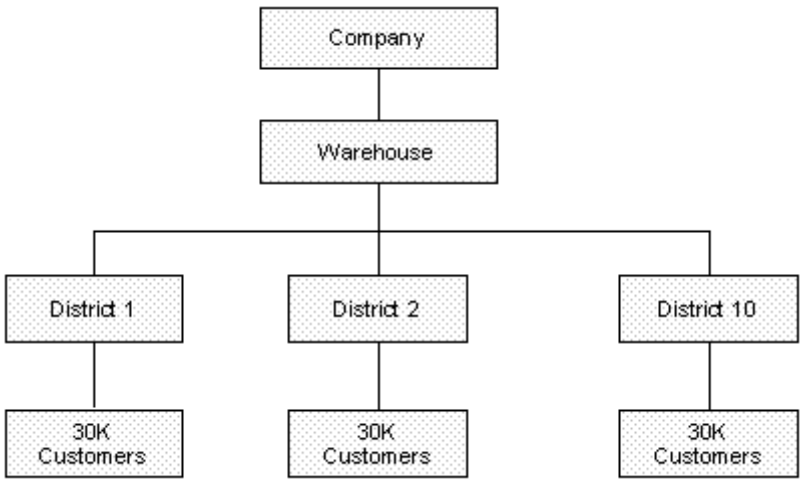


Figure 60. The Company Structure

5.1.2 The ABC Company Database

The company runs its business with a database. This database is used in a mission critical, OLTP (online transaction processing) 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)

5.1.3 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 (CSTMR).
   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.

5.1.4 Application Flow

The RPG Order Entry Application consists of the following components:

Note

To download the sample code used in this redbook, please refer to Section A.1, “Downloading the Files from the Internet” on page 415, for more information.

- **ORDENTD (Parts Order Entry)**—Display File
- **ORDENTR (Parts Order Entry)**—Main RPG processing program
- **PRTORDERP (Parts Order Entry)**—Print File
- **PRTORDERR (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

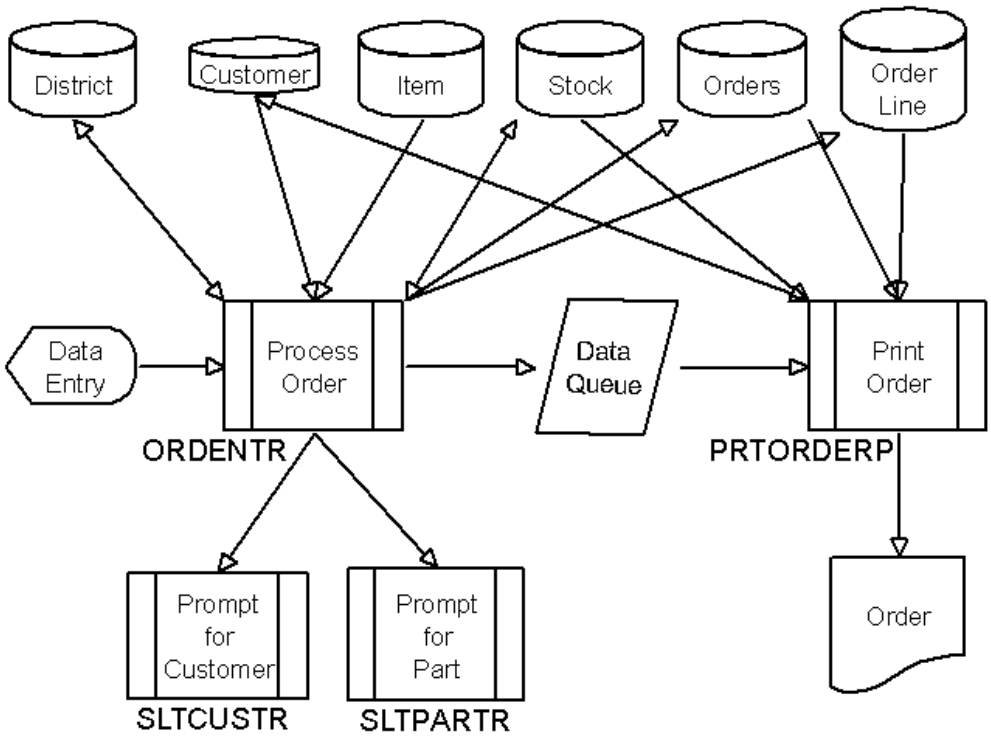


Figure 61. RPG Application Flow

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.

5.1.5 Customer Transaction Flow

The following scenario walks through a customer transaction showing the application flow. By understanding the flow of the AS/400 application, you can understand the changes made to this application to support a graphical client.

5.1.5.1 Starting the Application

To start the application, the customer calls the main program from an AS/400 command line:

```
CALL ORDENTR
```

When the Order Entry application is started, the display shown in Figure 62 appears.

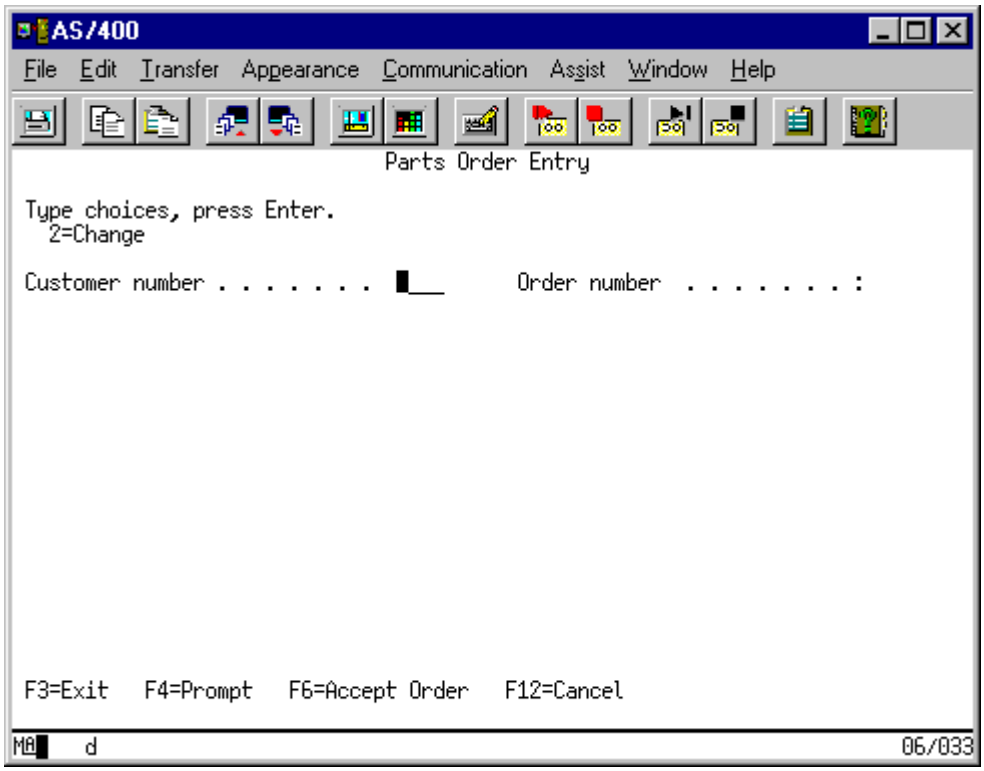


Figure 62. Parts Order Entry

When the Parts Order Entry display appears, the user has two options:

- Type in 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.

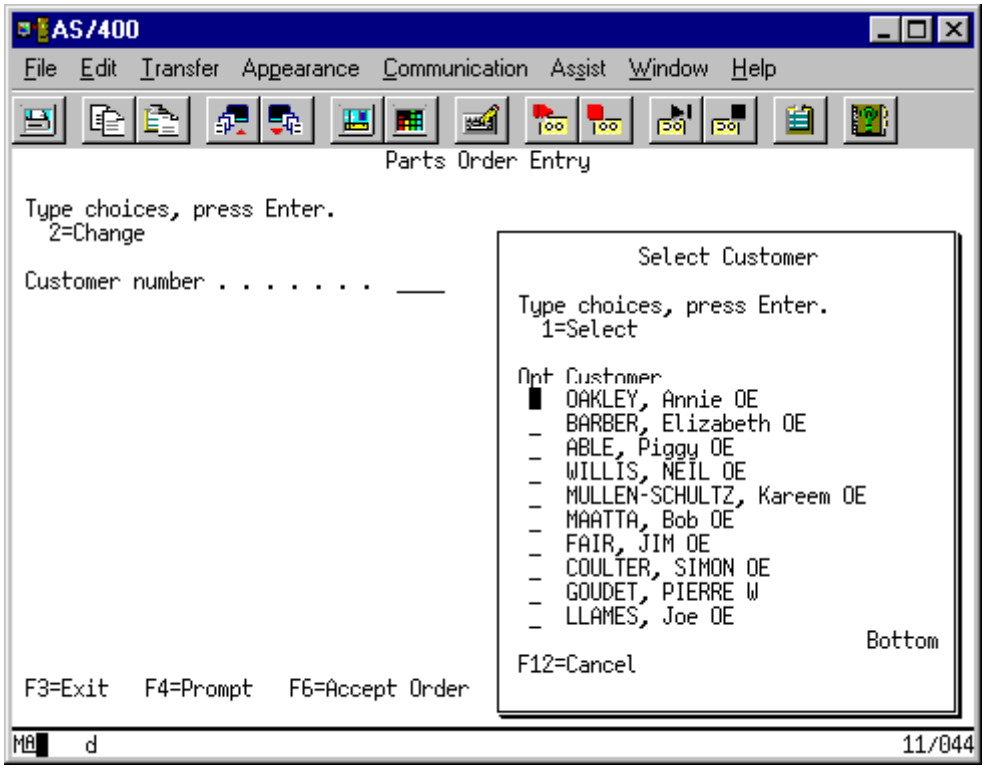


Figure 63. Select Customer

The user presses F12 to remove the window and return to the initial panel, or scrolls through the items in the list until they find the customer they want. By typing a 1 in the option field and pressing the Enter key, the user indicates their choice. The selected customer is then returned to the initial panel (Figure 64 on page 92).

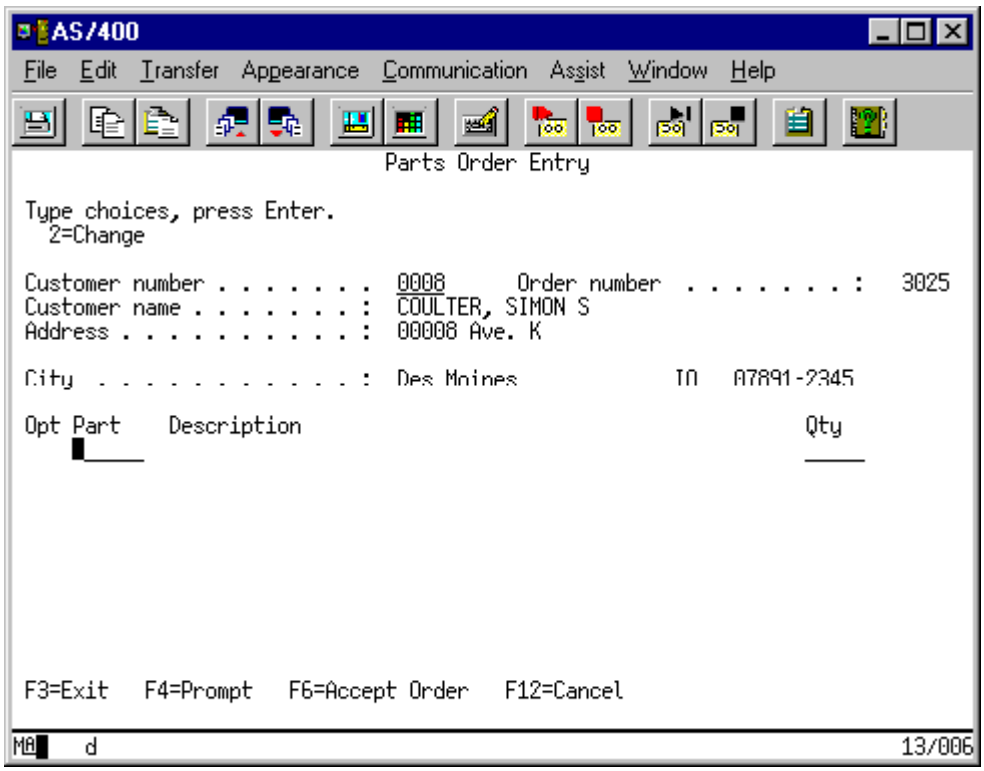


Figure 64. Parts Order Entry

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 allowing 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 65 on page 93).

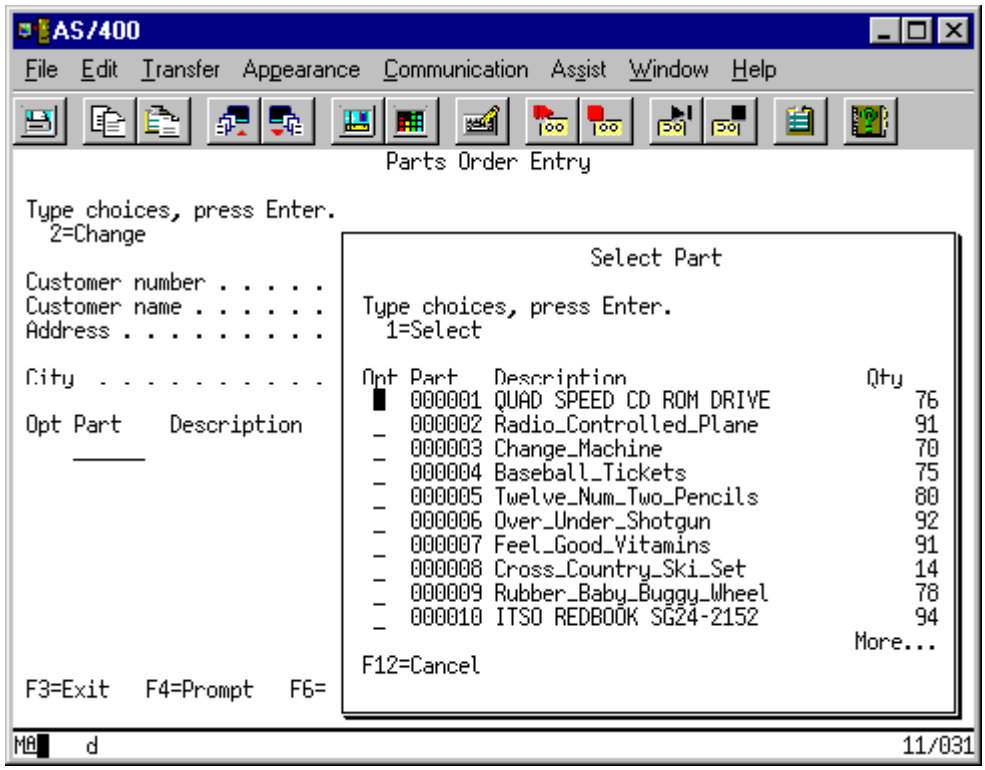


Figure 65. Select Part

The user presses F12 to remove the window and return to the initial panel, or scrolls through the items in the list until they find the part they want. By typing a 1 in the option field and pressing the Enter key, they indicate their choice. The selected part is returned to the initial panel (Figure 66).

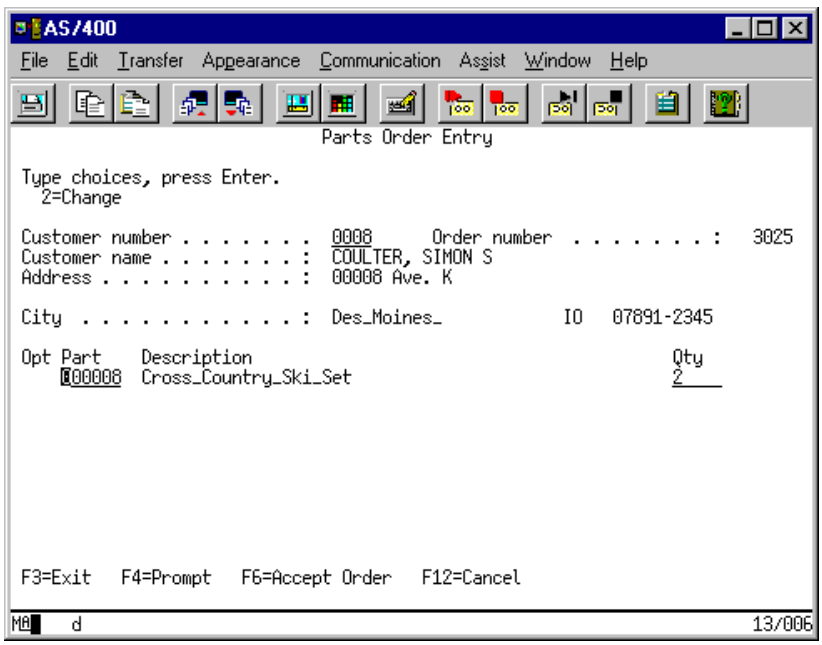


Figure 66. Parts Order Entry

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 67).

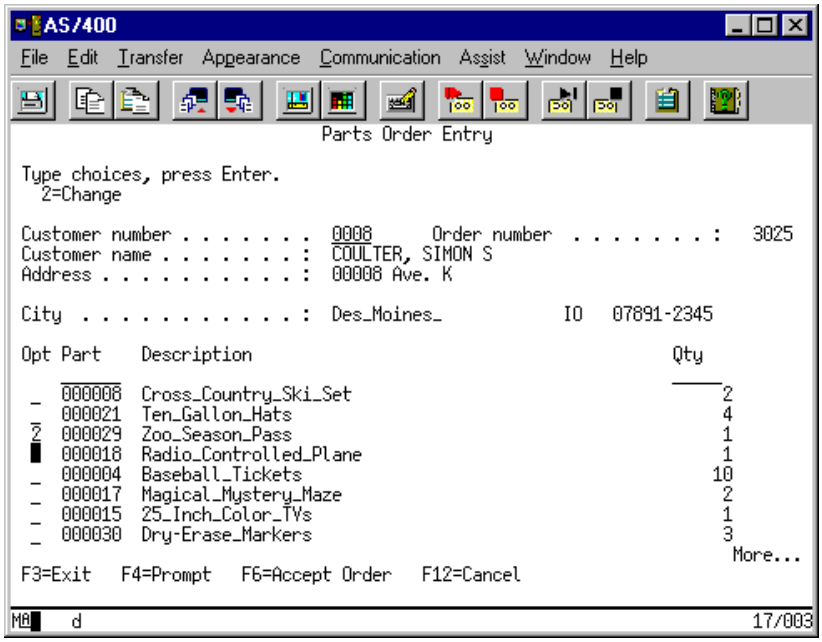


Figure 67. Parts Order Entry

The user may type a 2 beside an entry in the list to change the order. When the Enter key is pressed, a window appears that allows the order line to be changed (Figure 68).

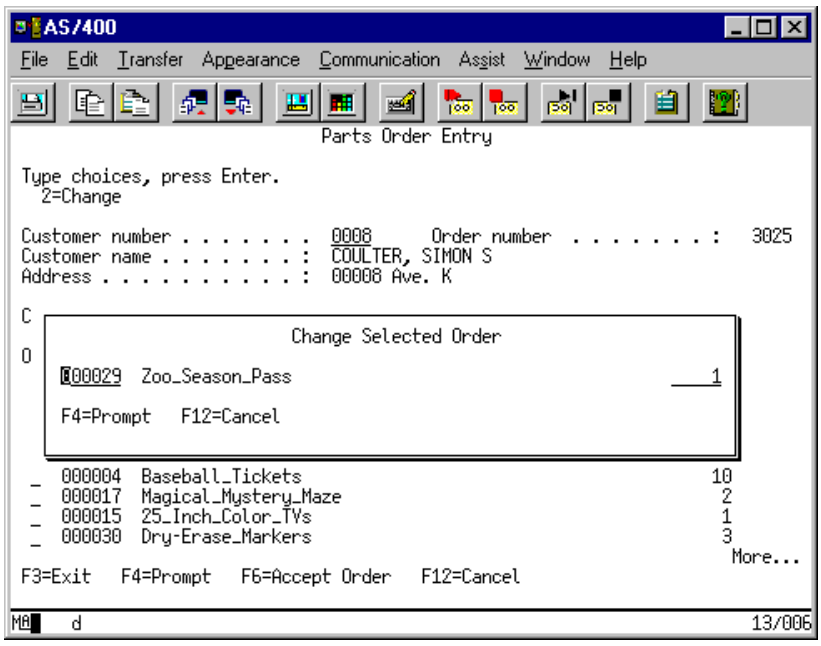


Figure 68. Change Selected Order

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.

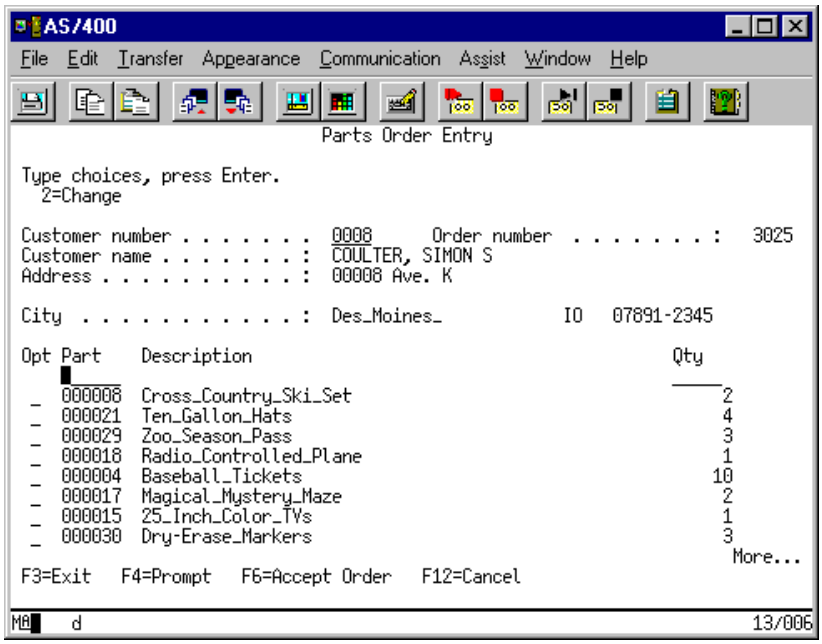


Figure 69. Completed Order

In Figure 69, you see the quantity for Zoo Season Pass is changed to 3. When the order is complete, the user presses F6 to update the database. Then, an order is placed on the data queue for printing.

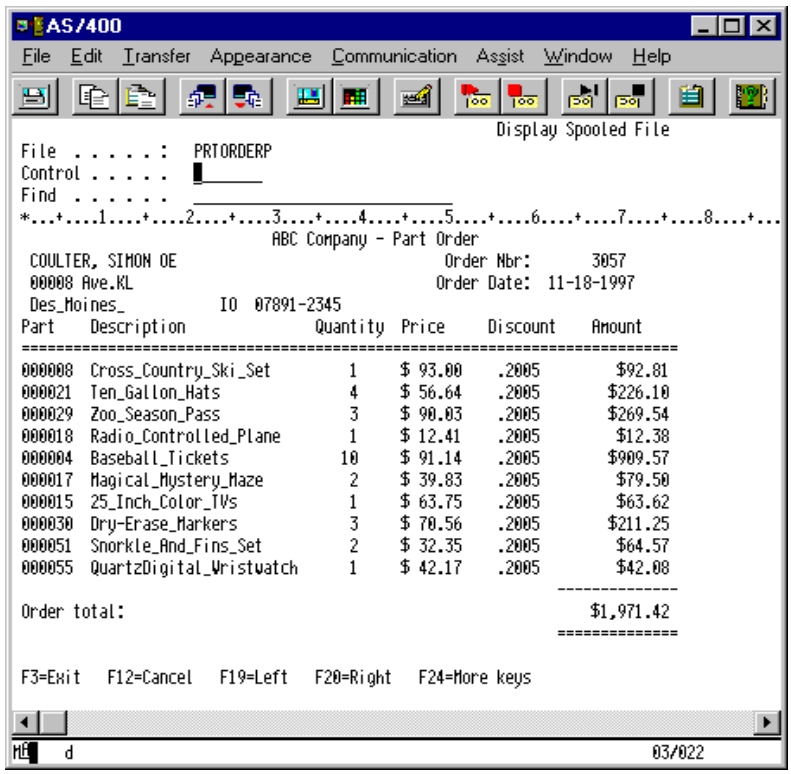


Figure 70. Printed Order

The printed order (Figure 70 on page 95) is created by a batch process. It shows the customer details and the items, quantities, and cost of the order.

5.1.6 Database Table Structure

The ABC Company database has eight tables:

- District
- Customer
- Order
- Order Line
- Item
- Stock
- Warehouse
- History

The relationship among these tables are shown in Figure 71.

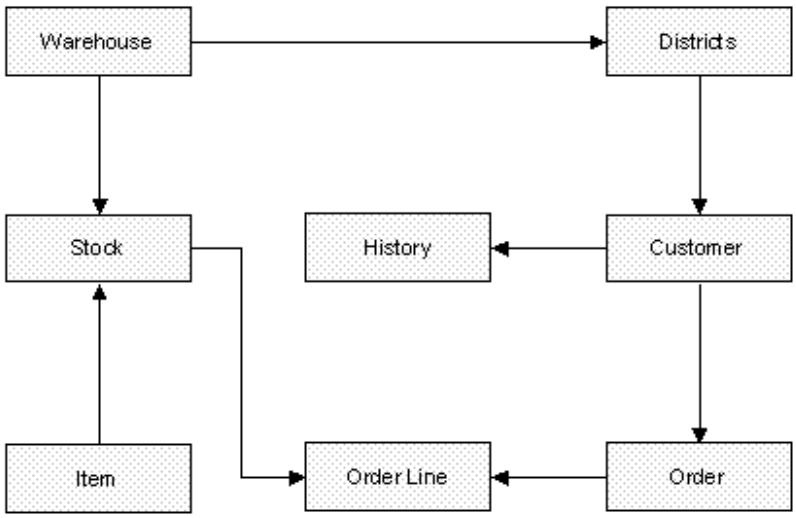


Figure 71. Table Relationships

5.1.7 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 sections describe, in detail, the layout of the database.

5.1.7.1 Tables

Table 2. 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 3. 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
CCREDT	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
CCRDLM	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 4. 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 5. 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. 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 7. 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
STDI01	District Information	Character	24
STDI02	District Information	Character	24
STDI03	District Information	Character	24
STDI04	District Information	Character	24
STDI05	District Information	Character	24
STDI06	District Information	Character	24
STDI07	District Information	Character	24
STDI08	District Information	Character	24
STDI09	District Information	Character	24
STDI10	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			

5.1.8 Database Terminology

This redbook concentrates on the use of the AS/400 system as a database server in a client/server environment. In some cases, we use SQL to access the AS/400 database. In other cases, we use native database access.

The terminology used for the database access is different in both cases. In Table 8, you find the correspondence between the different terms.

Table 8. Database Terminology

AS/400 Native	**SQL**
Library	Collection
Physical File	Table
Field	Column
Record	Row
Logical File	View or Index

Chapter 6. Migrating the User Interface to 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 5, “Overview of the Order Entry Application” on page 87. 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. Figure 72 shows the original design of the Order Entry application.

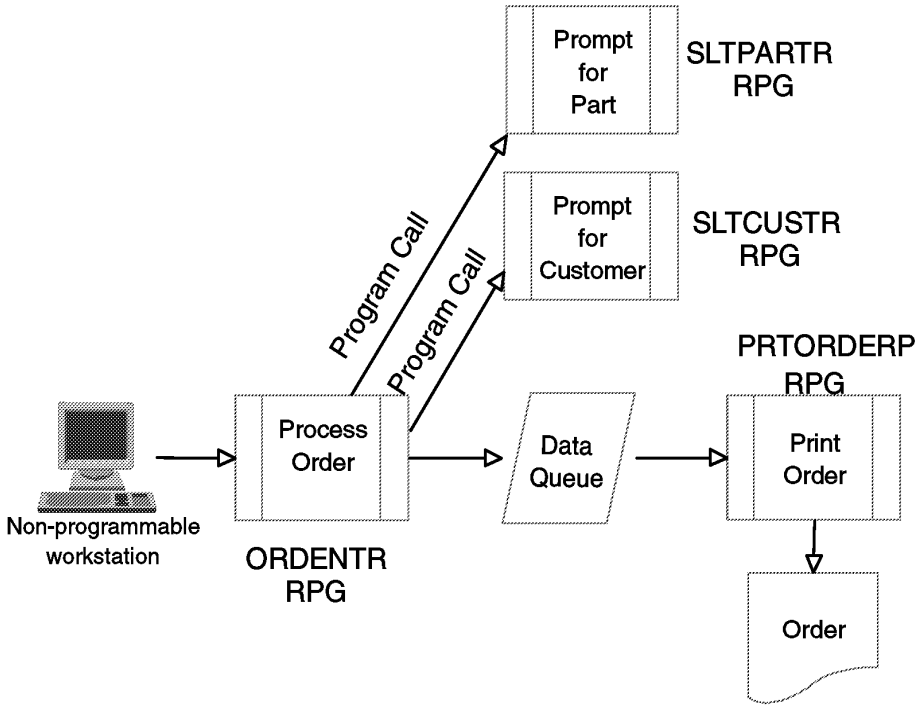


Figure 72. Original Order Entry Application

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 73 on page 102 shows the new design.

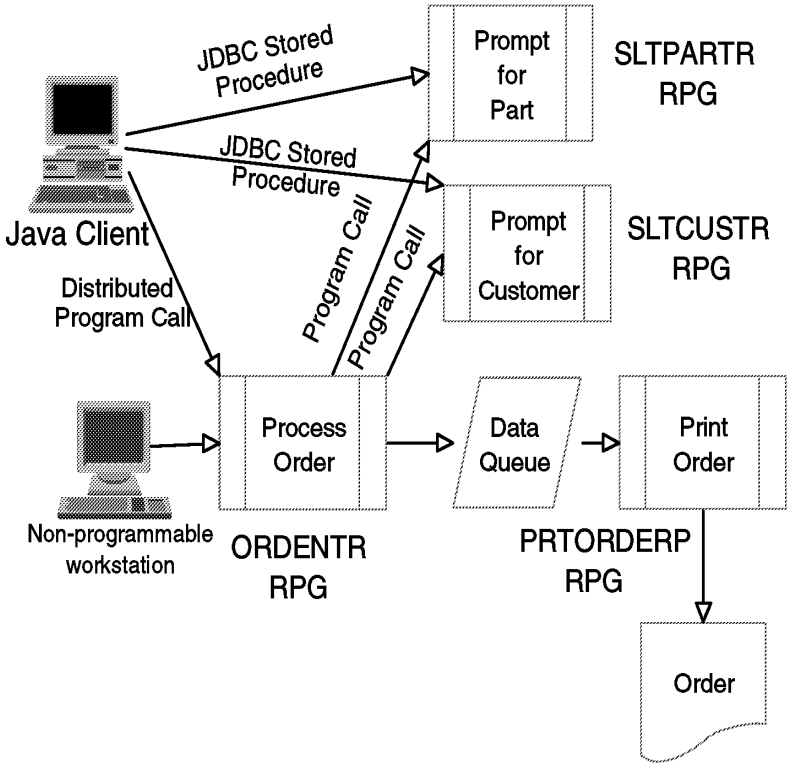


Figure 73. Java Client Order Entry Application

6.1 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. The AS/400 Toolbox for Java accesses data and programs on the AS/400 system. These AS/400 Toolbox for Java topics are covered:

- Stored procedures using the AS/400 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 AS/400 Toolbox for Java. For more information about these topics, see the redbook *Building AS/400 Client/Server Applications with Java*, SG24-2152.

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.

6.2 Overview of the Parts Order Entry Window

Figure 74 shows the main Order Entry window. This window is built using VisualAge for Java. The name of the class that defines this window is OrderEntryWdw2. It is the controlling class, or entry point, of the client application.

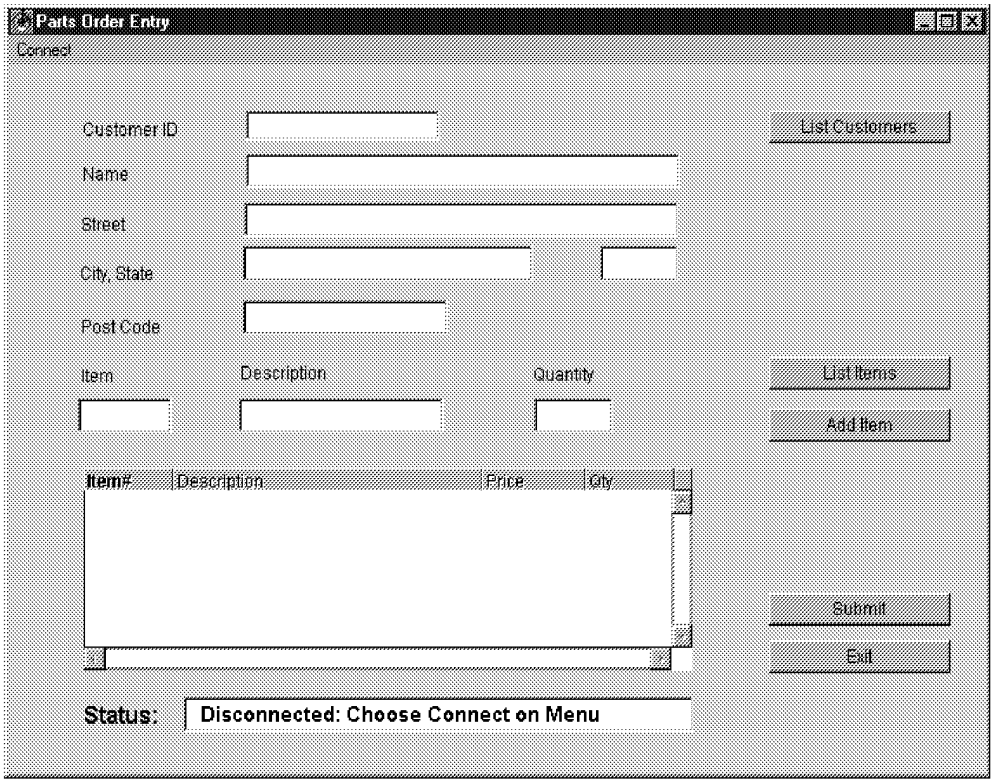


Figure 74. Parts Order Entry — Initial Panel

This list contains these primary window components:

- 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 multi-column list box 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 AS/400 system).

This Java code is a partial listing of the class definition for OrderEntryWdw2:

Program Listings

For complete listings of all the code examples shown throughout this book, refer to Appendix A, “Example Programs” on page 415, for instructions to access the redbook Web site and download the example code.

```
import java.math.*;
import java.sql.*;
import com.ibm.as400.access.*;
import OrderEntry.*;
public class OrderEntryWdw2 extends java.awt.Frame
             implements java.awt.event.ActionListener,
             java.awt.event.KeyListener,
             java.awt.event.WindowListener
{
    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. Additionally, 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 AS/400 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.

6.3 Application Flow through the Java Client Order Entry Window

The series of tasks that the client application supports is summarized in this list:

- 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.

Now, we examine each of these tasks.

6.3.1 Connecting to the Database

Once the Parts Order Entry window has initially displayed, choose the **Connect** menu option to connect to the AS/400 system, as shown in Figure 75.

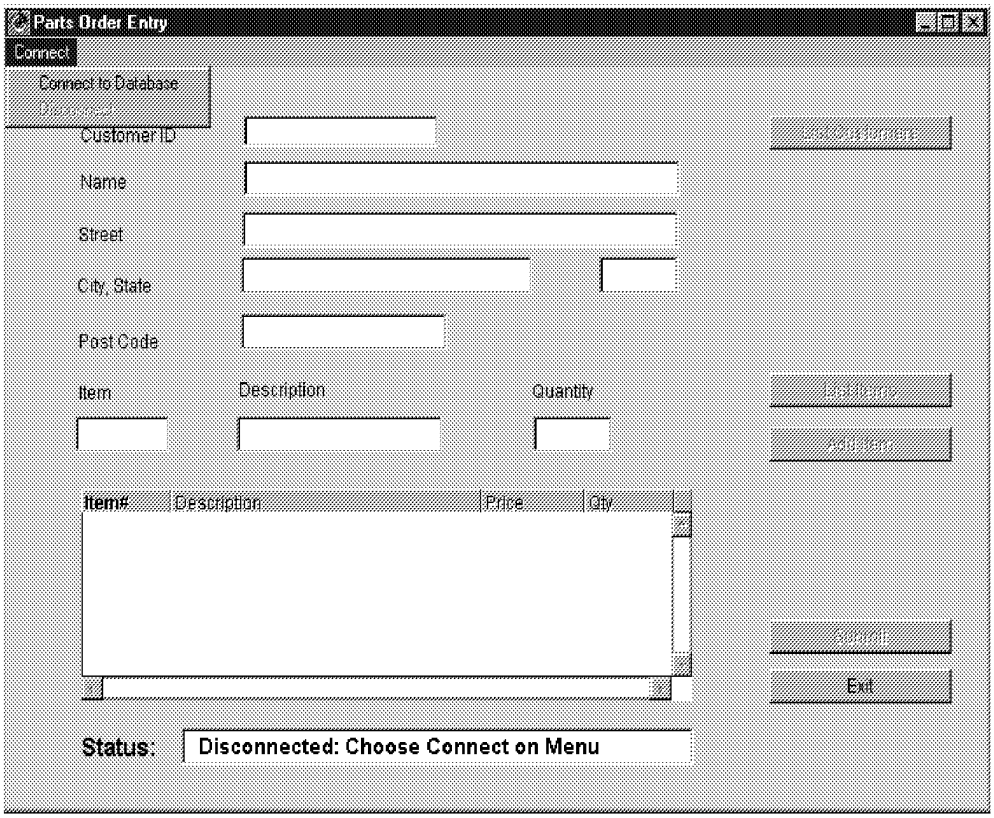


Figure 75. Parts Order Entry — Database Connect

A dialog is shown that requests sign-on information. You must enter a machine name, user ID, and password, as shown in Figure 76.

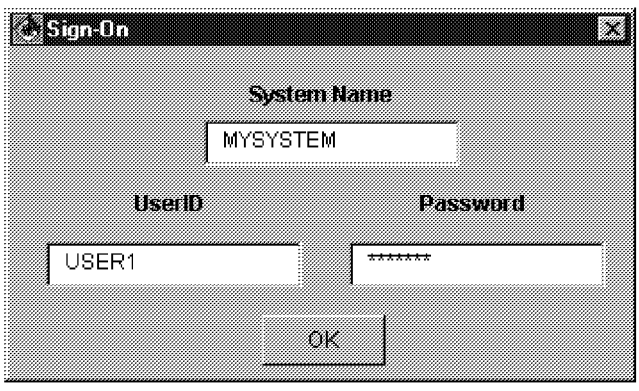


Figure 76. Sign On to System

When you press the **OK** button, it creates a connection to the connectToDb() method of the OrderEntryWdw2 class.

Here is the code for this method:

```
public void 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",
              userid,password);
  }
  catch(Exception e)
  {
       updateStatus("Connect failed");
       handleException(e);
       return;
  }

  updateStatus("Connected to " +  systemName);

  return;

 } // end method
```

This method has two main responsibilities. It starts the openItemFile() method and establishes a JDBC connection. The openItemFile() method opens an AS/400 remote file using record-level access functionality that is provided by the AS/400 Toolbox for Java. This is discussed in greater detail later.

After starting the openItemFile() method, the AS/400 JDBC driver is loaded with this statement:

```
DriverManager.registerDriver
     (new com.ibm.as400.access.AS400JDBCDriver());
```

Next, you establish the SQL connection by starting 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",
              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 AS/400 system. This provides better performance than using dynamic SQL. The name of the package we store the statements in is JavaMig. 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 AS/400 system. This is the code for the method:

```
public void openItemFile()
{
    // initialize the as400 data member
    as400 = new AS400(systemName, userid, password);

    // declare a path name for the file
    QSYSObjectPathName fileName = new QSYSObjectPathName
    ("APILIB","ITEM","*FILE","MBR");

    // establish a handle to the ITEM file
   itemFile = new KeyedFile(as400, fileName.getPath());

   try
   {
      as400.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(as400,
                 "/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 the open options
   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;

} // end method
```

The openItemFile() method initializes the as400 data member and establishes the RECORDACCESS connection for this object. The itemFile object is initialized, so it becomes a handle to the ITEM file on the AS/400 system. 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 AS/400 system. In summary, an SQL connection is established and a handle to the ITEM file on the AS/400 system 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 AS/400 Customer Master database.

6.3.2 Program Interfaces

The Java client program uses a number of different interfaces, as shown in Figure 77 on page 109, to access the AS/400 system. All of these interfaces are provided by the AS/400 Toolbox for Java. JDBC stored procedures populate the customer listbox and the item listbox. 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.

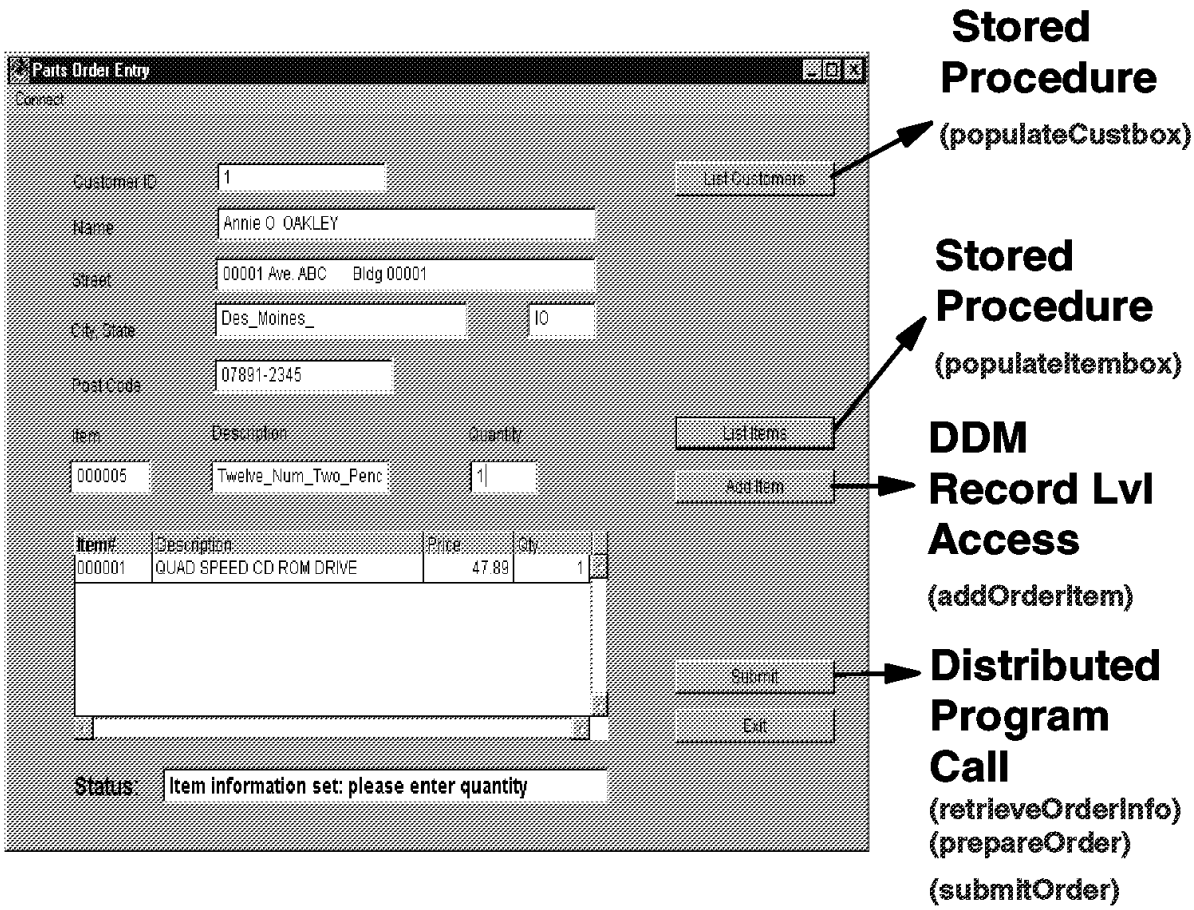


Figure 77. Java Client Programming Interfaces

6.3.3 Retrieving the Customer List

In this section, we look at the code that allows us to retrieve a list of valid customers from the AS/400 system. Figure 78 on page 110 shows the window that is displayed once the list of customers has been retrieved.

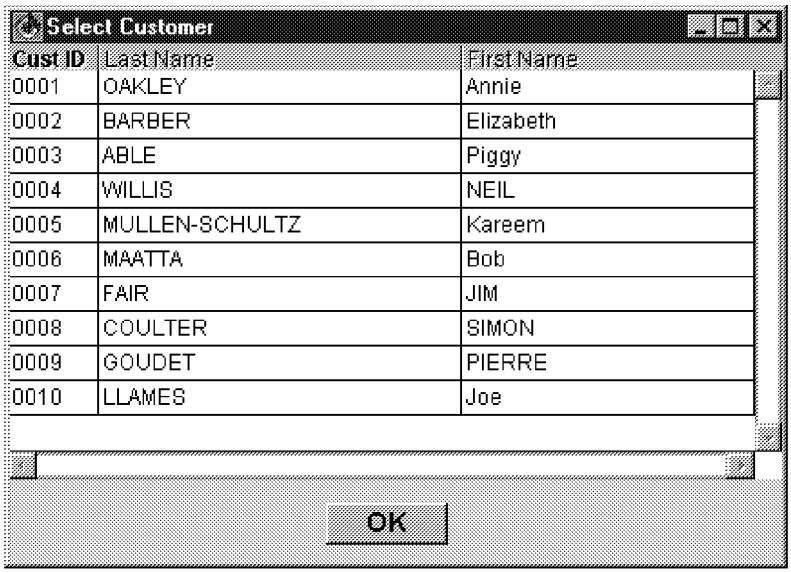


Figure 78. Select a Customer

SltCustWdw is the class that defines this window. It contains a multi-column list box and one button. JDBC accesses an SQL stored procedure on the AS/400 system. 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.*;

public class SltCustWdw extends java.awt.Frame
             implements java.awt.event.ActionListener
{
  private Connection dbConnect = null;
  private com.ibm.ivj.eab.dab.IMulticolumnListbox
             ivjIMulticolumnListbox1 = null;
  private java.awt.Button ivjSltCustBTN = null;
  private OrderEntryWdw2 orderWindow = null;
}
```

When you click on the **List Customers** button on the Order Entry window, the constructor for SltCustWdw starts. The constructor receives a reference to the Order Entry window and SQL connection object. Here is the code for the constructor:

```
public SltCustWdw ( OrderEntryWdw2 orderWdw,
                    Connection dbConnect)
{
  super();
  initialize();
  orderWindow = orderWdw;
  this.dbConnect = dbConnect;
  setupCustBox();
  populateCustBox();
  this.show();
}
```

The setupCustBox() method simply adds the columns to the list box and sets some border characteristics. Data retrieval is done 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
  String[] array = new String[9];

  ResultSet rs = null;
  CallableStatement callableStatement = null;

  try
  {
     // invoke the stored procedure on the AS/400
     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");
         getIMulticolumnListbox1().addRow(array, array[0]);
     }
   }
   catch(SQLException e)
  {
          orderWindow.updateStatus
                  ("Error retrieving customer list");
          handleException(e);
      return;
  }

  orderWindow.updateStatus("Customer list retrieved");

  return;
} // end method
```

As previously shown, populateCustbox() receives a result set through an invocation of a remote stored procedure. This is done by starting 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 AS/400 system 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 list box is filled, the user selects a valid customer with a mouse click and then clicks on the **OK** button. This action is connected to the invocation of the custSelected() method. Here is the code for this method:

```
private void custSelected()
{

  Object[] selectedRow = getCustMLB().getSelectedRow();

  // declare an array of strings the same size as the
  // number of columns in the list box
  String[] custInf = new String[selectedRow.length];

  // retrieve the data from the selected row. It is
  // returned as an array of Object and each element
  // will be converted to a String object.
  for(int i=0;i<selectedRow.length;i++)
  {
       custInf[i] = selectedRow[i].toString();
   }

  // instantiate a Customer object, passing the
  // constructor the array of String data
  Customer custSelected = new Customer(custInf);

  // invoke the method that will set the text fields
  // in the OrderEntryWdw2
  orderWindow.setSelectedCust(custSelected);

  // 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 list box.

Now, we examine the Customer class.

```
public class Customer
{
  public 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 upon 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 OrderEntryWdw2 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);
  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 79 on page 114 shows the current state of the main window.

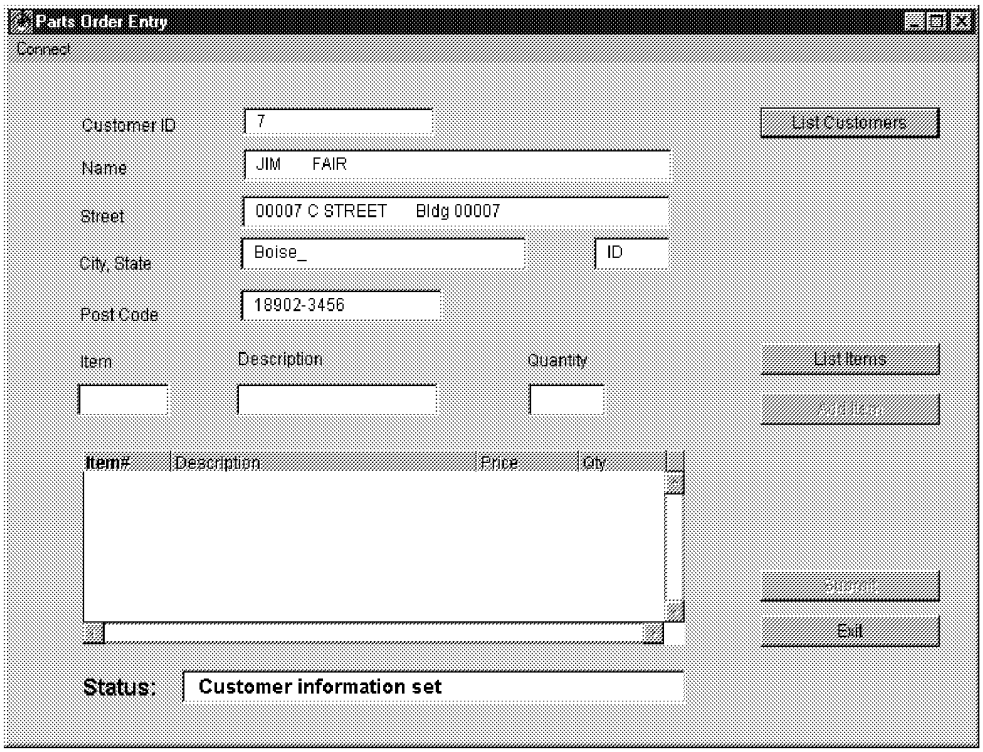


Figure 79. Order Entry Window with Customer Data

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 AS/400 system.

6.3.4 Retrieving the Item List

In this section, we examine the code that is needed to retrieve a list of valid items from the AS/400 system. The process is similar to retrieving the customer list. Figure 80 on page 115 shows the window that is displayed once the list of items has been retrieved.

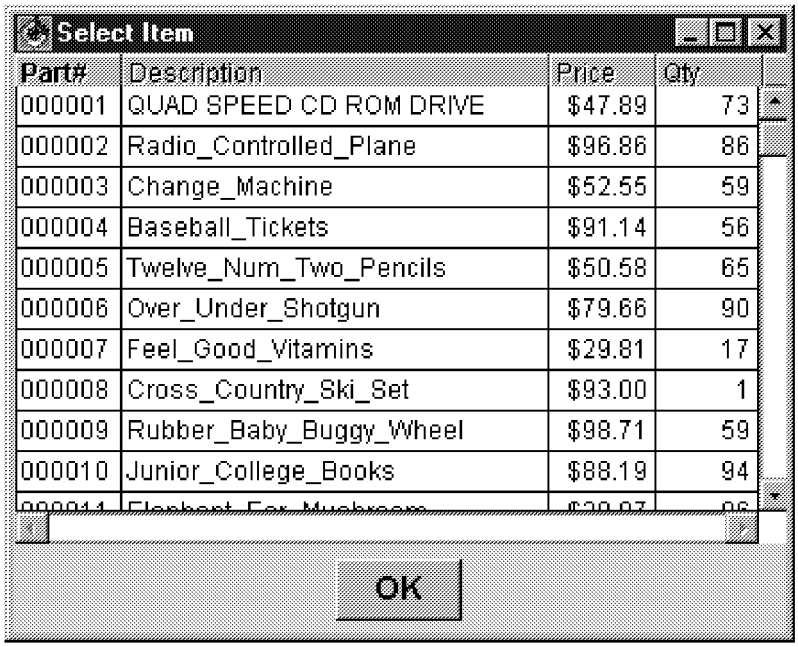


Figure 80. Selected Items

SltItemWdw is the class that defines this window. It contains a multi-column list box and one button. JDBC accesses an SQL stored procedure on the AS/400 system. 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 SltItemWdw:

```
import java.sql.*;

public class SltItemWdw extends java.awt.Frame
             implements java.awt.event.ActionListener,
             java.awt.event.WindowListener
{
  private Connection dbConnect = null;
  private java.awt.Button ivjButton1 = null;
  private com.ibm.ivj.eab.dab.IMulticolumnListbox
                ivjIMulticolumnListbox = null;
  private OrderEntryWdw2 orderWindow = null;
}
```

When you click on the **List Items** button on the Order Entry window, the constructor for SltItemWdw starts. 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 (OrderEntryWdw2 orderWdw,
                   Connection dbConnect)
{
  super();
  initialize();
  orderWindow = orderWdw;
  this.dbConnect = dbConnect;
```

```
  setupItemBox();
  populateItemBox();
  this.show();
}
```

The setupItemBox() method simply adds the columns to the list box and sets some border characteristics. Data retrieval is done 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
  String[] array = new String[4];

  ResultSet rs;
  CallableStatement callableStatement;

  try
  {
   // invoke the stored procedure on the AS/400 system
    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", 2).toString();
        array[3] = Integer.toString(rs.getInt("STQTY"));
       getIMulticolumnListbox1().addRow(array, array[0]);
    }   // end while

  }  // end try
  catch(SQLException e)
  {
    orderWindow.updateStatus("Error retrieving item list");
    handleException(e);
    return;
  }
  orderWindow.updateStatus("Item list retrieved");
  return;
}
```

The process of populating the ITEM list box is almost identical to the process of populating the CUSTOMER list box. However, two of the fields that are retrieved are not characters. The IPRICE field (column) is stored as packed decimal data on the AS/400 system. 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, start the static toString() method of the Integer class:

```
array[3] = Integer.toString(rs.getInt("STQTY"));
```

After the item list box is filled, the user selects a valid item with a mouse click and clicks on the OK button. This action is connected to the invocation of the itemSelected() method. Here is the code for this method:

```
private void itemSelected()
{
  Object[] selectedRow = getItemMLB().getSelectedRow();

  // declare an array of String the same size as the row
  String[] itemInf = new String[selectedRow.length];

  // retrieve each item in the selected row, convert to
  // String and put it into the String array
  for(int i=0;i<selectedRow.length;i++)
  {
       itemInf[i] = selectedRow[i].toString();
   }

   // instantiate an Item object and pass it to the
   // setSelectedItem() method of the order window
  Item itemSelected = new Item(itemInf);
  orderWindow.setSelectedItem(itemSelected);

  // 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. Next, examine the ITEM class.

```
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 OrderEntryWdw2 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();

  updateStatus("Item information set: please enter quantity");

  return;
}
```

6.3.5 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 id text field is passed in as a parameter. This method uses the RECORDACCESS functionality that the AS/400 Toolbox for Java provides to retrieve a record from the ITEM file. If the record is found, it adds the information to the Order list box 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();
        getIMulticolumnListbox1().addRow(orderRow, orderRow[0]);
        getIMulticolumnListbox1().repaint();
        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;
 }

  return;
 }
```

Once an item is added to the order list, the Submit button is enabled. More items may be added to the list, or the order may be submitted. Figure 81 on page 120 shows the state of the window at this point.

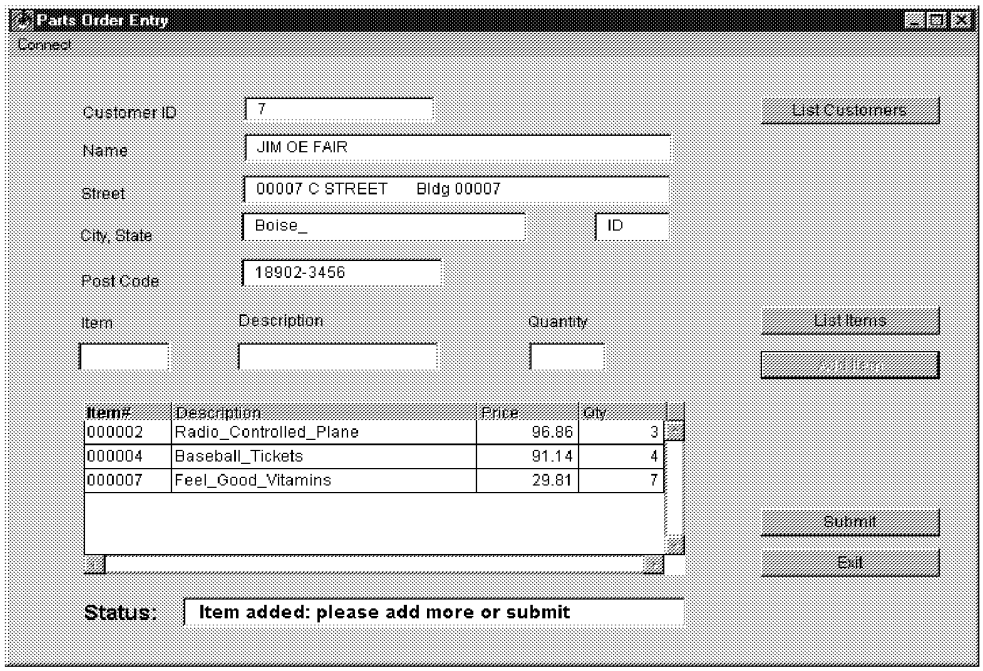


Figure 81. Parts Order Ready to Submit

6.3.6 Submitting the Order

When you click on the **Submit** button, the retrieveOrderInf() 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 by retrieving each row in the Order list box.

```
public void retrieveOrderInfo()
{

 int numEntries = getOrderMLB().countRows( );

 Order theOrder = new Order(getCustIDTF().getText());
 for(int i=0;i<numEntries;i++)
 {
    Object[] detailRow = getOrderMLB().getRow(i);
    OrderDetail detail = new OrderDetail
                      (detailRow[0].toString(),
                     detailRow[1].toString(),
                     new BigDecimal(detailRow[2].toString()),
                     new BigDecimal(detailRow[3].toString()));
    theOrder.addEntry(detail);
 }

 prepareOrder(theOrder);

 return;
}
```

The Order class is now examined:

```
public class Order
{
   private StringBuffer customerId = new StringBuffer(4);
   private OrderDetail[] entryArray = new OrderDetail[50];
   private int index = -1;
   private int cursor = 0;
}
```

As shown, an Order object contains an array of OrderDetail objects. The size of this array is arbitrarily set to 50. You can also use a Vector, which avoids having to determine a size in advance. Vectors allow dynamic allocation, where arrays do not.

The customerId field is declared as a StringBuffer rather than a String. This is because leading zeros may have to be inserted. The AS/400 system 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. This is shown in the constructor:

```
public Order ( String cid)
{
   // Set the customer id making sure leading zeros are
   // included.
   for(int i=0;i<4-cid.length( );i++)
   {
       customerId.append('0')
   }
   customerId.append(cid);
}
```

Since the AS/400 system 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:

1. Starting at offset 0 for a length of 4 bytes: The customer ID
2. Starting at offset 4 for a length of 5 bytes: The number of detail entries
3. 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:

Program Listings

The complete listing of the class is available in the example code. See Appendix A, “Example Programs” on page 415, for more information. You can also view other methods, such as getFirstEntry() and getNextEntry() there. The getNextEntry( ) may also be viewed there.

```
public String toString()
{
  // declare a StringBuffer
  StringBuffer orderBuffer =
              new SringBuffer(9+(40*getNumEntries()));
  // 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
  StringBuffer numEntryBuffer = new StringBuffer(5);
  String numEntryString = Integer.toString(getNumEntries( ));

  for(int i=0;i<5-numEntryString.length( );i++)
  {
      numEntryBuffer.append('0');
  }
  numEntryBuffer.append(numEntryString);

  // append the number of entries string to the order buffer
  orderBuffer.append(numEntryBuffer);

  // now append a string representation of all entries to
  // the buffer
  OrderDetail entry = getFirstEntry();

  while(entry != null)
  {
     // 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(entry.toString( ));
     entry = getNextEntry( );
  }

  // return the complete StringBuffer as a String
  // this is done by invoking the toString( ) method that is
  // provided by Java's StringBuffer class
  return orderBuffer.toString( );
}
```

The OrderDetail class is now shown:

```
public class OrderDetail
{
  StringBuffer itemId = new StringBuffer(6);
  StringBuffer itemDsc = new StringBuffer(24);
  BigDecimal itemPrice = null;
  BigDecimal itemQty = null;
}
```

As in the Order class, take care so that certain data members have a specific length. In this case, itemId and itemDsc are declared as StringBuffer types. In some cases, the constructor must add leading zeros to the itemId. 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
  for(int i=0;i<6-itemId.length();i++)
  {
      this.itemId.append('0');
   }
  this.itemId.append(itemId);

  // set the description making sure trailing blanks are there
  this.itemDsc.append(itemDsc);
  for(int j=itemDsc.length()+1;j<25;j++)
  {
          this.itemDsc.append(' ');
  }

  this.itemPrice = itemPrice;
  this.itemQty = itemQty;
}
```

The OrderDetail class also provides a toString() method. Once again, this facilitates the call to the AS/400 system 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:

1. Starting at offset 0 for 6 bytes: The item ID
2. Starting at offset 6 for 24 bytes: The item name (description)
3. Starting at offset 30 for 5 bytes: The price
4. 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
```

```
    StringBuffer priceBuffer = new StringBuffer(5);
    String priceString = itemPrice.toString();
    // find out the position of the decimal point
    int decimalPosition = priceString.indexOf('.');
    // create a new string that contains the digits
    // before the decimal point
    String priceString1=
           priceString.substring(0,decimalPosition);
    // create a new string that contains the digits
    // after the decimal point
     String priceString2 = priceString.substring(decimalPosition+1)
    // now combine the 2
     priceString = priceString1 + priceString2;

    // insert any needed leading zeros
   for(int i=0;i<5-priceString.length();i++)
   {
       priceBuffer.append('0');
    }
    priceBuffer.append(priceString);
    // now append it to the entry buffer
    entryBuffer.append(priceBuffer);

    // convert the quantity field to String and make sure
    // it is 5 bytes
    StringBuffer qtyBuffer = new StringBuffer(5);
    String qtyString = itemQty.toString();
    for(int i=0;i<5-qtyString.length();i++)
    {
      qtyBuffer.append('0');
    }
    qtyBuffer.append(qtyString);
    // now append it to the entry buffer
    entryBuffer.append(qtyBuffer);

   // now return the whole entry as a String
   return entryBuffer.toString();
}
```

It was previously mentioned that the action of the Submit button was connected to an invocation of the retrieveOrderInf() method. That method creates an Order and adds OrderDetail objects to it. Next, the retrieveOrderInf() method starts the prepareOrder() method and passes the Order object as a parameter. The prepareOrder() method builds the constructs that are used as parameters when starting the AS/400 system RPG program that processes orders (ORDENTR). 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 Order.toString().

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 Order.toString(). It begins at offset 9 of the string.

This is the prepareOrder() method:

```
public void prepareOrder(Order theOrder)
{

  String orderString = theOrder.toString();
  StringBuffer headerBuffer = new StringBuffer(9);
  headerBuffer.append(orderString.substring(0,9));

  StringBuffer detailBuffer =
                   new StringBuffer(orderString.length()-9);
  detailBuffer.append(orderString.substring(9));

  // now invoke submit() and pass the 2 parms
  // note that the toString( ) method of StringBuffer is
  // invoked

  submitOrder(headerBuffer.toString(),detailBuffer.toString());
  return;
}
```

After setting up the two parameters, prepareOrder() starts the submitOrder() method. SubmitOrder() calls the ORDENTR program on the AS/400 system using the Distributed Program Call class that is provided by the AS/400 Toolbox for Java:

```
public void submitOrder(String header, String detail)
{
    updateStatus
          ("Processing...if you hang here check QZRCSRVS");
    try
    {
       as400.connectService(AS400.COMMAND);
       ProgramCall ordEntrPgm = new ProgramCall(as400 system);
       QSYSObjectPathName pgmName =
       new QSYSObjectPathName("APILIB","ORDENTR","PGM");

       ProgramParameter[] parmList = new ProgramParameter[2];
      // set the first parameter which is the order header
      // we use the data conversion classes in the Toolbox to
      // do this.
      AS400Text text1 = new AS400Text(9);
      byte[] headerBytes = text1.toBytes(header);
      parmList[0] = new ProgramParameter(headerBytes);

      AS400Text text2 = new AS400Text(detail.length());
      byte[] detailBytes = text2.toBytes(detail);
      parmList[1] = new ProgramParameter(detailBytes);

      ordEntrPgm.setProgram(pgmName.getPath(),parmList);

      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. First, we start the AS400.COMMAND service for the as400 object.

Note: The AS/400 Toolbox for Java implicitly starts this if it needs to be started. It is shown here for clarification:

```
as400.connectService(AS400.COMMAND);
```

We declare a ProgramCall object called ordEntrPgm. Then, we set the name of the AS/400 system program associated with this object by declaring and initializing a QSYSObjectPathName object:

```
ProgramCall ordEntrPgm = new ProgramCall(as400 system);
  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 AS/400 system data type object. In this case, we are passing string data, so we declare an AS400Text object.

   b. Start 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(9);
      byte[] headerBytes = text1.toBytes(header);
      ```

3. Start 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 AS/400 program that is being called. The parameter list must also be specified by starting the setProgram() method:

```
ordEntrPgm.setProgram(pgmName.getPath(),parmList);
```

We are now ready to run the program. The run() method starts 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 AS/400 system 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 AS/400 system. 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.

6.4 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: either as a native application with 5250 screen interaction or 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.

6.4.1 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 starting an SQL stored procedure using JDBC. The stored procedure is actually an RPG program called SLTCUSTR. 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 SLTCUSTR, 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 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 starts from a Java client. The partial line is shown here:

```
  ...WORKSTN SFILE(CUSTSFL:SflRrn) USROPN
```

It is interesting to note that the functionality handled by the BldSfl subroutine is analogous to the processing done by the SltCustWdw.populateCustBox() method. The functionality handled by the Process subroutine is handled by the SltCustWdw.custSelected() and OrderEntryWdw2.setSelectedCust() methods.

6.4.2 Providing an Item List

We saw earlier that the List Items button starts the SLTPARTR stored procedure. The changes made in SLTPARTTR 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 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.

6.4.3 Verifying an Item

From the client, an item is verified using the direct RECORDACCESS capability that the AS/400 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.

6.4.4 Processing the Submitted Order

As previously discussed, the AS/400 system program that handles a request to submit an order is ORDENTR. When the Order Entry application is run from an AS/400 system 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 starts 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 started 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:SflRrn) USROPN
```

The mainline logic of the program is changed to check the number of parameters. If there are parameters, a new subroutine, called CmtOrder2, starts 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_O
```

```
C                   MOVE      PartDscX      PARTDSC_O
C                   MOVE      PartQtyX      PARTQTY_O
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.

6.5 Summary

The common thread pervasive across all the changes in the host RPG code deals with the display file processing. When ORDENTR starts 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 AS/400 system 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 7, “Moving the Server Application to Java” on page 133.

Chapter 7. Moving the Server Application to Java

So far, we have shown you how to move the user interface 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 application to Java on AS/400. We use the Java Remote Method Invocation (RMI) interface to allow the client Java program to interface with the server Java code. Figure 82 shows the new design.

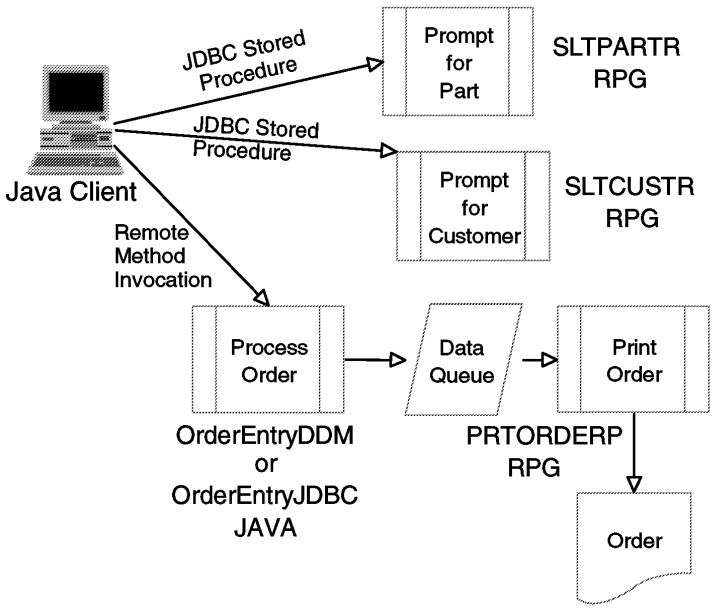


Figure 82. Java Client/Java Server Order Entry Application

This chapter 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 AS/400 JDBC. We also discuss the changes that are required for the client code.

There are three approaches to creating Java on AS/400 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 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. A full object-oriented version can be implemented later.

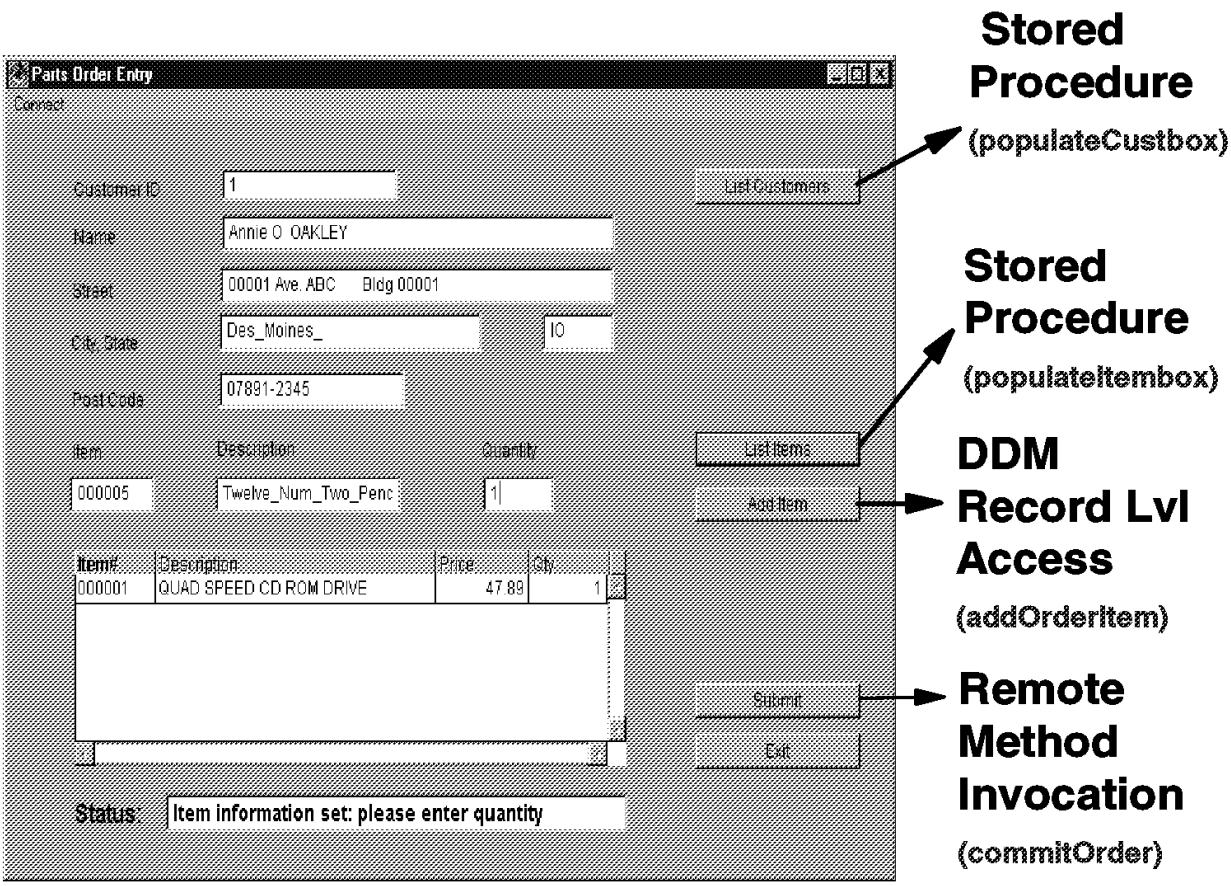


Figure 83. Java Client/Java Server Interfaces

The client Java program is comparable to the Java client program that was covered in Chapter 6, “Migrating the User Interface to Java Client” on page 101. The major difference is that now, the submit order processing is done through the Java RMI interface. In the following sections of this chapter, we 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 final 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 AS/400 database. These are the four choices:

- JDBC-ODBC bridge
- AS/400 Toolbox for Java JDBC running on the AS/400 system

- AS/400 Toolbox for Java Record Level Access (DDM) running on the AS/400 system
- AS/400 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 5, “Overview of the Order Entry Application” on page 87.

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.

Program Listings

For complete listings of all the code examples shown throughout this book, please refer to Appendix A, “Example Programs” on page 415. It also contains instructions on how to access the redbook Web site to download the example code.

7.1 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 OrderEntryDDM package. The class and its methods are shown in Figure 84.

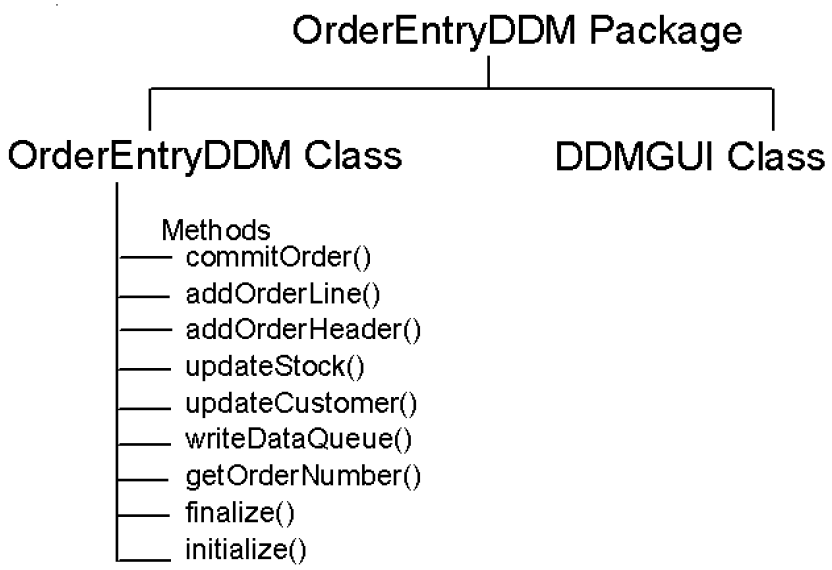


Figure 84. OrderEntryDDM Class

2. Specify that the class OrderEntryDDM is in the OrderEntry package:

```
package OrderEntryDDM;
```

3. Define the packages that are used by this class:

```
import com.ibm.as400.access.*; // for AS/400 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, your-password, and your-system 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 AS/400 session is 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 = "your-system";
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 the 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 data 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 of them 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;
}
```

7.1.1 Method Logic

Next, we add the program logic to each of the methods that we have created. We also need to consider exception handling. Many of the classes that we use to access the AS/400 system 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 AS/400 system.

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 exception with 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.

7.1.1.1 The initialize() Method

This is the complete initialize() method. This method is started by the constructor for the OrderEntry class. First, we create a connection to the AS/400 system (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 AS/400 system 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 AS/400 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.

Note: We must use the integrated file system naming convention.

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/PRODDATA.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/PRODDATA.LIB/ORDERS.FILE" on the system that is represented by the as400 connection.

Note: We are using the integrated file system naming convention even though we are connecting to the DB2/400 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 AS/400 system 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, and initializes it 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 our 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;
 }
```

7.1.1.2 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 6, “Migrating the User Interface to Java Client” on page 101. It contains this logic:

- Request the customer number and the number of order lines from the order object.
- Determine the next order number by starting an internal method.
- Pass the order and the order number to another internal 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 our 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, and 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.getNumEntries());

       // 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.");
}
```

7.1.1.3 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 conversions 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 our 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() == false)
  {
   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;
}
```

7.1.1.4 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. The first order line is extracted from the order object.
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 our 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() == false)
  {
       orderLineFile.open(AS400File.READ_WRITE, 0,
          AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
  }
  // Create an empty record
  Record orderLineRcd = orderLineFormat.getNewRecord();
```

```
  // Priming read
  OrderDetail orderLine = anOrder.getFirstEntry();

  // While we have order lines to process
  while(orderLine != null)
  {
       // 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),
                                 java.math.BigDecimal.ROUND_DOWN))).
                                 multiply(orderLine.getItemQty());
       orderLineRcd.setField("OLAMNT",
           orderAmount.setScale(2, BigDecimal.ROUND_HALF_UP));
       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.add(orderAmount);

       // Update the stock record
       updateStock(orderLine.getItemId(), orderLine.getItemQty());

       // Get the next order line
       orderLine = anOrder.getNextEntry();
  }

  return orderTotal;
}
```

7.1.1.5 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 our caller.

Any exceptions that the methods generate are simply passed back to our caller.

```
private BigDecimal getCustomerDiscount (String aCustomerID)
throws Exception
{
  // Open the file
  if (customerFile.isOpen() == false)
  {
       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;
  }
}
```

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 is increases and is updated in the file. The order number is returned to our 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 our caller.

```
private BigDecimal getOrderNumber () throws Exception
{
  System.out.println("getOrderNumber:");

  // Open the file
  if (districtFile.isOpen() == false)
  {
       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);
  }
}
```

7.1.1.6 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 our caller.

```
private void updateCustomer (String aCustomerID,BigDecimal
      anOrderTotal )
throws Exception
{
  // Get the current date and time
  java.util.Date currentDateTime = 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;
}
```

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 our caller.

```
private void updateStock (String aPartNbr,BigDecimal aPartQty )
    throws Exception
{
  // Open the file
  if (stockFile.isOpen() == false)
  {
       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;
}
```

7.1.1.7 The writeDataQueue() Method

The complete writeDataQueue() method follows this sequence of events:

1. Create a description of the data queue layout.

   First, it creates objects that represent the different data types that are used in the layout. Then, it creates an object that represents the AS/400 system data queue object. Next, it defines the layout of each record in the data queue by using the field definitions that were created earlier.

2. Populate the record with the values for the data queue.

3. Send the record to the data queue to initiate printing the order.

Any exceptions that the methods generate are simply passed back to our 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), "customerID" .);
  PackedDecimalFieldDescription as4DistrictID =
     new PackedDecimalFieldDescription(new AS400PackedDecimal(3,0),
        "districtID");
  CharacterFieldDescription as4WarehouseID =
     new CharacterFieldDescription(new AS400Text(4),
        "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;
}
```

7.1.1.8 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));
}
```

7.1.1.9 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));
}
```

7.1.2 Cleaning Up

After an order has been processed, we need to perform some clean up, such as closing files, dropping the connection to the AS/400 system, 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 AS/400 system and start the finalize() method of our 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 AS/400 system. If the Order Entry application is transportable, it is better to use portable Java classes. We can accomplish this by by using JDBC as our database access mechanism. We discuss this in the next section.

7.2 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.

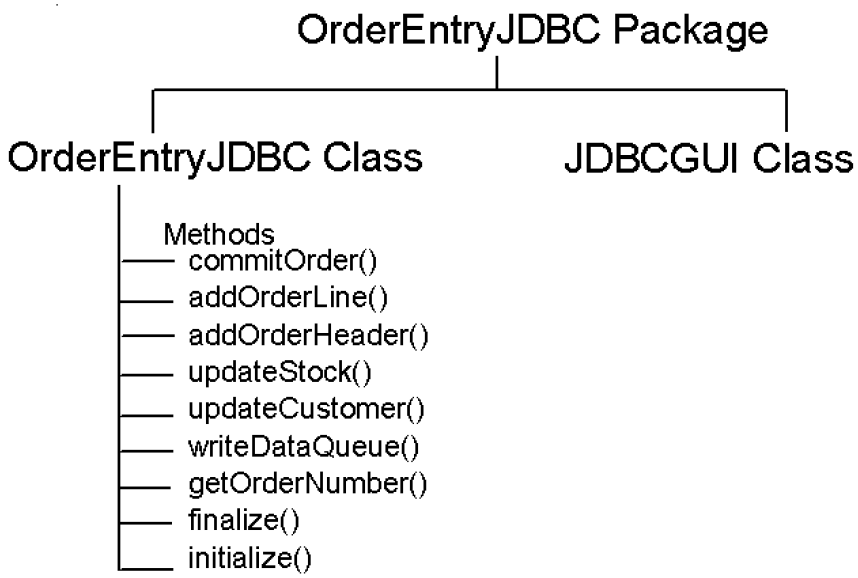


Figure 85. OrderEntryJDBC Class

OrderEntryJDBC is a platform-independent version of the OrderEntry class, as shown in Figure 85. We achieve platform independence by using JDBC, which shields our application from platform unique considerations.

There are three options of JDBC available to us:

- JDBC through the ODBC bridge
- JDBC through the AS/400 Toolbox for Java
- JDBC through the AS/400 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:"
- AS/400 Toolbox for Java JDBC — "jdbc:as400://"
- AS/400 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 starting 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 Class.forName() is that a ClassNotFound 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 AS/400 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 OrderEntry;

import com.ibm.as400.access.*; // for AS/400 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 = "your-system";
  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;
}
```

7.2.1 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.

This is the complete JDBC initialize() method. This method is started by the constructor for the OrderEntry class. It uses a different technique to create a connection to the AS/400 system. 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 AS/400 system 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 AS/400 system 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 AS/400 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 AS/400 system 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, OLDID, OLWID, OLNBR, OLSPWH, OLIID," +
          " OLQTY, OLAMNT, OLDLVD, 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;
}
```

7.2.1.1 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.getNumEntries());

        // 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.");
}
```

7.2.1.2 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 currentDateTime = 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(currentDateTime)+
                    ", "+getRawTime(currentDateTime)+")";

     addOrderHeader.executeUpdate(sql);
     return;
}
```

7.2.1.3 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());

  // Priming read
  OrderDetail orderLine = anOrder.getFirstEntry();

  // While we have order lines to process
     while(orderLine != null)
     {
       // 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),
                                 java.math.BigDecimal.ROUND_DOWN))).
                                 multiply(orderLine.getItemQty());
       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.add(orderAmount);

        // Update the stock record
        updateStock(orderLine.getItemId(), orderLine.getItemQty());

        // Get the next order line
        orderLine = anOrder.getNextEntry();
    }
    return orderTotal;
}
```

7.2.1.4 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 that we are interested in. 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), a 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 field is a packed decimal file.

We return the value of the customer discount to our 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", 4);

  return customerDiscount;
}
```

If we need to process many rows from the result set, the Java code looks similar to this example:

```
while( rs.next() != null )
{
     // do row processing
}
```

7.2.1.5 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. A performance penalty for raising the exception or a maintenance concern with global variables.

First, we create two SQL statement objects: one for fetching the records and one for updating the records. We name the cursor, so we can reference it in the UPDATE statement.

Then, 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 our caller.

```
private BigDecimal getOrderNumber ()
throws Exception
{

  // 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", 0);

  // 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;
}
```

7.2.1.6 The updateCustomer() Method

This is another method that uses the positioned update technique that was described earlier. Again, we create statement objects, name the cursor, 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 currentDateTime = 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", 2);
  BigDecimal yearToDateSales = rs.getBigDecimal("CYTD", 2);

  // Update the current balance and year to date sales (positioned update)
  sql = "UPDATE CSTMR "+
       "SET CLDATE = "+getRawDate(currentDateTime)+
       ", CLTIME = "+getRawTime(currentDateTime)+
       ", CBAL = "+(currentBalance.add(anOrderTotal)).toString()+
       ", CYTD = "+(yearToDateSales.add(anOrderTotal)).toString()+
       " "+
       "WHERE CURRENT OF CUSTOMER";

  setCustomer.executeUpdate(sql);
  return;
}
```

7.2.1.7 The updateStock() Method

Again, we use the positioned update technique that was described earlier. We create statement objects, name the cursor, 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", 0);

  // Update the order number (positioned update)
  sql = "UPDATE STOCK "+
    "SET STQTY = "+ stockQty.subtract(aPartQty).toString() +" "+
      "WHERE CURRENT OF STOCK";

  setStockQty.executeUpdate(sql);

  return;
}
```

7.2.1.8 The writeDataQueue() Method

No changes are required to this method.

7.2.1.9 The getRawDate() Method

No changes are required to this method.

7.2.1.10 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.

7.2.2 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;
    }
```

7.3 Remote Method Invocation Support

At this point, we have successfully created two Java classes that are functionally equivalent to the RPG Order Entry program we started with. 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 start from a client front-end and we cannot do that yet, because there is no linkage to the client.

Remote Method Invocation (RMI) is the mechanism that Java uses to support starting 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:

- The class must be a subclass of the UnicastRemoteObject class.
- The class must implement an interface that describes the public methods.
- The interface must be a subclass of the Remote class.
- The interface must describe each public method.
- The interface methods must throw RemoteException.
- An RMI registry must be running on the server.
- An instance of the class must register with the registry.
- The client and the 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, but 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.

```
package OrderEntry;

import java.rmi.*;
/**
 * This interface was generated by a SmartGuide.
```

```
 *
 */
public interface OrderEntryI extends Remote
{
public String commitOrder(Order anOrder) throws RemoteException;
}
```

7.3.1 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 AS/400 program (OrderEntryDDM or OrderEntryJDBC) through a class named OrderSubmitter, as shown in Figure 86. An Order object is passed from the client program to the server program.

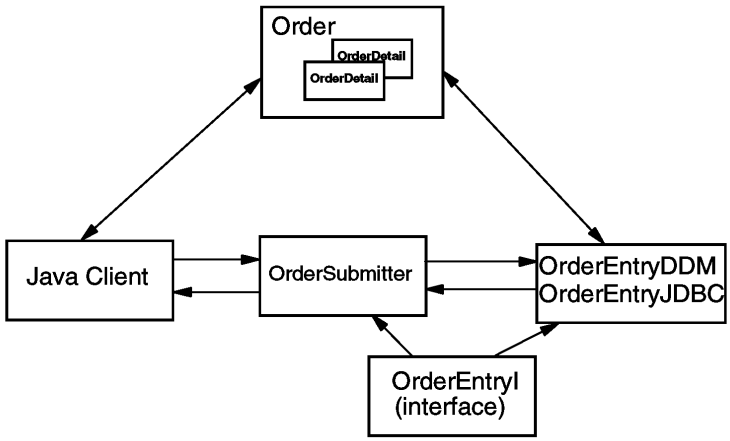


Figure 86. RMI Application Design

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 87.

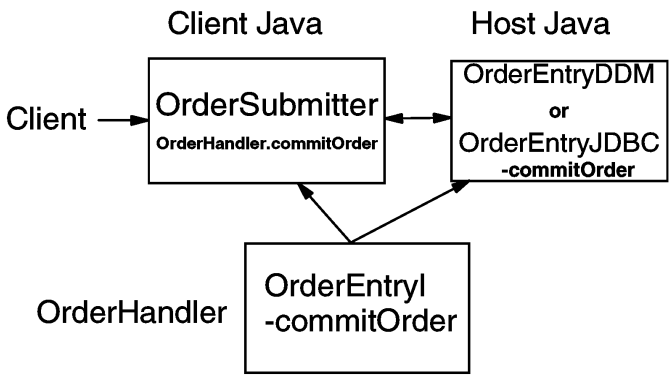


Figure 87. RMI Interface

7.3.2 Adding RMI Support to Server Classes

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 162.

```
import java.rmi.*; // for Remote Method Invocation
import java.rmi.registry.*;
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
     {
       System.setSecurityManager(new RMISecurityManager());
       OrderEntryJDBC oeJDBC = new OrderEntryJDBC();
       Naming.rebind("//"+SYSTEM+"/OrderEntryJDBC", oeJDBC);
     } catch(Exception e) {e.printStackTrace();}

     return;
}
```

This block of code perform these functions:

- Creates a new security manager
- Creates a new instance of our own class
- Binds the new object as a service with an appropriate name (in this case, OrderEntryJDBC)

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.

Once you create the proxies, you must start the RMI registry. This must be done from the shell. Use the **rmiregistry** shell command to start 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 do choose to start 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/OrderEntryJDBC", 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 OrderEntry.OrderEntryJDBC
```

7.3.3 Adding RMI Support to the Client

In Chapter 6, “Migrating the User Interface to Java Client” on page 101, 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 AS/400 Toolbox for Java, was 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 oject, 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 6, “Migrating the User Interface to Java Client” on page 101, 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 start the remote methods that are made available through these new classes.

7.3.4 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.*;
import java.rmi.RMISecurityManager;

public class OrderSubmitter
{
     private String serverURL = null;
     private OrderEntryI orderHandler = null;
}
```

This class contains only two data members. 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 the previously discussed OrderEntryI and OrderEntryJDBC classes for details.

```
public boolean linked(String hostServer, String port)
{
     // set an RMISecurityManager - if we have none
     if(System.getSecurityManager() == null)
     {
        System.setSecurityManager(new RMISecurityManager());
   }

     // obtain reference to the remote OrderEntryJDBC object
     try     {
       serverURL = "//" + hostServer + "/OrderEntryJDBC"
       orderHandler = (OrderEntryI)Naming.lookup(serverURL);
   }

   catch(Exception e)
   {
        e.printStackTrace();
        return(false);
   }
     return(true);
}
```

The linked() method handles two primary tasks. First, it sets an RMISecurityManager for the session. Next, it obtains a reference to the remote object by starting the Naming.lookup() method. Once these two steps are completed, the application can start a remote method. The hostServer is passed in as a parameter. This is the same value that was retrieved from the sign-on dialog discussed in Chapter 6, “Migrating the User Interface to Java Client” on page 101.

The 'OrderEntryJDBC' string is the name that the OrderEntryJDBC class is registered as on the host machine. See the OrderEntryJDBC class.

The OrderSubmitter class imbeds the RMI in its own submit() method:

```
public String submit(Order theOrder)
{
  String status;
```

```
  try
  {
       status = orderHandler.commitOrder(theOrder);
  }
  catch(Exception e)
  {
       e.printStackTrace();
       status = "Error invoking remote method";
       return(status);
  }

  return(status);
}
```

The submit() method simply starts 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.

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:

```
import java.io.Serializable;
public class Order implements Serializable
{
     private StringBuffer customerId = new StringBuffer(4);
     private OrderDetail[] entryArray = new OrderDetail[50];
     private int index = -1;
     private int cursor = 0;
}
```

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.

7.3.5 Conclusion

To allow the Java client to interface with the new AS/400 server Java classes, we use RMI. In this section, we discussed the changes that are necessary to the client code and the server code to support RMI. We covered implementing RMI for the JDBC example (OrderEntryJDBC). We can also use the same methodology to implement an RMI interface between the Java client and the OrderEntryDDM class.

Chapter 8. Structured Query Language for Java

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. This chapter explains using SQLJ on the AS/400 system 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 AS/400 system 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 AS/400 system and a personal computer.
- **Order Entry Client Application**
  We convert the Order Entry client application, discussed in Chapter 6, “Migrating the User Interface to Java Client” on page 101, 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:
  – OrderEntryWdw2
  – SltCustWdw
  – SltItemWdw
- **Order Entry Server Application**
  We convert the host JDBC Java application discussed in Chapter 7, “Moving the Server Application to Java” on page 133, 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.

8.1 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 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 AS/400 system, 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 AS/400 system. Compiling a SQLJ program does not generate an access plan.

To run SQLJ statements, you must convert the SQLJ statements into JDBC statements, this is done by a pre-processor, which hides the complexities of the implementation.

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.

8.1.1 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 that you need to do 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 it needs to be converted into Java code using a pre-processor. This pre-processor converts the SQL statement into JDBC statements. For example, we have the following sqlj source file:

```
HelloWorld.sqlj
```

This member type needs to be pre-processed by the sqlj translator, for example:

```
Sqlj HelloWorld.sqlj
```

As a result of executing 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.

8.1.2 SQLJ and the AS/400 System

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 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

The translator.zip package contains the class that contains the actual pre-processor. 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 AS/400 system also works on a PC without any changes.

8.1.3 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, see this site: http://www.oracle.com/java/sqlj/index.html

8.2 The CstmrInq Example

CstmrInq is a Java program that reads the customer table (CSTMR) on the AS/400 system using SQL. The program runs on both an AS/400 system and on a PC without change. The program perform the following actions:

- Establishes a connection to the AS/400 system host database server using a JDBC driver. If the program is running on an AS/400 system, it loads the AS/400 native JDBC driver. If it is running on a PC, it loads the AS/400 Toolbox for Java JDBC driver.
- Sets the DefaultContext, which is a requirement to use SQLJ.
- Reads the CSTMR file using SQL and stores the result in host variables.
- Writes the information read from the CSTMR file to the display.

8.2.1 Setting Up the AS/400 System and PC for the CstmrInq Example

You need to set up both the AS/400 system and your workstation to use SQLJ with the CstmrInq example.

8.2.1.1 Setting Up the AS/400 System

To compile with the SQLJ translator and run SQL created classes, you need to include the SQLJ packages in your CLASSPATH. Setting up the CLASSPATH is explained in Section 3.4, “Setting Up the Environment” on page 57. You need these directories in the CLASSPATH:

- /QIBM/ProdData/Java400/ext/runtime.zip
- /QIBM/ProdData/Java400/ext/translator.zip

The SQLJ translator itself is a Java program. An example of compiling using the SQLJ pre-compiler is shown in Figure 88.

```
===> java sqlj.tools.Sqlj -status CstmrInq.sqlj
```

Figure 88. Compiling on the AS/400 System Using the SQLJ Pre-compiler

It creates the Java source and class files as well as some supporting classes as explained earlier.

8.2.1.2 Setting Up a PC

On a PC, you only need to include the SQLJ packages in your CLASSPATH. By assigning a network drive, you can directly use the SQLJ packages from the AS/400 system integrated file system.

As shown in Figure 89, once the classpath is set up properly, you use exactly the same method to compile on a PC as on an AS/400 system.

```
C:\>java sqlj.tools.Sqlj -status CstmrInq.sqlj
```

Figure 89. Compiling on a PC Using the SQLJ Pre-Compiler

The compile creates the same Java source, class, and support files as it does on the AS/400 system.

The files that you generate are exactly the same regardless of whether the compilation was done on the AS/400 system or on a PC. However, while writing this redbook, we found that the AS/400 system was much faster in compiling and was better at giving more detailed error messages.

8.2.2 Running CstmrInq

You can run the CstmrInq program on an AS/400 system or a PC. This section explains both scenarios.

8.2.2.1 Running CstmrInq on an AS/400 System

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 90.

```
===> java CstmrInq



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

Figure 90. Java Command to Run CstmrInq

Figure 91 on page 173 shows the display after you run the program.

```
$
> java CstmrInq
  Please enter Customer number between 0001 and 0010 as class initiating argume
  nt
  $

===>



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

Figure 91. Running a Java Class

Now, start the class again. However, this time give a customer number as an argument, as shown in Figure 92. The first name of the customer is displayed.

```
  $
> java CstmrInq 0001
  Please wait . . . accessing AS/400 CSTMR file . .
  The first name of Customer no 0001 is Annie
  $

===>



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

Figure 92. Entering a Customer Number

If you use a different customer number, as shown in Figure 93, the first name for that customer is displayed.

```
  $
> java CstmrInq 0003
  Please wait . . . accessing AS/400 CSTMR file . .
  The first name of Customer no 0003 is Piggy
  $

===>



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

Figure 93. Entering a Second Customer Number

To summarize, this Java program retrieves a single customer record at a time from AS/400 CSTMR file using SQLJ and Java. It writes the first name from the record retrieved to the display.

8.2.2.2 Running CstmInq on a PC

To run CstmInq on a PC, make sure that you change the current directory to where the classes are located or add the directory that contains the classes to your path, as shown in Figure 94.

```
C:\>f:

F:\>cd home\a999501a

F:\HOME\A999501A>
```

Figure 94. Changing the Current Directory

To run the class, simply type the `java` command followed by the class name, as shown in Figure 95.

```
F:\HOME\A999501A>java CstmrInq
```

Figure 95. Running the Class

After you run the class, the message shown in Figure 96 is displayed.

```
F:\HOME\A999501A>java CstmrInq
Please enter Customer number between 0001 and 0010 as class initiating argument
```

Figure 96. Displaying the Customer Number Message

Now, start the class again, but this time with a customer number, as shown in Figure 97.

```
F:\HOME\A999501A>java CstmrInq  0001
Please wait . . . accessing AS/400 CSTMR file . .
The first name of Customer no 0001 is Annie
```

Figure 97. Starting the Class with a Customer Number

Try starting the class again with a different customer number, as shown in Figure 98.

```
F:\HOME\A999501A>java CstmrInq  0003
Please wait . . . accessing AS/400 CSTMR file . .
The first name of Customer no 0003 is Piggy
```

Figure 98. Starting the Class with Another Customer Number

The message displayed is the same as when running the program on an AS/400 system. Any conversions, for example from EBCDIC to ASCII, are automatically handled by JDBC and SQLJ. You do not have to do any coding to handle these conversions. This example demonstrates that the same program runs on both a

Personal Computer and on the AS/400 system, without change, to produce exactly the same results.

8.2.3 CstmrInq Explanation

Next, we explain the CstmrInq program using code snippets. Figure 99 shows the import statement for the package named java.sql. It is a standard Java package that is required whenever you use JDBC. This package is also required for a SQLJ program. The other three packages that start with sqlj are the SQLJ specific packages, which are required to use SQLJ.

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.*;
import sqlj.runtime.ref.*;
import sqlj.runtime.ref.DefaultContext;

public class CstmrInq
{
        static public Connection dbConnect = null;
        static public String USER          = "a999501a";
        static public String PASSWORD      = "java1a";
        static public String SYSTEM        = "AS20";
        static public String LIBRARY       = "apilib";
```

Figure 99. CustmrInq Cass Description

In the getJdbcConnection() method that is shown in Figure 100 on page 176, a JDBC connection is established with the AS/400 host server. It is a generic method, which allows the program to run on both a PC and AS/400 system without modification. If the Java program runs on an AS/400 system, then the AS/400 native JDBC driver is loaded. Otherwise, if the program runs on a PC, the AS/400 Toolbox for Java JDBC driver is loaded.

```
static public void getJdbcConnection() throws SQLException
    {
            String url = "jdbc:db2:" + SYSTEM;

            try
            {
                    Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            }
            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);
                  }
            }
            dbConnect = DriverManager.getConnection(url, USER, PASSWORD);
    }
```

Figure 100. The getJdbcConnection() Method

In the first `try` block, the AS/400 JDBC driver is loaded. If you initiate this program from the AS/400 system, the load is successful and you establish the connection. If the Java class runs on a PC, the load fails and a "ClassNotFoundException" is generated. This exception is caught and handled in the `catch` statement. Here, another driver, which is the AS/400 Toolbox for Java JDBC driver is loaded. For the PC, this driver is loaded and the connection is established.

The initContext() method, as shown in Figure 101, is a unique requirement for SQLJ. The classes in the SQLJ runtime packages use the DefaultContext object that is created in this method.

```
static public DefaultContext initContext() throws SQLException
       {
              DefaultContext ctx = DefaultContext.getDefaultContext();
              if (ctx == null)
              {
                      ctx = new DefaultContext(dbConnect);
                      DefaultContext.setDefaultContext(ctx);
              }
              return ctx;
       }
```

Figure 101. The initContext() Method

If we look at the class description for this program, the DefaultContext class is imported into the CstmrInq program. You need to instantiate a DefaultContext object after the JDBC connection is established. The code to create the DefaultContext object is shown in Figure 102 on page 177.

```
ctx = new DefaultContext(dbConnect);
DefaultContext.setDefaultContext(ctx);
```

Figure 102. Setting the DefaultContext Object

The dbConnect object is an object that contains the JDBC connection information and was created by the getJdbcConnection() method. It is needed as input to the constructor of the DefaultContext object. Without the DefaultContext object, the SQLJ application cannot run because the SQLJ runtime support uses this object to run the Java program.

Figure 103 shows the complete code of the main method in which the actual SQLJ statement is executed.

```
public static void main (String args[]) throws SQLException
       {
          if(args.length > 0)
          {
          System.out.println("Please wait . . . accessing AS/400 CSTMR file
. . ");

      getJdbcConnection();
      initContext();

      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(1);
       }
```

Figure 103. The main Method

The statement, shown in Figure 104, starts with `#sql`. This symbol denotes that it is an SQLJ statement that is used for pre-processing by the SQLJ pre-processor. The SQLJ pre-processor 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 104. SQLJ Statement

The statement within {} is the actual SQL statement. This is the standard ANSI static SQL. This statement reads the CSTMR AS/400 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. All of the conversions from the Java String type to AS/400 EBCDIC data type and reverse are handled automatically by SQLJ and JDBC. You do not need to handle these conversions. If we did not use JDBC or SQLJ, but another technique such as JNI, we would have to handle these conversions as explained in Chapter 9, “Java Native Interface” on page 205.

In the main method, shown in Figure 103 on page 177, we use an `if` statement to check if a parameter was passed in. Otherwise, we display a message: `Please enter a customer number between 0001 and 0010`.

The next step is to establish the JDBC connection and create the DefaultContext object. Then, the host variables are set, and the SQLJ call to the AS/400 database is made. The host variable that is returned from the database call is shown on the display.

To summarize, this Java program uses SQLJ to retrieve data from an AS/400 table and displays it. The program runs on both the AS/400 system and a PC without change.

8.3 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 AS/400 system and on a PC without change. The program performs the following tasks:

- Sets the iterator (or cursor) to store a group of records. This is done by using an SQLJ statement.
- Establishes a connection to the AS/400 system by using the JDBC driver. If the program is running on the AS/400 system, it loads the AS/400 native JDBC driver. If it is running on a PC, it loads the PC AS/400 Toolbox for Java JDBC driver.
- Sets the DefaultContext, which is a requirement to use SQLJ.
- 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.
- 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.

8.3.1 Setting Up the AS/400 System and a PC for the CstmrList Example

The setup is exactly the same as shown in Sections 8.2.1.1, “Setting Up the AS/400 System” on page 171, and 8.2.1.2, “Setting Up a PC” on page 172.

8.3.2 Running CstmrList on the AS/400 System and PC

You can run the CstmrList program on an AS/400 system or a Personal Computer. This section explains how to run the program in both environments.

8.3.2.1 Running CstmrList on an AS/400 System

Figure 105 on page 179 shows running the Java program, called CstmrList, 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

   Please wait . . . accessing AS/400 CSTMR file . .
    Cust no 0001 is Annie           OAKLEY           from Des_Moines_        IO
    Cust no 0002 is Elizabeth       BARBER           from Maryville_         TX
    Cust no 0003 is Piggy           ABLE             from Denver_            CO
    Cust no 0004 is NEIL            WILLIS           from Concord_           AR
    Cust no 0005 is Kareem          MULLEN-SCHULTZ   from Minneapolis_       MN
    Cust no 0006 is Bob             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
---------------------------- End of CSTMR -------------------------------
   $

 ===>



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

Figure 105. Retrieving a Group of Records on the AS/400 System

8.3.2.2 Running CstmrList on a PC

To run this program on a PC, make sure that the CLASSPATH is properly set up and enter the command, as shown in Figure 106.

```
F:\HOME\A999501A>java CstmrList
Please wait . . . accessing AS/400 CSTMR file . .
 Cust no 0001 is Annie           OAKLEY           from Des_Moines_        IO
 Cust no 0002 is Elizabeth       BARBER           from Maryville_         TX
 Cust no 0003 is Piggy           ABLE             from Denver_            CO
 Cust no 0004 is NEIL            WILLIS           from Concord_           AR
 Cust no 0005 is Kareem          MULLEN-SCHULTZ   from Minneapolis_       MN
 Cust no 0006 is Bob             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
---------------------------- End of CSTMR -------------------------------
```

Figure 106. Retrieving a Group of Records on a PC

The data type conversions from String to ASCII or EBCDIC are all handled transparently by JDBC and SQLJ. This program demonstrates implementing SQLJ selection by using a cursor. It also shows that the program runs both on the AS/400 system and on a PC to produce exactly the same result.

8.3.3 CstmrList Explanation

The basic explanation of SQLJ statements that was covered in the previous CstmrInq example in Section 8.2.3, “CstmrInq Explanation” on page 175, 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 107, 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 pre-processor 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.*;
import sqlj.runtime.ref.*;
import sqlj.runtime.ref.DefaultContext;

#sql iterator CustCursor (String id, String firstname, String lastname, String
city, String state);

public class CstmrList
{
        static public Connection dbConnect = null;
        static public String USER          = "a999501a";
        static public String PASSWORD      = "java1a";
        static public String SYSTEM        = "AS20";
        static public String LIBRARY       = "apilib";
```

Figure 107. CstmrList Class Code

The getJdbcConnection() method, shown in Figure 108, is exactly the same as the one in the CstmrInq program. Refer to the explanation of this method in the description of the CstmrInq example shown in Section 8.2.3, “CstmrInq Explanation” on page 175.

```
static public void getJdbcConnection() throws SQLException
   {
           String url = "jdbc:db2:" + SYSTEM;

           try
           {
                   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
           }
           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);
                   }
           }
           dbConnect = DriverManager.getConnection(url, USER, PASSWORD);
   }
```

Figure 108. The getJdbcConnection() Method

The method initContext(), shown in Figure 109 on page 181, is exactly the same as the one in the CstmrInq class. Refer to the explanation of this method in the

description of the CstmrInq class example shown in Section 8.2.3, “CstmrInq Explanation” on page 175.

```
static public DefaultContext initContext() throws SQLException
       {
              DefaultContext ctx = DefaultContext.getDefaultContext();
              if (ctx == null)
              {
                     ctx = new DefaultContext(dbConnect);
                     DefaultContext.setDefaultContext(ctx);
              }
              return ctx;
       }
```

Figure 109. The initContext() Method

Figure 110 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 executed to establish the JDBC connection and set the DefaultContext.

```
public static void main (String args[]) throws SQLException
   {

          System.out.println("Please wait . . . accessing AS/400 CSTMR file . .

          getJdbcConnection();
          initContext();

          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(1);
       }
}
```

Figure 110. The main Method

Each of the SQLJ related statements in the main method are explained individually in the following text.

The code, shown in Figure 111, declares an object based on the CustCursor class. This is the same class that is created by the SQLJ pre-processor because of the definition of the iterator at the beginning of this class.

```
CustCursor cCursor;
```

Figure 111. Creating an Instance of the CustCursor Class

In Figure 112, 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 **id** 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 112. 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 113.

Note: You 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 113. Retrieving All of the Records

Finally, the main program ends with the `System.exit` statement. This statement releases the AS/400 connection and the thread. Otherwise, it is possible for the Java class to not end and just keep waiting because of the active JDBC connection.

To summarize, in this example, the following actions occurred:

- Defined an iterator
- Established a JDBC connection
- Set the default context

- 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

8.4 Order Entry Client Application

In this section, we take the client side of the Order Entry application that was discussed Chapter 6, “Migrating the User Interface to Java Client” on page 101 and convert 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:

- OrderEntryWdw2
- SltCustWdw
- SltItemWdw

Figure 114 shows the new application design.

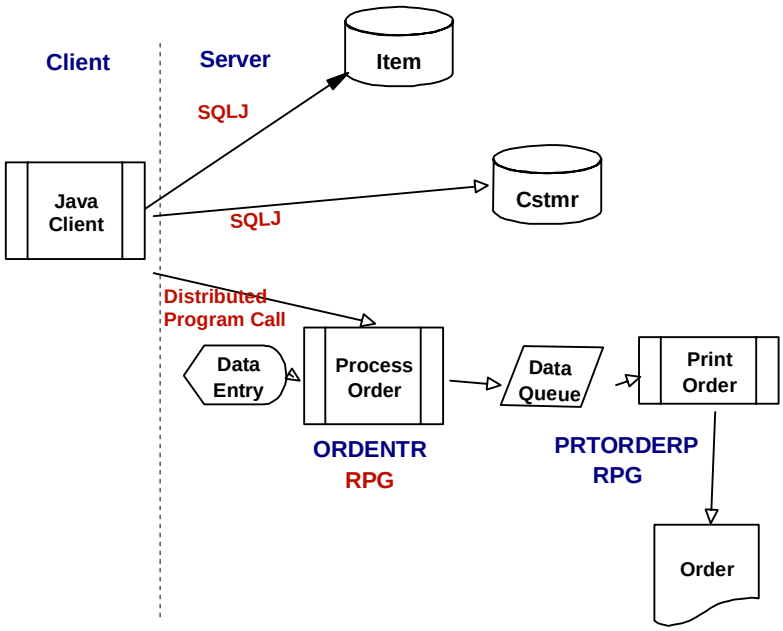


Figure 114. Order Entry Client Using SQLJ

Figure 115 on page 184 shows the main Order Entry window and the interfaces used.

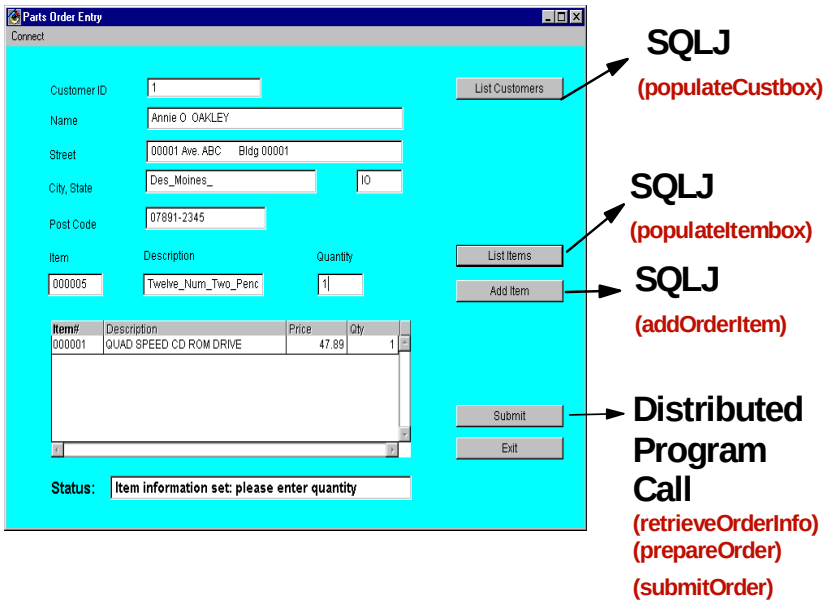


Figure 115. Order Entry Client Interfaces

8.4.1 Converting the OrderEntryWdw2 Class to Use SQLJ

This section shows the code that we add, change, and remove to use SQLJ in this application.

8.4.1.1 Additions to the OrderEntryWdw2 Program

You need to import the sqlj packages, shown in Figure 116, into the OrderEntryWndw2 program, because they are required for the runtime support of SQLJ.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import sqlj.runtime.ref.DefaultContext;
```

Figure 116. Importing the sqlj Packages

You need to declare the Connection object as static, as shown in Figure 117.

```
private static Connection dbConnect = null;
```

Figure 117. Declaring the Connection Object

You need to set the DefaultContext, as shown in Figure 118 on page 185.

```
static public DefaultContext initContext()
  {
        DefaultContext ctx = DefaultContext.getDefaultContext();
        if (ctx == null)
        {
          try
                {
                ctx = new DefaultContext(dbConnect);
                }
          catch (SQLException e)
                {
                  System.out.println("Error: could not get default context");
                  System.err.println(e);
                  System.exit(1);
                }
          DefaultContext.setDefaultContext(ctx);
        }
        return ctx;
  }
```

Figure 118. Setting the DefaultContext

You also need to add a statement in the connectToDB method to execute the initContext() method shown in Figure 118. For a detailed description of this method, see the CstmrInq example in Section 8.2.3, “CstmrInq Explanation” on page 175.

This program also reads the ITEM file to verify that an item being ordered in valid. Figure 119 shows how we use SQLJ to validate an item.

```
        String[]  orderRow      = new String[4];
        String    oid           = new String();
        String    oname         = new String();
        String    oprice        = 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;
                getIMulticolumnListbox1().addRow(orderRow, orderRow[0]);
                getIMulticolumnListbox1().repaint();
                getSubmitBTN().setEnabled(true);
                updateStatus("Item added: please add more or submit");
```

Figure 119. Adding Statements to Read the ITEM Table

In the statements that are shown in Figure 119, 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 data members are required. This is the reason retrieved variables are first loaded into simple String data members and then these data members are loaded into array cells. The array is used as a row to populate the multi-column list box for the display.

8.4.1.2 Removal from the OrderEntryWdw2 Class

We need to either comment out or remove the following statements from the OrderEntryWdw2 program to change from using JDBC stored procedures and DDM record level access to use SQLJ.

The Connection object is used in the initContext() method, which is a static method. The statement in Figure 120 needs to be removed and a statement with a static Connection object needs to be added, as described in Section 8.4.1.1, “Additions to the OrderEntryWdw2 Program” on page 184.

```
private static Connection dbConnect = null;
```

Figure 120. Original Connection Object

In the OrderEntryWdw2 class, DDM was used to access the ITEM files. DDM requires that an as400 object be initialized and an AS400 object connection be established. You must declare files, formats, keys, and so on. You have to open the file before it can be accessed using DDM. Figure 121 on page 187 shows the code for opening the DDM file. Since we no longer will use DDM, we can eliminate this code.

```
private void openItemFile() {
        // initialize the as400 data member
        as400 = new AS400(systemName, userid, password);

        // 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 AS/400
        itemFile = new KeyedFile(as400, fileName.getPath());

        try
        {
                as400.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 1
  try
  {
           AS400FileRecordDescription recordDescription = new
           AS400FileRecordDescription (as400,"/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;
}
```

Figure 121. Original DDM Code

The openItemFile() method is invoked from two places. One invocation statement is in the connectToDB() method and the other is in the addOrderItem() method. We need to comment out or remove these statements. With SQLJ, none of the above steps are required, because all of the linking and conversions are done by the JDBC driver.

The code in Figure 122 shows how the addOrderItem() method uses DDM. We remove this method because it is replaced by the addition of functionally equivalent SQLJ code.

```
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) {
              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();
              getIMulticolumnListbox1().addRow(orderRow, orderRow[0]);
              getIMulticolumnListbox1().repaint();
              getSubmitBTN().setEnabled(true);
              updateStatus("Item added: please add more or submit");
       } else {
             updateStatus("Invalid item...");
       }
```

Figure 122. AddOrderItem() DDM Code

8.4.2 Converting SltCustWdw to Use SQLJ

In this section, we convert the SltCustWdw program from using JDBC stored procedures to access the CSTMR table to use SQLJ. As shown in Figure 123 on page 189, this program is used to display a list of customers to the end user.

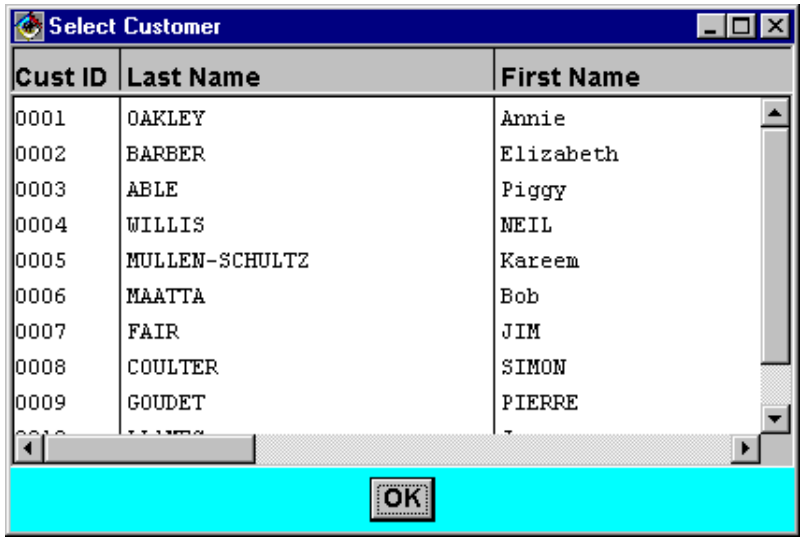


Figure 123. SltCustWdw Display

8.4.2.1 Addition to the SltCustWdw Class

The following code segments are added in the SltCustWdw class to support SQLJ. You must add the `import` statements shown in Figure 124 because they contain the classes that the SQLJ runtime support uses.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import sqlj.runtime.ref.DefaultContext;
```

Figure 124. Importing the Required SQLJ Classes

Figure 125 defines the iterator or the cursor class CustList. For a detailed explanation, see the CstmrList class example in Section 8.3.3, “CstmrList Explanation” on page 179.

```
#sql iterator CustList (String id,
                        String last,
                        String first,
                        String init,
                        String addr1,
                        String addr2,
                        String city,
                        String state,
                        String zip);
```

Figure 125. Defining the CustList Class

In Figure 126 on page 190, an instance of the iterator CustList class is created. A group of records from the CSTMR file is loaded into the cList instance. By using a while loop and getter methods, the data members are retrieved from the cList instance and loaded into the multi-column list box for display.

```
// Begin addition for SQLJ
        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  };

                  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();
                       getIMulticolumnListbox1().addRow(array, array[0]);
                  }
}
```

Figure 126. Creating an Instance of the CustList Class

8.4.2.2 Removal from the SltCustWdw Class

We need to comment out or remove the following statements in the SltCustWdw class to covert from using JDBC stored procedures to use SQLJ. In Figure 127 on page 191, the JDBC `prepareCall` statement is used to call an RPG stored procedure, which returns a group of records that are loaded into the multi-column list box. In the new program, we use SQLJ to retrieve the records. Therefore, we can remove the following statements.

```
        ResultSet rs = null;
        CallableStatement callableStatement = null;
        try {
// invoke the stored procedure on the AS/400
   callableStatement = dbConnect.prepareCall("CALL APILIB.SLTCUSTR(' ','R')");
   rs = callableStatement.executeQuery();
   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");
          getIMulticolumnListbox1().addRow(array, array[0]);
```

Figure 127. Removing the Statements that Use the JDBC prepareCall Method

8.4.3 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 128, this program is used to display a list of items to the end user.

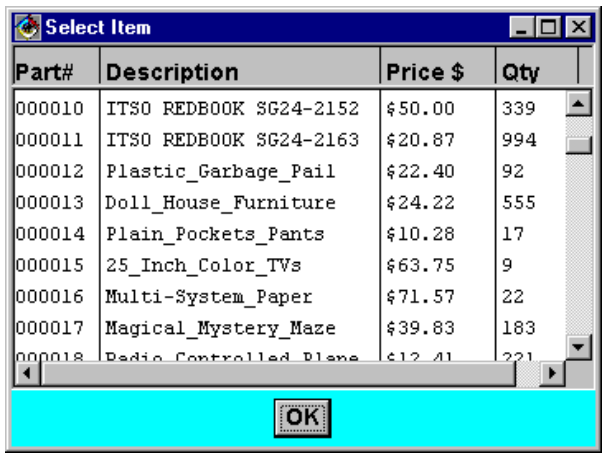


Figure 128. SltItemWdw Display

8.4.3.1 Addition to the SltItemWdw Class

We need to add the following statements to the SltItemWdw class to support SQLJ. We must import the packages shown in Figure 129 to use SQLJ because they contain the classes that the SQLJ runtime support uses.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
```

Figure 129. Importing the Required Classes

Figure 130 defines the iterator or the cursor class ItemList. For a detailed explanation, see the CstmrList class example in Section 8.3.3, “CstmrList Explanation” on page 179.

```
#sql iterator ItemList (String ijid,
                        String iname,
                        String iprice,
                        String iqty);
```

Figure 130. Defining the ItemList Class

In Figure 131, 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 and load them into the multi-column list box for display.

```
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
                                      '0001' = STWID
                                      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();
                      getIMulticolumnListbox1().addRow(array, array[0]);
              }
```

Figure 131. Creating an Instance of the ItemList Class

8.4.3.2 Removal from the SltItemWdw Class

To convert SltItemWdw from using JDBC stored procedures to use SQLJ, we need to comment out or remove the following statements in the SltItemWdw program.

In Figure 132 on page 193, the JDBC prepareCall statement calls an RPG stored procedure, which returns a group of records that were loaded into the multi-column list box. In the new program, we use SQLJ to retrieve a group of records from the AS/400 system. Therefore, we can remove these statements.

```
ResultSet rs;
CallableStatement callableStatement;
try {
     // invoke the stored procedure on the AS/400
     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", 2).toString();
              array[3] = Integer.toString(rs.getInt("STQTY"));
              getIMulticolumnListbox1().addRow(array, array[0]);
}
```

Figure 132. Calling an RPG Stored Procedure Using JDBC

In this example, the JDBC prepareCall was used to call an RPG stored procedure, which returned a group of records that are loaded into the multi-column list box. In the new program, SQLJ is used to retrieve a group of records from the AS/400 system.

8.5 Order Entry Server Application

In this section, we convert the Order Entry server side application discussed in Chapter 7, “Moving the Server Application to Java” on page 133, 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 133 shows the new application design.

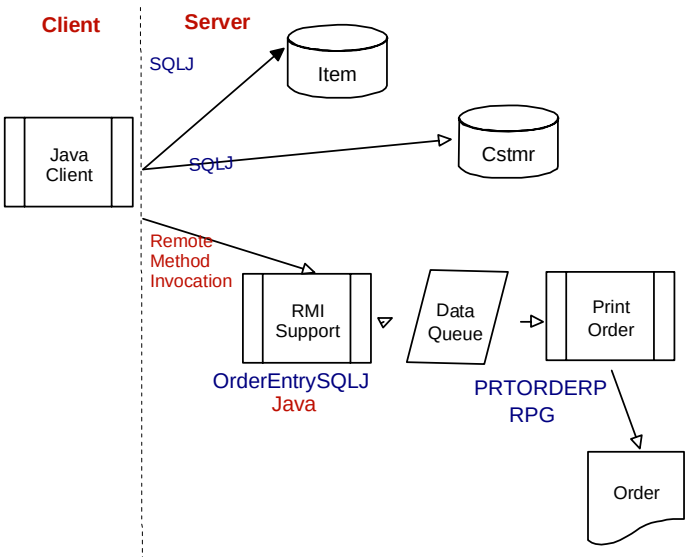


Figure 133. Order Entry Host SQLJ Application

Figure 134 on page 194 shows the main Order Entry window and the interfaces used.

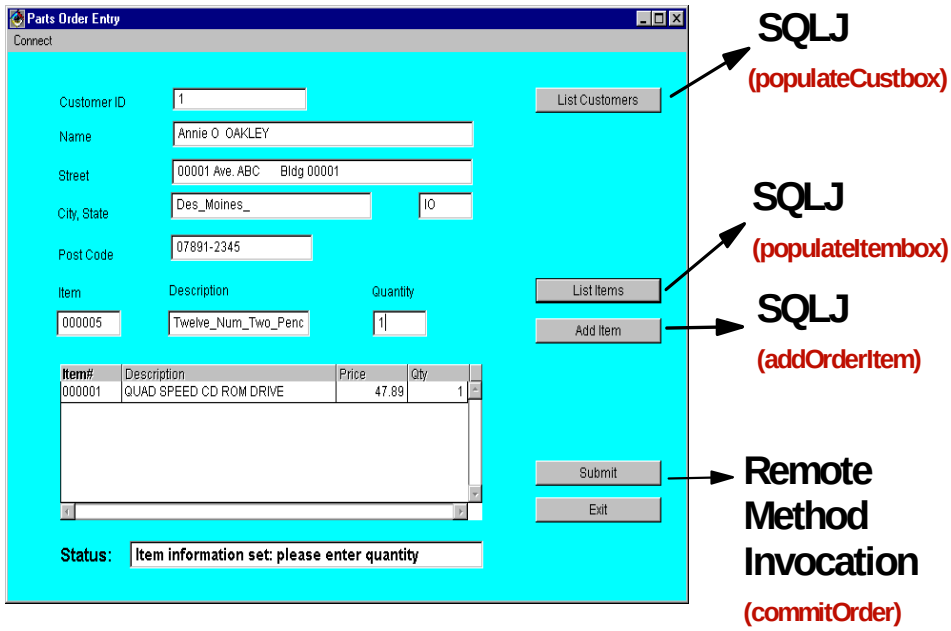


Figure 134. Order Entry Application Interfaces

8.5.1 Setting Up the Order Entry Server Application

The SQLJ application is based on the JDBC application named OrderEntryJDBC found in the OrderEntryJDBCPkg package. This package, which is available with the additional materials for this redbook, was exported from VisualAge for Java into the sqlj directory.

These packages were exported from VisualAge for Java, so we can use the existing application as a basis for the SQLJ application. This section shows you which statements you must add to use SQLJ and which JDBC statements you need to remove from the program source. We change the class and package name for this new application to the OrderEntrySQLJ class and the OrderEntrySQLJPkg package.

The current version of VisualAge for Java does not support SQLJ. We edit the program source using a Windows 95/NT editor. We compile the program using the Oracle Reference Interpreter (RI) that is provided with OS/400 V4R4, using the AS/400 QShell environment. An example of compiling an sqlj program, CstmrInq.sqlj, is shown in Figure 135 on page 195.

```
  $
> java sqlj.tools.Sqlj -status CstmrInq.sqlj
  [Translating 1 files.]
  [Reading file CstmrInq]
  [Translating file CstmrInq]
  [Compiling 1 Java files.]
  Message [t65] not found in file 'sqlj.mesg.TranslatorErrorsText'.
  $

===>



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

Figure 135. Compiling the CstmrInq.sqlj Program

Use the -status option when compiling, because it gives you messages on the steps of the compilation. You can ignore the message: `Message [665] not found in file 'sqlj,mesg.TranslatorErrorsText'`. It is due to a missing message in a file in the RI on the pre-release system that we used. This problem may have been fixed by the time you read this redbook.

Note: The current version of VisualAge for Java V2.0 does not provide direct support for using SQLJ. The SQLJ statements have this type of syntax:

```
#sql { select cfirst from CSTMR }
```

The symbol `#sql` is not recognized by VisualAge for Java V2.0 and will give you an error. If want to use VisualAge for Java, you can develop your graphical user interface using VisualAge for Java and generate the getter and setter methods. Then, export your application outside of VisualAge for Java and add any SQL statements that you need. This is the approach that we used in this redbook for this application.

A future release of VisualAge for Java may have support for SQLJ.

8.5.2 Running the Order Entry Server Application

You can run the Order Entry server application on an AS/400 system or a PC. This section explains both scenarios.

8.5.2.1 Running Order Entry on the AS/400 System

This application is based on the JDBC application, which is explained in detail in Chapter 7, “Moving the Server Application to Java” on page 133. It uses Remote Method Invocation (RMI) to invoke an AS/400 Java program from a workstation based GUI application.

To run this application, follow these steps on the AS/400 system:

1. Open an AS/400 5250 session, start the Qshell environment, and start the RMI Registry. Figure 136 on page 196 shows using the **rmiregistry** command to start the RMI Registry using TCP/IP port number 5005.

```
  $
> rmiregistry 5005









===>


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

Figure 136. Running the rmiregistry Command

2. Open another AS/400 5250 session and start the server Java application using the **java** command as shown in Figure 137. The TCP/IP port number used by the RMI Registry is input as a parameter.

```
                          Run Java Program (JAVA)

 Type choices, press Enter.

Class . . . . . .>  OrderEntrySQLJPkg.OrderEntrySQLJ
 Parameters . . . . . > 5005



              + for more values


 Classpath  . . . . . .   *ENVVAR




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

Figure 137. Starting the Server SQLJ Application Using the Java Command

If you want to see messages while the JVM is loading classes, page down to the next display and enter the *VERBOSE option as shown in Figure 138 on page 197.

```
                          Run Java Program (JAVA)

Type choices, press Enter.


                         Additional Parameters

Option . . . . . . . . . . . . . > *VERBOSE  *NONE, *VERBOSE, *DEBUG...
              + for more values

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

*Figure 138. Entering the *VERBOSE Option*

Once you press the **Enter** key, the loading of the Java classes starts. If you are running in verbose mode, you will see messages for all of the class loads. If the application starts correctly, you will see a message indicating that the program successfully registered with the RMI security manager. Figure 139 shows a successful program start.

```
                            Java Shell Display

  a400/jdk117/lib/classes.zip
  Loading class java/io/ObjectInputValidation.class from /QIBM/ProdData/Java400
  /jdk117/lib/classes.zip
  Loading class sun/rmi/transport/DGCClient.class from /QIBM/ProdData/Java400/j
  dk117/lib/classes.zip
  Loading class sun/rmi/transport/DGCClient$CleanRequest.class from /QIBM/ProdD
  ata/Java400/jdk117/lib/classes.zip
  Loading class sun/rmi/transport/DGCClient$CountTableEntry.class from /QIBM/Pr
  odData/Java400/jdk117/lib/classes.zip
  Loading class sun/rmi/transport/DGCClient$LeaseRenewer.class from /QIBM/ProdD
  ata/Java400/jdk117/lib/classes.zip
  Loading class sun/rmi/transport/DGCClient$LeaseTableEntry.class from /QIBM/Pr
  odData/Java400/jdk117/lib/classes.zip
  Main: Successfully registered with the security manager

===>



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

Figure 139. Starting the Host SQLJ Application

The message: `Main: Successfully registered with the security manager` must be displayed before for the client programs can establish a connection and invoke methods in this class using RMI.

8.5.2.2 Running Order Entry on a PC

To run the client application, enter the command as shown in Figure 140. Remember to include the Package name and the port number when starting the OrderEntryWdw2 class. This port number has to be the same as the one used by the RMI Registry Manager.

```
D:\>cd Javasamples

D:\Javasamples>cd sqlj

D:\Javasamples\sqlj>java SQLJGUI.OrderEntryWdw2 5005
```

Figure 140. Running the Client Application on a PC

Press **Enter**. The main Order Entry window appears as shown in Figure 141.

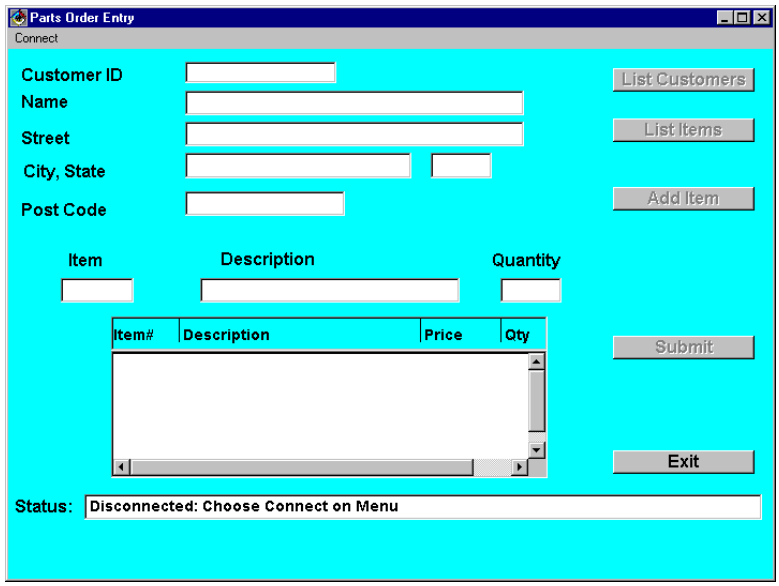


Figure 141. Main Order Entry Window

The rest of the steps are the same as explained in Chapter 6, “Migrating the User Interface to Java Client” on page 101. It is the same Order Entry application and it works in exactly the same way. The only difference is that, in this case, SQLJ is used to access AS/400 tables.

The changes to the source code that are required to implement SQLJ for the host Java program are explained in the following sections.

8.5.3 Converting the Host Program to Use SQLJ

In this section, we discuss the changes required to the host program to use SQLJ instead of JDBC for database access.

8.5.3.1 Additions to the OrderEntrySQLJ Class

Import the DefaultContext class as shown in Figure 142.

```
import sqlj.runtime.ref.DefaultContext;
```

Figure 142. Importing the DefaultContext Class

Call the initContext() method from the initialize method as shown in Figure 143.

```
initContext();
```

Figure 143. Calling the initContext() Method from the Initialize Method

In Figure 144, the initContext() method sets the DefaultContext.

```
static public DefaultContext initContext()
 {
       DefaultContext ctx = DefaultContext.getDefaultContext();
       if (ctx == null)
       {
         try
               {
               ctx = new DefaultContext(dbConnection);
               }
         catch (SQLException e)
               {
                   System.out.println("Error: could not get default context");
                   System.err.println(e);
                   System.exit(1);
                }
         DefaultContext.setDefaultContext(ctx);
       }
       return ctx;
 }
```

Figure 144. Setting the DefaultContext

In Figure 145 on page 200, we show the SQLJ statement we use to insert an order row in the Orders table.

```
// parm 1                   WAREHOUSE
// parm 2                   DISTRICT
// parm 3                   aCustomerNumber
// parm 4                   anOrderNumber
// parm 5                   anOrderLineCount
// parm 6                   'ZZ'
// parm 7                    1

String cDate = getRawDate(currentDateTime);     // parm 8
String cTime = getRawTime(currentDateTime);     // parm 9

#sql { INSERT INTO ORDERS (OWID, ODID, OCID, OID, OLINES, OCARID, OLOCAL, OENTDT,
       OENTTM)
       VALUES(:WAREHOUSE, :DISTRICT, :aCustomerNumber, :anOrderNumber,
       :anOrderLineCount,
              'ZZ', 1, :cDate, :cTime ) };
```

Figure 145. Inserting Rows into the ORDERS Table Using SQLJ

The following code snippets show the use of various SQLJ statements in the OrderEntrySQLJ class.

Figure 146 shows how we use SQLJ to insert order line rows into the ORDLIN table.

```
// parm 1                   anOrderNumber
// parm 2                   DISTRICT
// parm 3                   WAREHOUSE
// parm 5                   "JAVA"
// parm 9                   12311999
// parm 10                  235959

/* Variable and constant literals are allowed for SQLJ but expressions and
method calls are not allowed. Expressions and method calls needs to be
resolved before using it in SQLJ.
*/

BigDecimal blineCounter   = new BigDecimal(++lineCounter);          // parm4
String     bgetItemId     = orderLine.getItemId();                  // parm6
BigDecimal bgetItemQty    = orderLine.getItemQty();                 // parm7
BigDecimal orderAmount   = (orderLine.getItemPrice().
                            subtract(orderLine.getItemPrice().
                            multiply(customerDiscount).
                            divide(new BigDecimal(100),
java.math.BigDecimal.ROUND_DOWN))).
                            multiply(orderLine.getItemQty());    // parm8

#sql { INSERT INTO
       ORDLIN (OLOID, OLDID, OLWID, OLNBR, OLSPWH, OLIID, OLQTY, OLAMNT, OLDLVD,
       OLDLVT)
       VALUES(:anOrderNumber,:DISTRICT,:WAREHOUSE,:blineCounter,'JAVA',
       :bgetItemId
            ,:bgetItemQty,:orderAmount, 12311999, 235959) };

orderTotal = orderTotal.add(orderAmount);
```

Figure 146. Inserting Rows into the ORDLIN Table Using SQLJ

Figure 147 on page 201 shows using a SQLJ SELECT statement to retrieve the customer discount from the CSTMR table.

```
BigDecimal customerDiscount = null;
#sql { select CDCT into :customerDiscount from CSTMR where CID = :aCustomerID };
```

Figure 147. SQLJ SELECT from the CSTMR Table

Figure 148 shows a select and update combination. The row is selected from the DSTRCT table using a SELECT statement with a `for update` clause and the same row is updated using an UPDATE statement. The row is locked until we perform the update.

```
 BigDecimal orderNumber = null;

#sql { select DNXTOR into :orderNumber from DSTRCT where DID = :DISTRICT and
DWID = :WAREHOUSE for update };

BigDecimal aorderNumber = orderNumber.add(new BigDecimal(1));

#sql { update DSTRCT set DNXTOR = :aorderNumber where DID = :DISTRICT and
DWID =:WAREHOUSE };
```

Figure 148. Select and Update Combination

Figure 149 shows how we use the select statement with a `for update` clause to select the customer record so we can update it. This locks the row until we actually update it.

```
        String     uldate = null;
        BigDecimal ubal   = null;
        BigDecimal uytd   = null;
        String     ultime = null;

        #sql { select CLDATE, CBAL, CYTD, CLTIME into :uldate, :ubal, :uytd,
:ultime from CSTMR
               where CID = :aCustomerID for update };
```

Figure 149. SELECT from the CSTMR Table.

As shown in Figure 150, you have to change the date format type for an SQLJ implementation as compared to a JDBC implementation. For the JDBC date format, see Section 8.5.3.2, “Removal from the OrderEntrySQLJ Class” on page 202, to compare the difference in the date format.

```
        DateFormat dateFormatter = new SimpleDateFormat("MMddyyyy");
```

Figure 150. Changing the Date Format Type

Figure 151 on page 202 shows how we use the select statement with a for update clause to select the stock record so we can update it. This locks the row until we actually update it.

```
BigDecimal ustqty = null;
       #sql { select STQTY into :ustqty from STOCK where STIID = :aPartNbr
            and STWID = :WAREHOUSE for update };
```

Figure 151. SELECT from the STOCK Table

8.5.3.2 Removal from the OrderEntrySQLJ Class

We can remove the Statement class declarations from the OrderEntrySQLJ class.

The executeQuery() method is replaced by SQLJ statements for the ORDERS file. SQLJ does not use statement objects. All of the statement objects, shown in Figure 152, can be removed. Similarly, all of the other JDBC statements are replaced by the SQLJ statements shown in Section 8.5.3.1, “Additions to the OrderEntrySQLJ Class” on page 199.

```
private PreparedStatement psAddOrderLine = null;
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;
```

Figure 152. Removing the JDBC Statement Objects

Figure 153 shows the JDBC statement to insert rows into the ORDERS file. It is replaced by SQLJ statements shown in Figure 145 on page 200.

```
// 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(currentDateTime)+
                                      ", "+getRawTime(currentDateTime)+")";

      addOrderHeader.executeUpdate(sql);
```

Figure 153. Removing the JDBC Statement for the ORDERS Table

Figure 154 on page 203 shows how we insert rows into the ORDLIN table using JDBC. This code is replaced with the SQLJ statements shown in Figure 146 on page 200.

```
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),
                                          java.math.BigDecimal.ROUND_DOWN))).
                                          multiply(orderLine.getItemQty());

                psAddOrderLine.setBigDecimal(8, orderAmount);
                psAddOrderLine.setBigDecimal(9, new BigDecimal(12311999));
                psAddOrderLine.setBigDecimal(10, new BigDecimal(235959));

                // Add the order line record
                psAddOrderLine.executeUpdate();

                orderTotal.add(orderAmount);
```

Figure 154. Removing the JDBC Statements for the ORDLIN Table

The JDBC code to close() the statement objects shown in Figure 155 is removed.

```
                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;
```

Figure 155. Removing the JDBC Close Statements

All the other code in the original application what is JDBC related is removed. It is all replaced by SQLJ statements.

To summarize, these names were changed for the OrderEntrySQLJ application:

- Package name changed from `OrderEntryJDBCPkg` to `OrderEntrySQLJPkg`
- Class name changed from `OrderEntryJDBC` to `OrderEntrySQLJ`
- Constructor name changed from `OrderEntryJDBC()` to `OrderEntrySQLJ()`

- Class instance name changed from `OrderEntryJDBC oeJDBC = new OrderEntryJDLC();` to `OrderEntrySQLJ oeSQLJ = new OrderEntrySQLJ();`
- Binding name changed from `Naming.rebind("//"+SYSTEM+port+"/OrderEntryJDBC",oeJDBC);` to `Naming.rebind("//"+SYSTEM+port+"/OrderEntrySQLJ",oeSQLJ); oeSQLJ);`

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 use SQLJ.

Chapter 9. Java Native Interface

This chapter explains the integration of Java methods using the Java Native Interface (JNI) with C and RPG on the AS/400 system. We use these examples:

- Java primitives in RPG
- Java String object in RPG
- RPG Order Entry application (six RPG procedures)
- Hello C
- Change Java String Object from C
- C with Java Invocation API

In this chapter, you learn JNI through programming examples. Discussion of the general theory of JNI is kept to a minimum. Emphasis is placed in discussing the programming examples and how to set up your AS/400 system to use JNI.

This chapter is a starting point for using JNI with C and RPG on the AS/400 system. It is intended for Java programmers who want to use the Java Native Interface to call RPG or C programs. We assume that you have a basic understanding of Java and object oriented programming.

9.1 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 do 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.

9.1.1 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 uses:

- 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 JDK and this, as a standard interface, benefits programmers who want to load their native code libraries into different Java VMs. 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.

9.1.2 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++, RPG) is given the best option to run problem free in any standard implementation of a Java VM.

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 AS/400 system, several other alternatives to using the Java Native Interface are available:

- AS/400 Toolbox for Java access classes such as Distributed Program Call and Data Queue support.
- The PCML (Program Call Mark-up Language) support available with the AS/400 Toolbox for Java - modification 2.

The AS/400 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 AS/400 system, while the other methods start a new process (Job). This makes JNI calls faster at start up time and less resource intensive.

9.1.3 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 shown here:

- **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

9.2 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 AS/400 system at V4R4 or later. This is because RPG is not thread safe in releases prior to V4R4 and unpredictable results may occur.

To use JNI in an RPG program, you need to define the JNI functions and data types in your program. You also need to convert information from an AS/400 format to a Java format. To help with this, we describe several include files and support programs that we used to develop the examples in this redbook. The files that define the JNI functions and data types are called *include files in C*. In RPG, they are source members copied into an RPG source member with a /COPY directive. The files that we use are available as source members as part of the additional materials for this redbook.

9.2.1 Requirements for Using JNI and Java

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 *bound* type call rather than a *dynamic* one.
- The RPG application code to be called must be prototyped. It cannot use PARM or PLIST.
- The prototypes for the RPG code to be called using JNI must include two extra parameters:
  - The first extra parameter is a pointer to the "JNI information" structure. The main item in the structure is a list of procedure pointers to all the JNI functions for that particular instance of the JVM. JNIEnv@ is the base pointer for the structure in the include file JNI.
  - The second parameter is either a reference to the class (for a static function) or to the object (for an instance function).
- The EXTPROC name must adhere to a naming convention. For Java and RPG, the naming convention is "Java" + "_" + " + class name + " _" + method name.
- 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) option 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.

9.2.2 Setting Up JNI and RPG

This section shows you how to use the source members and explains the set up requirements, so you can use the JNI and RPG examples that are included in this redbook.

To set up to use JNI and RPG, you need to follow these steps:

1. Make sure that the support files from the redbook additional materials are available on your system. You need to have these five RPGLE source members available on your AS/400 system to use the JNI and RPG examples:
   - JNI
   - JNI_MD
   - CVTASCII
   - CVTASCIIH
   - CVTUCS (not required if PTF SF54594, which is available for Program Product 5769-RG1, is installed)

These RPG source members are available for download from the redbook Web site. They are found in the QRPGLESRC source physical file in the library named APILIB. Here are descriptions for each of the five source members:

- **JNI**

  This member defines the JNI data structures and prototypes. This is the RPG equivalent of the JNIEnv_ structure definition in the jni.h C header file. You need to include or copy this file in an RPG source member so it knows the format of all the JNI calls. Figure 156 shows the prototype for the GetStringUTFChar procedure, which we use in the example programs. It is a large file, so only part of the source code is shown.

```
D*------------------------------------------------------------------
D* Note: This function returns a  pointer to an array of "char".
D GetStringUTFChars...
D                 PR              *   EXTPROC(GetStringUTFChars@)
D env                                 LIKE(JNIEnv@) VALUE
D str                                 LIKE(jstring) VALUE
D isCopy                              LIKE(jboolean)

D*------------------------------------------------------------------
D*      void (*ReleaseStringUTFChars)
D*        (JNIEnv *env, jstring str, const char* chars);
D*------------------------------------------------------------------
D ReleaseStringUTFChars...
D                 PR                  EXTPROC(ReleaseStringUTFChars@)
D env                                 LIKE(JNIEnv@) VALUE
D str                                 LIKE(jstring) VALUE
D chars                           *   OPTIONS(*STRING) VALUE
```

Figure 156. Partial Source of the JNI Header File

- **JNI_MD**

  This is a copy member to include in the JNI member. This file contains the JNI data types. The JNI data types define the data formats that you can use when using JNI. This member contains the JNI types defined in RPG format. The data types are meant to be used with the LIKE keyword. For example, if you have a Java type of "float", you declare your RPG field or parameter as LIKE(jfloat). This member is copied into the JNI header file. Partial source code is shown in Figure 157 on page 210.

```
D                 DS                  BASED(JNIJavaTypePointer)
D jbyte                   1      1U 0
D jint                    1      4I 0
D jlong                   1      8I 0
D jlonghigh                     10I 0 OVERLAY(jlong:1)
D jlonglow                      10U 0 OVERLAY(jlong:5)
D jboolean                1      1U 0
D jchar                   1      2C
D jshort                  1      2I 0
D jfloat                  1      4F
D jdouble                 1      8F
D jsize                   1      4I 0
D va_list                 1     16*
D*-------------------------------------------------------------------
D javatype                1      4I 0
D jobject                           like(javatype)
D jclass                            like(javatype)
D jthrowable                        like(javatype)
D jstring                           like(javatype)
D jarray                            like(javatype)
D jbooleanArray                     like(javatype)
D jbyteArray                        like(javatype)
D jcharArray                        like(javatype)
D jintArray                         like(javatype)
D jshortArray                       like(javatype)
D jlongArray                        like(javatype)
D jfloatArray                       like(javatype)
D jdoubleArray                      like(javatype)
D jobjectArray                      like(javatype)
D jvalue                            like(javatype)
D jfieldID                          like(javatype)
D jmethodID                         like(javatype)
```

Figure 157. JNI_MD Source

- **CVTASCII**

  This is an RPG source member that is compiled into an RPG module, which is bound into a service program. This service program can then be used in an RPG application program to convert data from ASCII format to EBCDIC format. We use this module in a JNI example application to convert Java String data to EBCDIC format, so we can use it in the RPG program. In the RPG application, we use JNI calls to convert Java String data to ASCII. Then, we use this service program to convert the ASCII data to EBCDIC data. Part of the source code is shown in Figure 158 on page 211.

```
H NOMAIN THREAD(*SERIALIZE)
  /COPY CVTASCIIH
 D EBCDIC_CCSID    C                   37

 /*-------------------------------------------------------------
 /* Prototypes for local functions
 /*-------------------------------------------------------------
 D initConv        PR
 D destConv        PR
 D DestConv_P      C                   %PADDR('DESTCONV')
 D doConv          PR                  LIKE(convType)
 D   convCode                          LIKE(iconv_t)
 D   fromData                          LIKE(convType)
 D                                     CONST OPTIONS(*VARSIZE)
/* Procedure toAscii
/*-------------------------------------------------------------
 P toAscii         B                   EXPORT
 D toAscii         PI                  LIKE(convType)
 D EbcdicVal                           LIKE(convType)
 D                                     CONST OPTIONS(*VARSIZE)

 C                   RETURN    doConv(EtoA : EbcdicVal)

 P toAscii         E
```

Figure 158. CRTASCII Program

- **CVTASCIIH**

  This file is a copy member that is included in the CVTASCII module. It defines prototypes for the `toAscii` and `toEbcdic` procedures. The complete source code is shown in Figure 159.

```
 D convType        S            200A   VARYING
 D toAscii         PR                  LIKE(convType)
 D   Ebcdic                            LIKE(convType)
 D                                     CONST OPTIONS(*VARSIZE)
 D toEbcdic        PR                  LIKE(convType)
 D   Ascii                             LIKE(convType)
 D                                     CONST OPTIONS(*VARSIZE)
```

Figure 159. CVTASCIIH Source

- **CVTUCS**

  This is an RPG source file which is compiled into an RPG service program. This service program can then be used in an RPG application program to convert data from EBCDIC format to UCS (Universal Character Set) format. We use this routine in the RPG JNI example application to convert EBCDIC characters to Java format, so we can return them to the invoking Java program. The complete source code is shown in Figure 160 on page 212.

```
H THREAD(*SERIALIZE)
 D cvtucs          pr
 D  ucs                          200c   varying options(*varsize)
 D  alpha                        200a   varying options(*varsize) const
 D cvtucs          pi
 D  ucs                          200c   varying options(*varsize)
 D  alpha                        200a   varying options(*varsize) const
 C                   eval      ucs = %ucs2(alpha)
 C                   return
```

Figure 160. CVTUCS Source

CVTUCS Replacement

PTF SF54594, which is available for Program Product 5769-RG1, eliminates the requirement for the CVTUCS service program.

2. Compile two of the source files into modules:

   - CVTASCII
   - CVTUCS

   These programs are already available in a service program in APILIB. If you want to use them in another library, you need to copy them or compile them into modules and create a service program for them.

   Enter the Create Service Program (CRTSRVPGM) CL command to create a service program based on these two modules:

```
CRTSRVPGM  SRVPGM(Yourlib/JSRV) +
      MODULE(YourLib/CVTUCS + YourLib/CVTASCII) EXPORT(*ALL)
```

3. Add copy statements to your RPG source member.

   To write an RPG application program that uses JNI, you need to include the prototypes and data structure definitions for the JNI procedures. In the redbook examples, we use /COPY directives to do this. In our examples, we use the following statements:

```
/COPY CVTASCIIH
/COPY JNI
```

4. Compile your RPG program into a module.

   After writing your RPG program, compile it into a module.

   If embedded SQL has been used in the RPG module, you either need to set Commitment control to *NONE, or implement complete commitment control. In this example, we set commitment control to *NONE.

   Figure 161 on page 213 shows the Create SQL ILE RPG Object display, for the DISTRICTR program, where we set Commitment control to *NONE when compiling an RPG source code with embedded SQL.

```
                    Create SQL ILE RPG Object (CRTSQLRPGI)

Type choices, press Enter.

Object . . . . . > DISTRICTR     Name
  Library  . . . . . > APILIB       Name, *CURLIB
Source file  . . . . . > QRPGLESRC     Name, QRPGLESRC
  Library  . . . . . >   APILIB      Name, *LIBL, *CURLIB
Source member  . . . . . > DISTRICTR     Name, *OBJ
Commitment control . . . . . *NONE        *CHG, *ALL, *CS, *NONE...
Relational database  . . . . . *LOCAL
Compile type . . . . . > *MODULE       *PGM, *SRVPGM, *MODULE
Listing output . . . . . *Print       *NONE, *PRINT
Text 'description' . . . . . *SRCMBRTXT
```

*Figure 161. Setting Commitment Control to *NONE*

5. Create a service program of the RPG modules.

   To use RPG with Java, you need to create a service program of the RPG modules. We bind to the service program, JSRV, that we created in step two from this service program containing the application code. You need to change these defaults on the CRTSRVPGM command:

   - Export . . . . . .EXPORT > *ALL
   - Bind service program . . . . . BNDSRVPGM > **JSRV**
   - Activation group . . . . . .ACTGRP > QILE

   Figure 162 on page 214 shows the Create Service Program (CRTSRVPGM) display with these changes.

   Rather than changing the defaults for the CRTSRVPGM command, you can also put the service program in a binding directory and use the BNDDIR option to locate the binding directory. Refer to the *ILE RPG for AS/400 Programmer's Guide*, SC09-2507, for details.

```
                    Create Service Program (CRTSRVPGM)

Type choices, press Enter.

Service program  . . . . . SRVPGM      > DISTRICTR
    Library  . . . . . .                 >   APILIB
Module . . . . . . . . . . . MODULE      > DISTRICTR
    Library  . . . . . .                 >   APILIB
                         + for more values
                                           *LIBL
Export . . . . . . . . . . . EXPORT      > *ALL
Export source file . . . . . . . SRCFILE       QSRVSRC
  Library  . . . . . .                           *LIBL
Export source member . . . . . . SRCMBR        *SRVPGM
Text 'description' . . . . . . . TEXT          *BLANK
                    Create Service Program (CRTSRVPGM)

Type choices, press Enter.


                           Additional Parameters

Bind service program . . . . . . BNDSRVPGM   > JSRV
  Library  . . . . . .                           *LIBL
                         + for more values
                                                 *LIBL
Binding directory  . . . . . . . BNDDIR        *NONE
  Library  . . . . . .
                         + for more values

Activation group . . . . . . . . ACTGRP      > QILE
Creation options . . . . . . . . OPTION
                         + for more values
Listing detail . . . . . . . . . DETAIL        *NONE
Allow update . . . . . . . . . . ALWUPD        *YES

                                                        More...
```

Figure 162. CRTSRVPGM Command

This completes the prerequisites and steps for creating an RPG program that can be used with JNI.

6. Write your Java program.

   You can write your Java program anywhere. For simplicity, we recommend that you use the Qshell Interpreter and Notepad editor. To use the Qshell Interpreter on the AS/400 system, see Section 3.4.2, “Setting Up the Environment on the AS/400 System” on page 60.

   To compile your Java program, enter the **javac** command on the Qshell command line.

   To run your Java program, enter the **java** command, followed by the program name and any parameters. If you have problems running your Java program, take a closer look at your CLASSPATH setting. For details on how to set your CLASSPATH, see Section 3.4.1.2, “Setting Up the CLASSPATH Variable” on page 60.

9.2.3 Java Primitives in RPG 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 parameter named DistrictId. This parameter must contain a value between one and ten.
2. DistrictJ passes the DistrictId 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 DistrictId 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 DistrictId value 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.

9.2.3.1 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 a static method, as shown in Figure 163.

```
public class DistrictJ
{
   static native float dist(int DistrictId);
```

Figure 163. Using Non-Java Methods in Java

The Java program needs to know in which service program the procedure is located. Figure 164 shows the standard way of defining an RPG service program within Java. You must use a static declaration and the System.loadLibrary method to load this service program and link it to the Java program.

```
static
    {
    System.loadLibrary("DISTRICTR");
    }
```

Figure 164. Defining an RPG Service Program in Java

Figure 165 demonstrates the use of Java primitive types with RPG. The String object args[0] that is received as keyboard input is converted into an integer, which will be passed to the RPG method.

```
public static void main(String args[])
{
if(args.length > 0)
{
Integer DistrictIdI = Integer.valueOf(args[0]);
int     DistrictId  = DistrictIdI.intValue();
```

Figure 165. Java Primitive Types with RPG

The call of the dist method is shown in Figure 166. This code is part of the Java program, named DistrictJ.

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.

```
float YTDBalance;

YTDBalance = DistrictJ.dist(DistrictId);
```

Figure 166. Calling the RPG dist Procedure

The YTDBalance returned by the RPG program is displayed by the Java program using the code shown in Figure 167.

```
    System.out.println("For district " + DistrictId +
      " YTD Balance retrieved by RPG JNI program = " +
       YTDBalance);
    }
    else
    {
    System.out.println("Please enter DistrictId between 1 and 10");
    }
  }
}
```

Figure 167. Displaying the YTD Balance

This completes the Java program. Next, the corresponding RPG program is explained.

9.2.3.2 RPG Procedure: DistrictR

RPG code 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 code.

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)
```

As shown in Section 9.2.2, “Setting Up JNI and RPG” on page 208, you need to copy these two source members into the RPG source program to use JNI:

```
/COPY CVTASCIIH

/COPY JNI
```

Figure 168 shows the PR prototype definition. This definition ensures that the Java and RPG binding signature match.

```
D dist            PR                EXTPROC(
D                                   'Java_DistrictJ_dist')
D                                   LIKE(jfloat)
D JNIEnv                        *   VALUE
D object                            VALUE LIKE(jobject)
D DISTID                            VALUE LIKE(jint)
```

Figure 168. PR Prototype Definition

For example, in these lines of code, dist is the name of the RPG sub-procedure. The actual external name is Java_DistrictJ_dist (Java_class name_method name).

```
D dist            PR                EXTPROC(

D                                   'Java_DistrictJ_dist')
```

The RPG sub-procedure (dist) is linked with the Java program, called DistrictJ. The name of the native method in the Java program is also named dist.

This line shows the return value of type Java float:

```
D                                   LIKE(jfloat)
```

The Java float data type is the same as an RPG float, so no conversion is required.

This line of code is the definition to return the pointer value of the JNI environment:

```
D    JNIEnv                     *   VALUE
```

It is used to establish the link between RPG and Java.

The object definition makes the current Java program object in the JNI environment available, as shown here:

```
D    object                      VALUE LIKE(jobject)
```

In this line of code, the parameter is passed from Java to RPG. No conversion is required between Java int and RPG int.

```
D    DISTID                      VALUE LIKE(jint)
```

Figure 169 shows the PI prototype interface definition, which matches the PR definition that was shown in the beginning of this program.

```
P dist            B                   EXPORT
D dist            PI                  LIKE(jfloat)
D JNIEnv                          *   VALUE
D object                              VALUE LIKE(jobject)
D DISTID                              VALUE LIKE(jint)
```

Figure 169. PI Prototype Interface Definition

The following line of code marks the beginning of the sub-procedure and shows that the sub-procedure is to be exported into the service program. It is publicly available in the system for access by Java or any other language.

```
Pdist             B                   EXPORT
```

Figure 170 shows the C-specs, in which the JNI environment pointer that was received from the PI prototype interface is loaded into the JNIEnv@ RPG pointer variable.

Note: JNIEnv@ is not defined in this program because a pointer is implicitly defined in RPG.

```
D YTDBAL          S             13P 2

C                   EVAL      JNIEnv@ = JNIEnv
```

Figure 170. C Specifications

Figure 171 is a standard embedded SQL statement. The YTDBAL field 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 District (DSTRCT) file.

```
C/EXEC SQL
C+   select DYTD into :YTDBAL
C+          from DSTRCT
C+          where DID = :DISTID
C/END-EXEC

C                   RETURN    YTDBAL
Pdist             E
```

Figure 171. Standard Embedded SQL Statement

This completes the RPG program.

In this example, the RPG program receives an integer DISTID from a Java program and uses imbedded SQL to retrieve the record of 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.

9.2.4 Java String Object in RPG Example

This example demonstrates how to handle Java objects in RPG with the help of JNI. In this example, the following actions occur:

1. The Java program, named CstmrJ, accepts a customer number between one and ten.
2. It passes this number to an RPG sub-procedure as a String object.

   Note: String is an object type and is not directly supported in RPG.
3. JNI is used in the RPG program to convert the String object into ASCII characters, which are then converted into EBCDIC by a user sub-procedure.
4. The EBCDIC Customer Number is used to access the Customer Master File (CSTMR) using embedded SQL.
5. The results from the SQL execution are stored in an RPG data structure. This data structure is converted from EBCDIC to universal characters (UCS) by a user sub-procedure and then converted into a String object by a JNI function.
6. The String object is passed back to the Java program, which writes it to the display.

9.2.4.1 Java Class: CstmrJ

The Java program, shown in Figure 172 on page 220, calls the cust RPG procedure in the AS/400 service program, named CSTMRR. The only difference between this program and the previous example is that instead of passing a primitive data type, a String object is passed to the RPG program and a String object is returned from the RPG program.

A String object is handled very naturally in Java as part of the language. However, for RPG, an object data type not supported natively. JNI is used to handle Java objects in RPG. The RPG program (CSTMRR), shown in the next section, uses JNI statements to handle String objects. This Java program is similar to the previous example where JNI was explained in detail.

```
public class CstmrJ
{
    static native String cust(String s);
    static
    {
        System.loadLibrary("CSTMRR");
    }
    public static void main(String args[])
    {
        if (args.length > 0)
        {
        String customerDetails = null;
        customerDetails = CstmrJ.cust(args[0]);
        System.out.println(customerDetails);
        }
        else
        {
        System.out.println("Please Enter Customer No between 0001 and 0010 when
starting program ");
        }
    }
}
```

Figure 172. CstmrJ Java Program

9.2.4.2 RPG Procedure: CSTMRR

The complete RPG procedure is shown below in code snippets. However, the statements that we explained in the previous example will not be elaborated upon again. We only explain the new statements.

The CVTUCS prototype definition is shown in Figure 173. It converts RPG EBCDIC characters to Java UCS data type characters.

```
H nomain thread(*serialize)
 /COPY CVTASCIIH
 /COPY JNI

D cvtucs          pr
D  ucs                          200c   varying options(*varsize)
D  alpha                        200a   varying options(*varsize) const
```

Figure 173. CVTUCS Prototype Definition

The PR and PI definitions are shown in Figure 174 on page 221. Umsg is a text field to store character string for conversion between EBCDIC and Java Universal Character Set format. TEXTR is a data structure for collecting the SQL returned fields into the record type of a data structure.

```
D cust PR                 EXTPROC(
D                                  'Java_CstmrJ_cust')
D                                  LIKE(jstring)
D    JNIEnv                   *    VALUE
D    object                        VALUE LIKE(jobject)



P    cust B                        EXPORT
D                PI                LIKE(jstring)
D    JNIEnv                   *    VALUE
D    object                        VALUE LIKE(jobject)
D    CustId                        VALUE LIKE(jstring)
D    Umsg        S           200c  VARYING


D TEXTR           DS
D CLAST                       16A
D CFIRST                      16A
D CCITY                       20A
D CSTATE                       2A
D                              1A
D CPHONE                      16A
```

Figure 174. PR and PI Definitions

Figure 175 on page 221 shows how the conversion is performed from a Java String object to ASCII characters, and then to EBCDIC characters.

```
D CustId@         S                *
D CustIdA         S               4A   based(CustId@)
D CustIdE         S               4A
D CustIdB         S                    LIKE(jboolean)

C                   EVAL      JNIEnv@ = JNIEnv

C                   EVAL      CustId@ = GetStringUTFChars(JNIEnv@:
C                             CustId : CustIdB )
C                   EVAL      CustIdE = toEbcdic(CustIdA)
C                   CALLP     ReleaseStringUTFCChars (JNIEnv@:
C                             CustId : CustId@ )
```

Figure 175. Conversion from Java String to ASCII Characters to EBCDIC Characters

The statements from Figure 175 are explained here:

```
CustId@=GetStringUTFChars(JNIEnv@: CustId: CustIdB)
```

The call to JNI function GetStringUTFChars uses a PR definition that is made available in this program by the /COPY JNI statement. Refer to Figure 156 on page 209 to see the GetStringUTFChars prototype. This procedure converts the Java String into ASCII characters and returns the pointer to the start of this ASCII field in the return variable, which is CustId@. For this procedure to work, it needs these parameters as input:

- **JNIEnv@** — The pointer to the JNI environment.
- **CustId** — The Java String object that is defined in the prototype interface of this RPG program.

- **CustIdB** — A Java boolean variable in which the status is stored.
- **CustId@** — An AS/400 storage address pointer to store the return address of the ASCII field.
- **CustIdA** — The actual ASCII field based on the address.
- **CustIdE** — The EBCDIC equivalent of the ASCII field.

Note: The user defined toEbcdic function is used to convert from ASCII to EBCDIC. This RPG sub-procedure is made available in this program by including the QSRV service program at the time of creating of this service program using the CRTSRVPGM command, which is explained in the set up instructions.

Figure 176 shows the embedded SQL statement.

Note: This statement uses CustIdE in the where clause field. This is the EBCDIC version of the Java String CustId. If the Java String CustId or the ASCII CustIdA field is used, the statement results in an error because of a type mismatch.

```
C/Exec sql
C+   select  CFIRST, CLAST, CCITY, CSTATE, CPHONE  into
C+           :CFIRST, :CLAST, :CCITY, :CSTATE, :CPHONE
C+           from CSTMR
c+           where CID = :CustIdE
C/end-Exec
```

Figure 176. Standard Embedded SQL Statement

As shown in Figure 177, CVTUCS, whose prototype is defined at the beginning of this program, is used to convert the TEXTR data structure into Universal Character Set format. The JNI NewString procedure creates a new String object using the Universal Character Set field as its base. This Java String object is then returned to the Java program.

```
CALLP      CVTUCS(umsg : TEXTR)

RETURN    NewString(JNIEnv@
C                                    : %addr(Umsg) + 2
C                                    : %LEN(umsg))

P cust               E
```

Figure 177. CVTUCS Procedure

CVTUCS Replacement

PTF SF54594, which is available for Program Product 5769-RG1, eliminates the requirement for CVTUCS. If you have installed this PTF, change the code shown in Figure 177 on page 222 like this:

Replace `CALLP     CVTUCS(umsg : TEXTR)` with `EVAL      umsg = %UCS2(TEXTR)`

In summary, the Java program accepts a customer number as input and passes it as a Java String object to the RPG program. The RPG program converts this String object to EBCDIC and uses this field to retrieve a record from the Customer table using embedded SQL. The SQL results are consolidated in a TEXTR data structure and this data structure is converted to Universal Character Set format. The Universal Character Set string is used in the NewString JNI function to create a String object, which is returned to the Java program. This resulting Java String is then written to the display by the Java program.

9.3 RPG Order Entry Example

In this example, the AS/400 **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 Section 7.2, “Order Entry Using JDBC” on page 151. The Java server program, which runs on the AS/400 system, is invoked using Remote Method Invocation (RMI) from a PC Java program. The RMI invoked AS/400 server Java program uses JNI to invoke RPG sub-procedures. Figure 178 shows the new design.

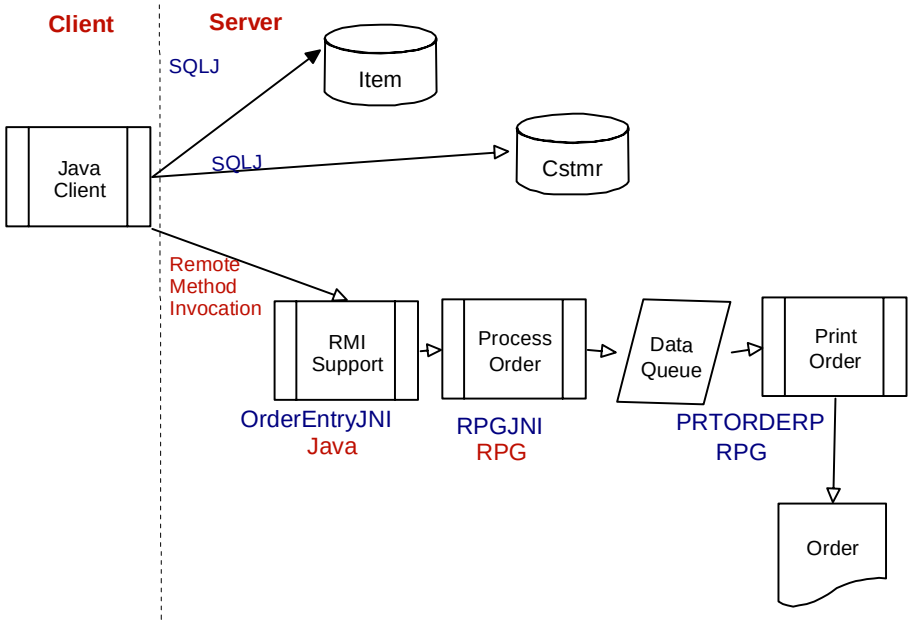


Figure 178. Order Entry RPG JNI Example

9.3.1 Java and RPG Programs

The RPG service program is called RPGJNI and it contains sub-procedures, which are invoked from the Java server program. These RPG sub-procedures are invoked:

- OrdNbr
- Discount
- OrdLin
- UpdStock
- addOrdHdr
- updCst

We explain the Java and RPG code together, in most of the cases, as program segments, so you can see the exact RPG code that is invoked from the Java method. The complete program is available, as part of the additional material for this redbook on the ITSO Web site. Here is a list of the programs and their names:

- **RPGJNI** — RPG service program
- **OrderEntryJNI** — Server Java program
- **OrderEntryWdw2** — Client Java program

9.3.1.1 Loading the RPG Service Program in Java

The Remote Method Interface (RMI) server AS/400 Java program (OrderEntryJNI) is invoked from the PC Java program (OrderEntryWdw2). In Figure 179, the RPGJNI server program is loaded by this Java program.

```
public class OrderEntryJNI extends UnicastRemoteObject implements OrderEntryI

   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 179. Loading the RPGJNI Server Program

9.3.1.2 Signature Matching between Java and RPG

In the following code snippets, the Java native method definition and the RPG prototype definition are shown together. This allows you to see the matching of signatures. Signatures include the name of program, the method name, and number and type of parameters passed between the Java native method and RPG sub-procedure.

- **OrdNbr()**

  Figure 180 shows the Java native method declaration of OrdNbr and the matching RPG sub-procedure Prototype OrdNbr.

```
static native float OrdNbr();


D OrdNbr          PR                EXTPROC(
D                                   'Java_OrderEntrySQLJPkg+
D                                   _OrderEntryJNI_OrdNbr')
D                                   LIKE(jfloat)
D    JNIEnv                       *  VALUE
D    classParm                       VALUE LIKE(jobject)
```

Figure 180. OrdNbr Signature Matching

- **Discount()**

  Figure 181 shows the Java native method declaration of Discount and the matching RPG sub-procedure Prototype Discount.

```
static native float Discount(String s);


D Discount        PR                EXTPROC(
D                                   'Java_OrderEntrySQLJPkg+
D                                   _OrderEntryJNI_Discount')
D                                   LIKE(jfloat)
D    JNIEnv                       *  VALUE
D    classParm                       VALUE LIKE(jobject)
D    customerId                      VALUE LIKE(jstring)
```

Figure 181. Discount Signature Matching

- **OrdLin()**

  Figure 182 on page 226 shows the Java native method declaration of OrdLin and the matching RPG sub-procedure Prototype OrdLin.

```
static native void  OrdLin(float ordernbr, int district, String warehouse, int
                           linecount, String ajava, String itemid, float itemqty,
                           float orderamt, float adate, float atime);


 D OrdLin          PR                EXTPROC(
 D                                   'Java_OrderEntrySQLJPkg+
 D                                   _OrderEntryJNI_OrdLin')
 D JNIEnv                        *   VALUE
 D classParm                         VALUE LIKE(jobject)
 D orderNbr                          VALUE LIKE(jfloat)
 D district                          VALUE LIKE(jint)
 D warehouse                         VALUE LIKE(jstring)
 D lineCount                         VALUE LIKE(jint)
 D ajava                             VALUE LIKE(jstring)
 D itemid                            VALUE LIKE(jstring)
 D itemqty                           VALUE LIKE(jfloat)
 D orderamt                          VALUE LIKE(jfloat)
 D adate                             VALUE LIKE(jfloat)
 D atime                             VALUE LIKE(jfloat)
```

Figure 182. OrdLin Signature Matching

- **UpdStock()**

  Figure 183 shows the Java native method declaration of UpdStock and the matching RPG sub-procedure Prototype UpdStock.

```
static native void  UpdStock(String aPartNbr, float aPartQty);


 D UpdStock        PR                EXTPROC(
 D                                   'Java_OrderEntrySQLJPkg+
 D                                   _OrderEntryJNI_UpdStock')
 D    JNIEnv                     *   VALUE
 D    classParm                      VALUE LIKE(jobject)
 D aPartNbr                          VALUE LIKE(jstring)
 D aPartQty                          VALUE LIKE(jfloat)
```

Figure 183. updStock Signature Matching

- **addOrdHdr**

  Figure 184 on page 227 shows the Java native method declaration of addOrdHdr and the matching RPG sub-procedure Prototype addOrdHdr.

```
static native void  addOrdHdr(String cstNbr, float ordNbr, float lineCount);
```

```
 D addOrdHdr        PR                  EXTPROC(
 D                                      'Java_OrderEntrySQLJPkg+
 D                                      _OrderEntryJNI_addOrdHdr')
 D    JNIEnv                       *    VALUE
 D    classParm                         VALUE LIKE(jobject)
 D cstNbr                               VALUE LIKE(jstring)
 D ordNbr                               VALUE LIKE(jfloat)
 D lineCount                            VALUE LIKE(jfloat)
```

Figure 184. addOrdHdr Signature Matching

- **updCst()**

  Figure 185 shows the Java native method declaration of updCst and the matching RPG sub-procedure Prototype updCst.

```
static native void  updCst(String cstNbr, float oTotal);
```

```
 D updCst           PR                  EXTPROC(
 D                                      'Java_OrderEntrySQLJPkg+
 D                                      _OrderEntryJNI_updCst')
 D    JNIEnv                       *    VALUE
 D    classParm                         VALUE LIKE(jobject)
 D cstNbr                               VALUE LIKE(jstring)
 D oTotal                               VALUE LIKE(jfloat)
```

Figure 185. updCst Signature Matching

9.3.1.3 Corresponding Java and RPG Code Segments

In the following code snippets, the Java native method invocation statements are shown along with the resulting code for the executable RPG sub-procedure. The details of the RPG statements were already explained in the earlier section.

In this section, we show the code examples. We only show partial Java code to highlight the relevant parameters used for passing values to the RPG procedures. The RPG procedures are the complete procedures.

- **OrdNbr()**

  The code segment in Figure 186 on page 228 shows the Java getOrderNumber method. It calls the OrdNbr RPG procedure, which returns the order number.

```
private BigDecimal getOrderNumber() throws Exception

               float oNumber = 0;

               oNumber = OrderEntryJNI.OrdNbr();
```

Figure 186. Calling the OrdNbr RPG Procedure

Figure 187 shows the complete code for the OrdNbr procedure.

```
P OrdNbr           B                   EXPORT
D                 PI                   LIKE(jfloat)
D    JNIEnv                        *   VALUE
D    classParm                         VALUE LIKE(jobject)

D DNXTORA          S              4F
D DNXTORB          S              4F

C                   EVAL      JNIEnv@ = JNIEnv
C                   EVAL      MYCLASS = classParm

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
C                   RETURN    DNXTORA
P OrdNbr           E
```

Figure 187. OrdNbr Procedure

- **Discount()**

  The code segment in Figure 188 shows how the Java getCustomerDiscount method calls the Discount RPG procedure.

```
private BigDecimal getCustomerDiscount (String aCustomerId) throws Exception

               float cDiscount;
               cDiscount = OrderEntryJNI.Discount(aCustomerId);
```

Figure 188. Calling the Discount RPG Procedure

Figure 189 on page 229 shows the complete code for the Discount procedure.

```
P Discount        B                   EXPORT
D                 PI                  LIKE(jfloat)
D    JNIEnv                       *   VALUE
D    classParm                        VALUE LIKE(jobject)
D    customerId                       VALUE LIKE(jstring)

D P@              S                    *
D CustASCII       S                4A   based(P@)
D CustEBCDIC      S                4A
D B               S                     LIKE(jboolean)

D cdiscount       S                4F

C                   EVAL      JNIEnv@ = JNIEnv
C                   EVAL      MYCLASS = classParm

C                   EVAL      P@ = GetStringUTFChars(JNIEnv@ : customer
C                                   : B)
C                   EVAL      CustEBCDIC = toEbcdic (CustASCII)

C/EXEC SQL
C+  select CDCT into :cdiscount
C+         from CSTMR
C+         where CID = :CustEBCDIC
C/end-exec

C                   return    cdiscount
PDiscount         E
```

Figure 189. Discount Procedure

- **OrdLin()**

  The code segment in Figure 190 on page 230 shows the parameter setting and how the OrdLin RPG procedure is called.

```
private BigDecimal addOrderLine(BigDecimal anOrderNumber, Order anOrder)
throws Exception

   BigDecimal orderTotal = new BigDecimal(0);
   int linecount = 0;
   BigDecimal customerDiscount = getCustomerDiscount(anOrder.getCustomerId());
   OrderDetail orderLine = anOrder.getFirstEntry();
   while(orderLine != null)
   {
       // parm 1
      float ordernbr    =     anOrderNumber.floatValue();
      // parm 2
      int district      =     1;
      // parm 3
      String warehouse  =     "0001";
      // parm 4
      linecount         =     linecount + 1;
      // parm 5
      String ajava      =     "JAVA";
      // parm 6
      String itemid     =     orderLine.getItemId();
      // parm 7
      float itemqty     =     (orderLine.getItemQty()).floatValue();
      // parm 8
      BigDecimal Amount =     OrderLine.getItemPrice().
                              subtract(orderLine.getItemPrice().
                              multiply(customerDiscount).divide(new
                              BigDecimal(100),
                              java.math.BigDecimal.ROUND_DOWN))).
                              multiply(orderLine.getItemQty());
      float orderamt    =     Amount.floatValue();
      // parm 9
      float adate       =     12311999;
      // parm 10
      float atime       =     235959;

      OrderEntryJNI.OrdLin(ordernbr, district, warehouse, linecount, ajava,
      itemid, itemqty, orderamt, adate, atime);
```

Figure 190. Calling the OrdLin RPG Procedure

Figure 191 on page 231 shows the complete code for the OrdLin procedure.

```
P OrdLin          B                   EXPORT
D OrdLin          PI
D    JNIEnv                       *   VALUE
D    classParm                        VALUE LIKE(jobject)
D orderNbr                            VALUE LIKE(jfloat)
D district                            VALUE LIKE(jint)
D warehouse                           VALUE LIKE(jstring)
D lineCount                           VALUE LIKE(jint)
D ajava                               VALUE LIKE(jstring)
D itemid                              VALUE LIKE(jstring)
D itemqty                             VALUE LIKE(jfloat)
D orderamt                            VALUE LIKE(jfloat)
D adate                               VALUE LIKE(jfloat)
D atime                               VALUE LIKE(jfloat)

D warehouse@      S                *
D warehouseA      S               4A  based(warehouse@)
D warehouseE      S               4A
D warehouseB      S                   LIKE(jboolean)

D ajava@          S                *
D ajavaA          S               4A  based(ajava@)
D ajavaE          S               4A
D ajavaB          S                   LIKE(jboolean)

D itemid@         S                *
D itemidA         S               6A  based(itemid@)
D itemidE         S               6A
D itemidB         S                   LIKE(jboolean)

C                   EVAL      JNIEnv@ = JNIEnv
C                   EVAL      MYCLASS = classParm

C                   EVAL      warehouse@ = GetStringUTFChars(JNIEnv@
C                             : warehouse : warehouseB)
C                   EVAL      warehouseE = toEbcdic (warehouseA)

C                   EVAL      ajava@ = GetStringUTFChars(JNIEnv@
C                             : ajava : ajavaB)
C                   EVAL      ajavaE = toEbcdic (ajavaA)

C                   EVAL      itemid@ = GetStringUTFChars(JNIEnv@
C                             : itemid : itemidB)
C                   EVAL      itemidE = toEbcdic (itemidA)

C/exec sql
C+  insert into ordlin
C+  (OLOID, OLDID, OLWID, OLNBR, OLSPWH,
C+   OLIID, OLQTY, OLAMNT, OLDLVD, OLDLVT)
C+  values(
C+  :ordernbr, :district, :warehouseE, :linecount, :ajavaE,
C+  :itemidE, :itemqty, :orderamt, :adate, :atime )
C/end-exec

C                   return
POrdLin           E
```

Figure 191. OrdLin Procedure

- **updStock()**

  The code segment in Figure 192 on page 232 shows the parameter setting and how the UpdStock RPG procedure is called.

```
orderTotal      =      orderTotal.add(Amount);

    OrderEntryJNI.UpdStock(orderLine.getItemId(), itemqty);
```

Figure 192. Calling the UpdStock RPG Procedure

Figure 193 shows the complete code for the UpdStock procedure.

```
P UpdStock         B                   EXPORT
D UpdStock         PI
D    JNIEnv                        *   VALUE
D    classParm                         VALUE LIKE(jobject)
D aPartNbr                             VALUE LIKE(jstring)
D aPartQty                             VALUE LIKE(jfloat)

D aPartNbr@        S                   *
D aPartNbrA        S                  6A  based(aPartNbr@)
D aPartNbrE        S                  6A
D aPartNbrB        S                      LIKE(jboolean)

D tqty             S                  4F

C                   EVAL      JNIEnv@ = JNIEnv
C                   EVAL      MYCLASS = classParm

C                   EVAL      aPartNbr@ = GetStringUTFChars(JNIEnv@
C                             : aPartNbr : aPartNbrB)
C                   EVAL      aPartNbrE = toEbcdic (aPartNbrA)

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 - aPartQty

C/Exec SQL
C+     update STOCK
C+            set       STQTY = :tqty
C+            where STIID     = :aPartNbrE
C+            and   STWID     = '0001'
C/end-exec

C                   return
PUpdStock         E
```

Figure 193. UpdStock Procedure

- **addOrdHdr**

  The code segment in Figure 194 on page 233 shows the parameter setting and how the addOrdHdr RPG procedure is called.

```
public String commitOrder (Order anOrder) throws RemoteException

     String cstNbr                   = anOrder.getCustomerId();
     BigDecimal orderLineCount       = new BigDecimal(anOrder.getNumEntries());

     BigDecimal orderNumber          = getOrderNumber();
     BigDecimal orderTotal           = addOrderLine(orderNumber, anOrder);

     float ordNbr                    = orderNumber.floatValue();
     float lineCount                 = orderLineCount.floatValue();

     OrderEntryJNI.addOrdHdr(cstNbr, ordNbr, lineCount);
```

Figure 194. Calling the addOrdHdr Procedure

Figure 195 shows the complete code for the addOrdHdr procedure.

```
PaddOrdHdr       B                   EXPORT
D addOrdHdr       PI
D    JNIEnv                      *   VALUE
D    classParm                       VALUE LIKE(jobject)
D cstNbr                             VALUE LIKE(jstring)
D ordNbr                             VALUE LIKE(jfloat)
D lineCount                          VALUE LIKE(jfloat)

D cstNbr@         S                   *
D cstNbrA         S                  4A  based(cstNbr@)
D cstNbrE         S                  4A
D cstNbrB         S                      LIKE(jboolean)

D                 DS
D timedate                          14  0
D time            S                  6  0
D date            S                  8  0

C                   EVAL      JNIEnv@ = JNIEnv
C                   EVAL      MYCLASS = classParm

C                   EVAL      cstNbr@ = GetStringUTFChars(JNIEnv@
C                             : cstNbr : cstNbrB)
C                   EVAL      cstNbrE = toEbcdic (cstNbrA)

C                   time                    timedate
C                   movel     timedate      time
C                   move      timedate      date

C/exec SQL
C+    insert into ORDERS
C+    (OWID, ODID, OCID, OID, OLINES,
C+    OCARID, OLOCAL, OENTDT, OENTTM)
C+    values
C+    ('0001', 1, :cstNbrE, :ordNbr, :lineCount,
C+    'ZZ', 1, :date, :time)
C/end-exec

C                   RETURN
PaddOrdHdr       E
```

Figure 195. addOrdHdr Procedure

- **updCst()**

  The code segment in Figure 196 shows the parameter setting and how the updCst RPG procedure is called.

```
float oTotal    = orderTotal.floatValue();

OrderEntryJNI.updCst(cstNbr, oTotal);
```

Figure 196. Calling the UpdCst RPG Procedure

Figure 197 on page 235 shows the complete code for the updCst procedure.

```
PupdCst           B                   EXPORT
D updCst           PI
D    JNIEnv                         *   VALUE
D    classParm                          VALUE LIKE(jobject)
D cstNbr                                VALUE LIKE(jstring)
D oTotal                                VALUE LIKE(jfloat)

D cstNbr@          S                    *
D cstNbrA          S                   4A  based(cstNbr@)
D cstNbrE          S                   4A
D cstNbrB          S                       LIKE(jboolean)

D                 DS
D timedate                            14  0
D time             S                   6  0
D date             S                   8  0
D bal              S                   4F
D ytd              S                   4F

C                   EVAL      JNIEnv@ = JNIEnv
C                   EVAL      MYCLASS = classParm

C                   EVAL      cstNbr@ = GetStringUTFChars(JNIEnv@
C                             : cstNbr : cstNbrB)
C                   EVAL      cstNbrE = toEbcdic (cstNbrA)

C                   time                    timedate
C                   movel     timedate      time
C                   move      timedate      date

C/exec sql
C+     select CBAL, CYTD
C+     into   :bal, :ytd
C+     from CSTMR
C+     where CID = :cstNbrE
C/end-exec

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
PupdCst           E
```

Figure 197. Updcst Procedure

9.4 Java Native Interface and C

This section discusses using the Java Native Interface and C on the AS/400 system.

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 so some of the primitive parameters have to be picked up from the stack.

9.4.1 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 get copied into a C program when using JNI. These source files are available on the AS/400 system in the files shown in Table 9.

Table 9. Source Members and Their Locations on the AS/400 System

Source Member	Location on the AS/400 System
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

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 198 on page 237.

```
                    Create Source Physical File (CRTSRCPF)

Type choices, press Enter.

File . . . . . .   H             Name
  Library  . . . . . .    APILIB      Name, *CURLIB
Record length  . . . . . .   92            Number
Member, if desired . . . . . .   *NONE         Name, *NONE, *FILE
Text 'description' . . . . . .   Header File for User JNI Headers
```

Figure 198. Create Source Physical File (CRTSRCPF) Display

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 AS/400 integrated file system directory.

3. Convert the header file from the integrated file system into an AS/400 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 199 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 199. Copy from Stream File (CPYFRMSTMF) Display

Figure 200 on page 238 shows an example of what the header file looks like 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 200. 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.

9.4.2 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.

9.4.2.1 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 the RPG section. 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 201 on page 239, 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 AS/400 Integrated Language Environment (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 201. HelloJ Java Class

9.4.2.2 Header File for Java Program: HelloJ

The header file is explained in Section 9.4.1, “Setting Up JNI and C” on page 236, which shows how to make your header file available on the AS/400 system.

9.4.2.3 C Program: HelloC

An example of a C Program, named HelloC is shown in Figure 202.

```
#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 202. 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 Section 9.4.1, “Setting Up JNI and C” on page 236. As explained in the set up 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 202 shows the minimum number of signature parameters. There must be at least two parameters: the JNI environment pointer and the Java class object.

9.4.3 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.

9.4.3.1 Java Class: ChangeStgJ

Figure 203 shows the Java program, which loads the AS/400 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 203. 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 executes 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.

9.4.3.2 Header File for Java Program: ChangeStgC

Figure 204 shows the header file for the ChangeStgC Java program. You must create this file for the Java class and 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 Section 9.4.1, “Setting Up JNI and C” on page 236.

```
/* 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 204. Header File for the ChangeStgC Java Program

9.4.3.3 C Program: JNICALL

The complete C program is shown in segments. Only new concepts and statements that are introduced in this example are explained.

Figure 205 on page 242 shows the beginning code of the C program. See the details for each line of code following the figure.

```
#include "nativeh.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 205. 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 AS/400 system. 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 AS/400 display.

The code segment in Figure 205 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 206 on page 243 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 206. 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 and explains how Java objects can be retrieved and manipulated in C. Of course, both RPG and C++ can also do this using JNI functions.

9.4.4 C with 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:

- Write a Java program.
- Create a header file.
- Create a C or RPG module and then make a service program out of it. We simply write a C program.

- Use a service program. Instead, we use a normal executable C program.
- Start a Java program. Instead, we start a C executable program, so no JVM is available. In all of our previous examples, the JVM was already available.

In this example, we demonstrate:

- Using a C program to create a Java Virtual Machine (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.

The code is very powerful and 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.

9.4.4.1 Special Considerations

When working with an executable C program, you need to consider *compilation* and *execution*. The compilation consideration of this program is different than the other examples. An executable C program needs to be created. When this program is created, you must specify a special service program that is to be bound with your program. The bind service program is named QJVAJNI.

Figure 207 on page 245 shows the Create Program (CRTPGM) command that is used to create an executable C program. QJVAJNI is specified as the Bind service program.

```
                          Create Program (CRTPGM)

Type choices, press Enter.

Bind service program . . . . . .   QJVAJNI     Name, generic*, *NONE, *ALL
  Library  . . . . . . . . . . .                Name, *LIBL
             + for more values

Binding directory  . . . . . . .   *NONE         Name, *NONE
  Library  . . . . . . . . . . .                Name, *LIBL, *CURLIB...
             + for more values

Activation group . . . . . . . .   *NEW          Name, *NEW, *CALLER
Creation options . . . . . . . .                *GEN, *NOGEN, *NODUPPROC...
             + for more values
Listing detail . . . . . . . . .   *NONE         *NONE, *BASIC, *EXTENDED...
Allow update . . . . . . . . . .   *YES          *YES, *NO
Allow *SRVPGM library update . .   *NO           *YES, *NO
User profile . . . . . . . . . .   *USER         *USER, *OWNER
Replace program  . . . . . . . .   *YES          *YES, *NO
                                                                      More...
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 207. Specifying QJVAJNI on the Create Program (CRTPGM) Command

Any job that creates a JVM must be multi-thread capable. The only jobs on the AS/400 system 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)
```

9.4.5 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 208 on page 246.

```
#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;
JNIEnv*        myEnv;
char*          myClasspath;
jclass         myClass;
jmethodID      mainID;
jclass         stringClass;
jobjectArray   args;

initArgs.version = 0x00010001;
```

Figure 208. INVOKEJAVA C Program

In Figure 208, the declaration of the main procedure should look like this:

```
int main (int argc, char *argv[])
```

Brackets ([]) are not available with the native AS/400 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 209 on page 247.

```
JNI_GetDefaultJavaVMInitArgs(&initArgs);

#pragma convert(819)

myClasspath = malloc( strlen(initArgs.classpath) +
             strlen(":/home/a999501") + 1 );
strcpy ( myClasspath, initArgs.classpath );
strcat ( myClasspath, ":/home/a999501a/omar");
initArgs.classpath = myClasspath;

JNI_CreateJavaVM(&myJVM, &myEnv, &initArgs);
```

Figure 209. Setting initArgs

As shown in Figure 209, 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(":/home/a999501a") + 1;
```

This variable is defined earlier. The length of this 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, ":/home/a999501a");
```

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 210, this statement locates the Java program, which 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 210. 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) -> NewObjectArrary(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.

9.4.5.1 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 211.

```
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 211. The javap -s Output

It shows us the signature for the main method. It expects a String array parameter.

Chapter 10. Applying Object Oriented Technology to the Application

In this chapter, we try to show you a more object-oriented approach in designing and coding the Order Entry application as discussed in the previous chapters. Our main intention is to provide you with the basic ideas behind object-oriented programming and show you how a proper design can help you in developing applications that are easy to use, extend, and most important, easy to maintain.

To achieve this goal, it is important to separate the different layers (that is, the view layer, business logic layer, and persistency layer) properly. These layers should talk to each other only through well-defined interfaces. To lead you through the different steps, we use the same application scenario as described in the earlier chapters.

This chapter covers a more object-oriented implementation of our VisualAge for Java application. In contrast to what you normally see in a VisualAge for Java demonstration, we use a different approach during the design and implementation phase. Normally, you place data access parts (for example, a keyed file part) directly onto the free form surface of your visual part, then connect the appropriate attributes to your window part. This technique is useful when you work on small projects. In a real-life object-oriented application design, you may consider not using the parts directly. You may want your application to be independent from the way you access the server resources. In this case, you can create several classes that all use a different technique to access the server system, but they all share a common interface for the domain objects to access them. Changing the access method only implies swapping these parts in your application.

In this chapter, we discuss the new implementation of the application that was described in the previous chapters. In the new design, the implementation of the domain objects and the domain logic is independent of the access technique. To access the data that resides on the AS/400 system, we reuse the same functionally as in the previous examples (that is, distributed program call, stored procedures, JDBC, and data queues). Since we want to show you that it is possible to move to a more object-oriented implementation without throwing everything away, we left some of the code already developed "as is" and we simply "wrap" this code with another class. A good example for this "wrapping approach" is the OrderEntryJdbcPersistencyManager class.

In our new design, we separate the application into three distinct layers:

1. **Views layer:** Represents the end-user interface
2. **Business objects layer:** Actual business logic that we implement to satisfy the application requirement. We also refer to this layer as the *Domain layer*.
3. **Database access layer:** Represents how we actually access the data that is stored in the AS/400 databases.

Our goal is to make each layer independent of the layer to which it interfaces. For example, we may decide to change the end user interface for our application, or change the way we access the Customer database from a JDBC interface to a JDBC Stored Procedure interface. We want to make these changes without having to change anything in the layers to which they interface.

Before we dig into the programming details, let us first look at the "big picture" of object-oriented programming. See Figure 212.

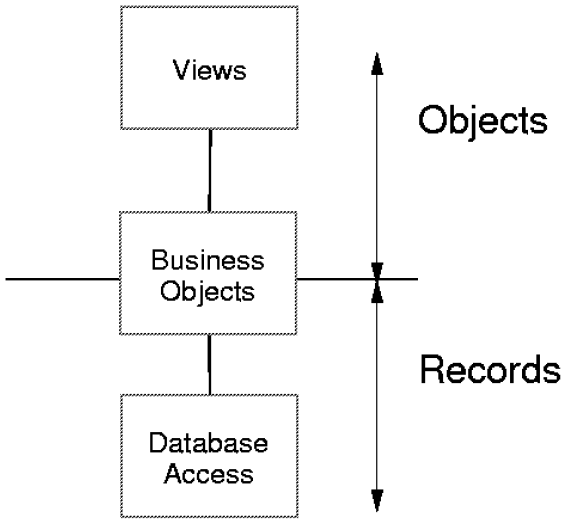


Figure 212. Object-Oriented Design Approach

10.1 Object Analysis

The technique used for the problem domain analysis is based on use cases and CRC cards (**C**lass, **R**esponsibility, **C**ollaborators). Keep in mind that the development process is an iterative process and that, at all stages, decisions have to be made. These can be changed later if they seem to be inappropriate.

To discover potential problem domain objects, start reading the scenario (use case) described in Chapter 5, “Overview of the Order Entry Application” on page 87. List all of the nouns that seem to relate to the application. This results in the following list:

- Customer
- District
- Order
- Service representative
- Customer number
- Item number
- Item
- Quantity
- Stock

By taking a closer look at this list of potential objects, you can see that:

- Service representative does not belong to the scope of the application. It is the person who works with the application.
- Customer number, item number, and quantity are properties of one of the other potential objects.

When we remove these candidates, we end up with this list of objects:

- Customer
- District
- Order
- Item
- Stock

Common sense tells us that this list is too limited. An order always contains multiple order lines and the company has a complete catalog of items to sell. Order line and catalog are added to the list of objects, which gives us this list of objects:

- Customer
- District
- Order
- Order Detail
- Item
- Stock
- Catalog

As mentioned in the previous section, the company has a complete catalog of items to sell. All items are listed in the catalog, and the catalog knows all of the items. If you want more information about a specific item, the catalog provides this information. In other words, through a catalog, you can manage the item-related information.

The relationship between an item and the catalog is a relation of the type *Manager - Managed Object*. This kind of relationship is common in commercial application environments.

Many types of objects exist in a real world environment and obey specific environmental rules. For example, a student studies in a school. In this case, you can see the school as the manager and the student as the managed object.

The first important role of the manager is that it knows the managed objects. If you want to know which students are enrolled in a specific class, you have to ask the school for this information. The same is true when you want to know which foreign students have applied for the next year. We can conclude that the manager knows its managed objects and that the manager has the responsibility to return information about these objects when they are asked for it.

Another important role of the manager is to know the rules that apply to the managed objects for them to belong to a certain environment and to enforce these rules. For example, a school does not accept students younger than five years old. This rule is known by the school and the school knows that it has to check the ages of the students before they are admitted.

We can take an even closer look at the manager. Suppose that we have to write an application that prints letters for the students that are enrolled in a specific class. In an object-oriented environment, the application asks the school first for all of the students enrolled for this specific class. The application then asks the students for their mailing address, and finally it prepares the letter based on this address.

As we already pointed out, the school (manager) knows the students (managed object) and returns them on request. There are two possible scenarios to handle such a request:

- The requested object already exists.
- The requested object has to be created.

For performance reasons, we normally create the objects only when they are actually needed. If the number of managed objects is small, you may consider keeping these objects permanently. In this case, the manager can reach the objects immediately. In case of a large amount of managed objects, you may want to choose the second possibility and create the managed objects only when the manager needs them.

For example, when the school has 1000 students enrolled and another 1800 students have applied for the next year, you choose the second possibility. It is impossible to manage all of these instances permanently in our application. When another object asks for the foreign students that have applied for the next year, the school creates these instances at that moment.

In both cases (managed objects always exist or are created on request), the manager must be able to retrieve the information necessary to build the managed objects. In most cases, this information comes from a database. That is, the manager is responsible for accessing the database, which is the third important role of the manager.

In this example, we have taken a high-level simplistic view of the role of a manager. In reality, if we implement a school application, one manager cannot possibly manage all of the objects that are required for the school. It is too complex, so we end up with several managers. For example, we may have a student manager, a curriculum manger, a schedule manager, and so on.

Summarizing these considerations, the manager has to play the following roles:

- Create, delete, return, and accept managed objects (in other words, manage the managed objects)
- Enforce environmental rules
- Access the database

Figure 213 illustrates the different roles of the manager.

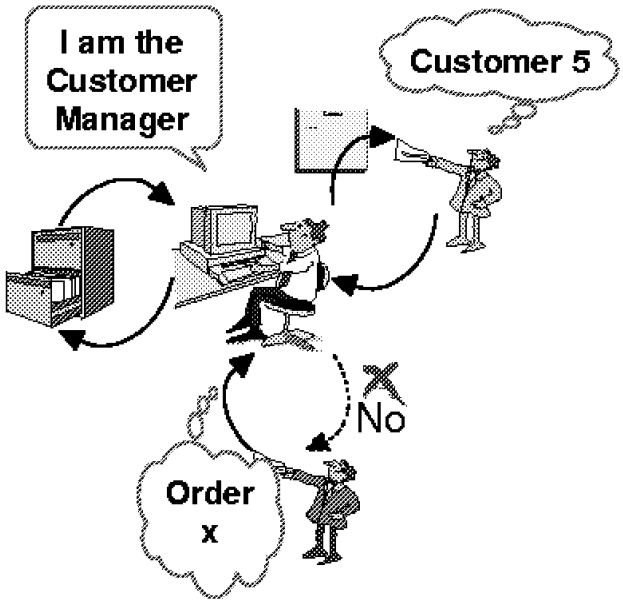


Figure 213. Roles of the Object Manager

Try to apply this design pattern to keep a consistent design. On the other hand, also keep in mind that this is not a mandatory rule, and that you can choose to do things differently if it is more appropriate for your design.

10.2 Object Model Design

The previous section helps to elaborate on and identify a more complete list of objects than the list we came up with in Section 10.1, “Object Analysis” on page 252. These objects communicate with each other and the rules of this communication are fixed in the object model.

This results in adding objects to the list of objects for two managers, one for the customers and one for the orders. We call these objects the Customer Portfolio and the Order Portfolio. This gives the final list of domain specific objects:

- Customer Portfolio
- Customer
- District
- Order Portfolio
- Order
- Order Detail
- Item
- Stock
- Catalog

Class Naming

The naming of classes and methods is important in object-oriented programming. The objects in your design usually represent real-life objects. To make the design easier to understand, it is good practice to give your application objects the same name as they have in real life. This is why Catalog is used instead of Item Manager.

By taking a closer look at this list, you can see that Order is a type of manager, because one Order manages various Order Details. You can say that Stock is also a Manager (more precisely, a Manager of Items), but these objects are already managed by the Catalog.

The static object model looks similar to the object model in Figure 214.

Note: In our application, the classes *Stock* and *District* are not implemented.

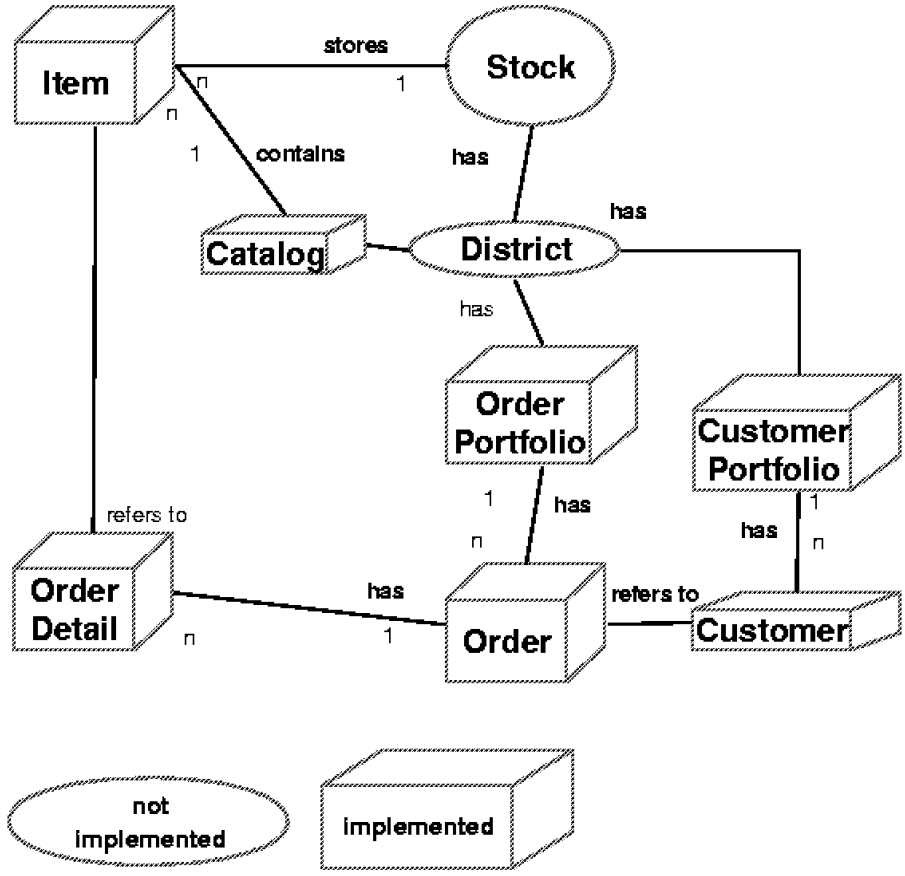


Figure 214. Object Model

10.2.1 Composition of Objects

As mentioned in the introduction of this chapter, the design should be independent of the access method that you use. That is, you have to find a general and easy-to-configure mechanism to deal with different access techniques.

The technique that we use to solve this problem is one of the most powerful object-oriented techniques, called *Composition*. This technique allows you to build complex objects using more simple components. The task that has to be performed by the complex object is divided into smaller subtasks, and each component performs its subtasks.

In our design, the responsibility of the Object Manager is to access the system resources through another object, called a *Persistency Manager*. A Persistency Manager is created for every access method. Composition allows changing easily from one access technique to another by replacing only the Persistency Manager. To enable the composition, you just have to provide a protocol that all Persistency Managers have to obey.

To use composition, our design has to apply these rules:

1. The Object Manager talks to the Persistency Manager in a precise and well-defined language. For our example, we call it the *Persistency Manager protocol*. This protocol is defined by the interface, known as the *PersistencyManager*. The PersistencyManager defines the methods that each of the concrete Persistency Manager classes must implement. Since PersistencyManager is an interface, there is no code to be performed, only method definitions. All of the concrete Persistency Managers have to implement the interface PersistencyManager. That is, they have to implement all of the methods defined by this interface. Their role is to fill the methods with the code that is necessary to access the corresponding resource and to return the requested information. A small example illustrates these concepts. The interface PersistencyManager defines a method getAll(). No code is written for this method. The StoredProcedureCustomerPortfolioPersistencyManager implements the getAll() method and defines the code that is necessary to access the AS/400 system using a stored procedure call (that is, create the stored procedure statement, open it, and retrieve all the data from the AS/400 system).

   Note: A Java interface is similar to an abstract class. It allows us to define methods without actually implementing them. We can implement the methods that are defined in the interface in another class. This class provides the actual code for the defined methods. Since Java allows a class to have only one super class, we do not have the concept of multiple inheritance. Implementing interfaces allows us to work around this restriction.

2. The Object Manager has to use methods that are defined in the PersistencyManager interface when it sends messages to one of the Persistency Managers. By enforcing this rule, you can switch from one Persistency Manager to another without much effort. This way of programming is also known as programming at an abstract interface level.

The information exchanged between the Domain (or Business) Object Manager and the Persistency Managers are records and keys. The interface between these classes is on an array level. This is illustrated in Figure 215.

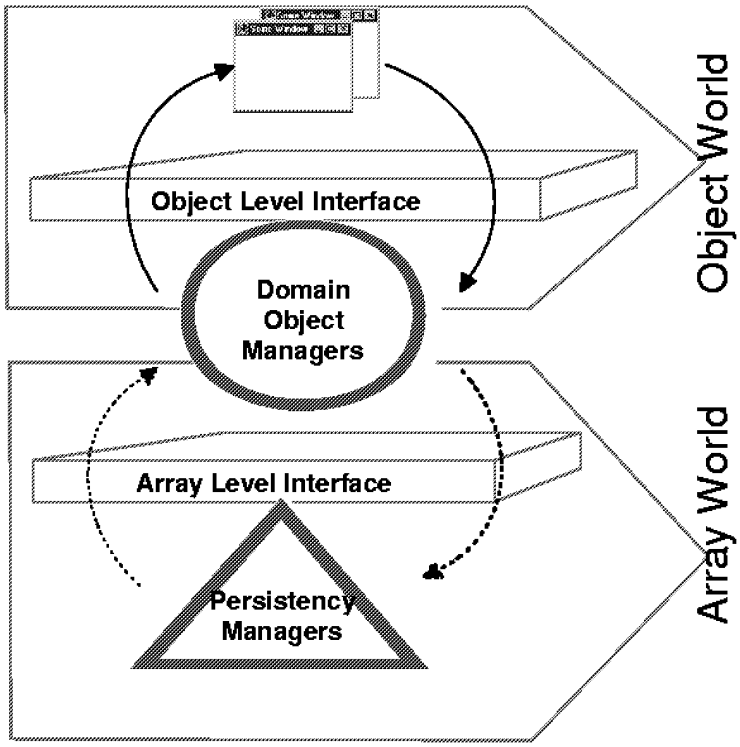


Figure 215. Interfaces

The record management is hidden in the Domain Object Manager. If an application object (for example, a window) asks the Domain Object Manager for a certain object, it receives the object and not the array. The interface between application objects and the Domain Object Manager is, therefore, on an object level.

After this rather theoretical discussion, let us take a closer look at the implementation details.

10.3 Implementation

This section focuses on the implementation of the different layers. As described in the previous sections, our application design consists of the following layers and classes. Only the major classes are listed. Some of them have also generated BeanInfo classes that are not listed in Table 10.

Table 10. Layers and Classes for Implementation

Layer	Classes	Description
Views	OrderEntryWdw	Same "main GUI class" as in the other packages with minor changes.
	SltCustWdw	Same class as in the other packages with minor changes.
	SltItemWdw	Same class as in the other packages with minor changes.
Domain Objects	Catalog	Responsible for keeping track of all Items. Has also a specific PersistencyManager associated.
	Customer	Object oriented representation of the customer file on the AS/400 system.
	CustomerPortfolio	Responsible for keeping track of all Customers. Has also a specific PersistencyManager associated.
	Item	Object oriented representation of the item file on the AS/400 system.
	Order	Responsible for keeping track of all OrderDetails. Has also a specific PersistencyManager associated.
	OrderDetail	Object oriented representation of the order detail file on the AS/400 system.
	OrderPortfolio	Responsible for keeping track of all Orders. Has also a specific PersistencyManager associated.
Persistency Managers	JdbcCustomerPortfolio PersistencyManager	This class is used to access the AS/400 database using JDBC. It returns a Vector of String[].

The relationship between the different classes and interfaces is shown in Figure 216.

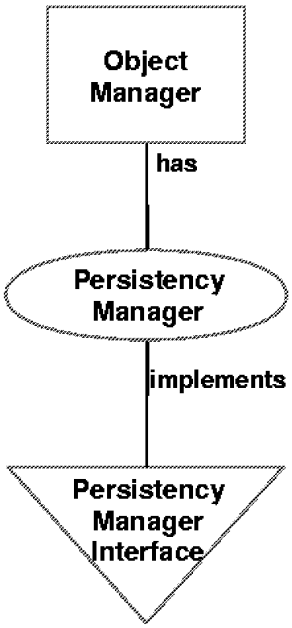


Figure 216. Relationships between Classes and Interfaces — Object Manager

This relationship works fine for all ObjectManagers, except for the OrderPortfolio, since this class has to provide an extended interface. In addition to the getAll() and getAt() methods that all PersistencyManagers have in common, OrderPortfolio has to submit the order after completing the order entry process. For that reason, we have decided to introduce a new interface that defines the submitOrder() method. Now, the relationship for the OrderPortfolio looks slightly different as illustrated in Figure 217.

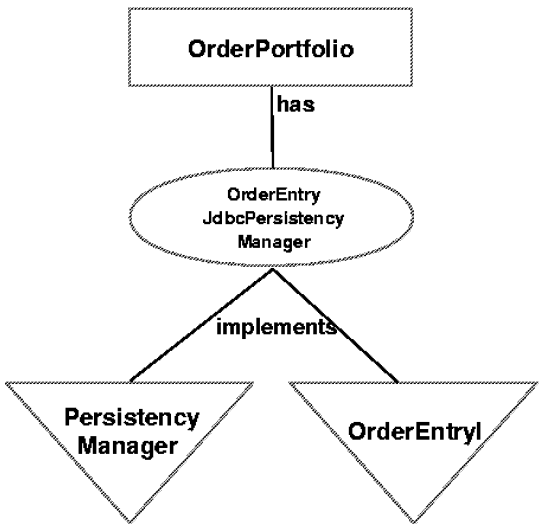


Figure 217. Relationships between Classes and Interfaces — OrderPortfolio

In the next sections, we describe the three different layers in more detail, and show you the actual implementation using code fragments. First, we cover the

interface to the domain objects since this interface is the most important one. Then, we cover the implementation of the Persistency Manager classes and interfaces. Finally, we show you how to use the classes.

10.3.1 Domain Object Interface

First, we review the old implementation. As an example, we use the DDMGUI.SltCustWdw, or more precisely, the methods populateCustBox() and custSelected(), as shown here:

```
 /**
* This method was created by a SmartGuide.
*
* This method invokes a stored procedure on
* the host AS/400 that returns a set of
* customer records. It then fills the customer
* list box with this information.
*
*/
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
 String[] array = new String[9];

 ResultSet rs = null;
 CallableStatement callableStatement = null;

 try
 {
 // invoke the stored procedure on the AS/400
 callableStatement = dbConnect.prepareCall
 ("CALL APILIB.SLTCUSTR(' ','R')");
 rs = callableStatement.executeQuery();

 while(rs.next())
 {
 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");

 getCustMLB().addRow(array);
 }
 }
 catch(SQLException e)
 {
 orderWindow.updateStatus("Error retrieving customer list");
 handleException(e);
 return;
```

```
 }

 orderWindow.updateStatus("Customer list retrieved");
 return;
}


/**
* This method was created by a SmartGuide.
*
* This method constructs a Customer object
* based upon the select row in the customer
* list box.
*/
private void custSelected() {

 Object[] selectedRow = getCustMLB().getSelectedRow();

 // declare an array of strings the same size as the
 // number of columns in the list box
 String[] custInf = new String[selectedRow.length];

 // retrieve the data from the selected row. It is
 // returned as an array of Object and each element
 // will be converted to a String object.
 // NOTE: the leading zeros in the CID field are
 // trimmed by the Object.toString() method!

 for(int i=0;iselectedRow.length;i++)
 {
 custInf[i] = selectedRow[i].toString();
 }

 // instantiate a Customer object, passing the
 // constructor the array of String data
 Customer custSelected = new Customer(custInf);

 // invoke the method that will set the text fields
 // in the OrderEntryWdw2
 orderWindow.setSelectedCust(custSelected);

 // close down
 this.dispose();

 return;
}
```

Keep in mind what we want to achieve with our new design.

Note: We want to keep the window and domain object classes *independent from* the access technique used.

When you look at the code, you can see that with the actual implementation. This is not the case. From within a method that belongs to a view class, we access the server system using JDBC stored procedures. For that reason, every time we want to use a new technique to access AS/400 system resources, we also need to also change the window code.

We need a way to access data through a consistent, access method neutral interface. For example, we want to ask the CustomerPortfolio these questions: Can you give me the list of all your customers? or Can I get the information for Customer 5? The manager, in turn, should access the data, retrieve it, and send it back to us in a convenient format. As soon as we get the data, all we have to do is to convert it to a format suitable for our MultiColumnListBox (that is, as a String[]) or any other format that we want to use to visualize the data.

How about a method, such as getCustomers(), implemented by the class CustomerPortfolio to get a list with all customers? Look at the methods that our CustomerPortfolio should understand as shown in Table 11.

Table 11. CustomerPortfolio Methods

Method	Description
getAt()	This method returns a Customer instance based on a key. The key is passed in as a String.
getCustomers()	This method returns a Vector of Customer instances.
setPersistencyManager()	This method accepts an instance of a PersistencyManager as the parameter. This Persistency Manager is responsible for accessing the data in a specific way.

Now, we examine the code that is written to retrieve a customer based on a key.

As you can see, this code example is independent of the actual access method used. It simply delegates its responsibility to access the data to the corresponding Persistency Manager. This method converts the data received from the Persistency Manager (that is, from the "array world") to the "object world" (see Figure 215 on page 258). The conversion is done in the constructor method of the Customer class.

```
/**
* Perform the getAt method.
* @param aKey java.lang.String
*/
public Customer getAt(String aKey) {
 /* Perform the getAtmethod. */

 String[] customerArray = new String[9];
 String[] key = new String[1];
 key[0] = aKey;
 java.util.Vector customerVector = getPersistencyManager().getAt(key);

 for (java.util.Enumeration customerElement = customerVector.elements();
  customerElement.hasMoreElements() ;)
 {
 customerArray = (String[])customerElement.nextElement();
 Customer aCustomer = new Customer(customerArray);
 return aCustomer;
 };

 return null;
}
```

The code needed to retrieve a list of all customers is similar. Here is the implementation of the getCustomers() method:

```
/**
* Gets the customersproperty (java.util.Vector) value.
* @return The customers property value.
* @see #setCustomers
*/
public java.util.Vector getCustomers() {
 /* Returns the customersproperty value. */
 if (fieldCustomers == null) {
 try {
 fieldCustomers = new java.util.Vector();
 getAll();
 } catch (Throwable exception) {
 System.err.println("Exception creating customersproperty.");
 }
 };
 return fieldCustomers;
}
```

We use lazy initialization to initialize the fieldCustomers variable. The actual processing is done in the getAll() method (see the following code snippet). Again, only the Persistency Manager is responsible for accessing the data. We simply delegate the request. The Persistency Manager returns a Vector, where each element is a String[]. With each element in the Vector, we can create a new instance of Customer, add it to the Vector tempCust, and set the property customers accordingly.

```
/**
* Perform the getAll method.
*/
public void getAll() {
 /* Perform the getAllmethod. */

 java.util.Vector tempCust = new java.util.Vector();
 String[] customerArray = new String[9];

 java.util.Vector allCustomers = getPersistencyManager().getAll();
 for (java.util.Enumeration customerElement = allCustomers.elements();
  customerElement.hasMoreElements() ;) {
 customerArray = (String[])customerElement.nextElement();
 Customer aCustomer = new Customer(customerArray);
 tempCust.addElement(aCustomer);
 };
 setCustomers(tempCust);

 return;
}
```

As stated earlier, the Persistency Manager is responsible for accessing system resources using a specific Persistency Manager. For that reason, we must tell the CustomerPortfolio which Persistency Manager to use. In this case, we want to access the data that resides on the AS/400 system through the JDBC interface. We have to use the setPersistencyManager() method and pass in an instance of a PersistencyManager (which concrete PersistencyManager to use). In this case, we use an instance of JdbcCustomerPortfolioPersistencyManager.

We describe how this specific JdbcCustomerPortfolioPersistencyManager implementation looks in the next section.

10.3.2 Persistency Manager Interface

The Persistency Manager interface defines the methods that each of the concrete Persistency Managers must implement. This is the interface that the Object Managers use to retrieve information from the server system. As long as the Object Managers use only these methods, the Persistency Managers can be exchanged easily.

Now, look at one specific implementation of a Persistency Manager, JdbcCustomerPortfolioPersistencyManager. Since this class implements the PersistencyManager interface, we have to provide at least two methods defined by the interface:

- getAll()
- getAt()

Here is the code for the getAll() method.

Note: We tried to reuse as much code from the previous examples as possible.

At the beginning, we have to make sure that the select statement (that is, the statement with the JDBC SQL SELECT definition) is set up properly and initialized (see the prepareSqlStatement() method).

After the selectStatement variable is set, we can execute the query, which returns a result set with the customer data in it. Since the interface between the Object Manager and the Persistency Manager is at the array level (see Section 10.2.1, “Composition of Objects” on page 257), we have to convert the data from the result set to a String[] and add each array element to a Vector named tempCust.

```
/**
* getAll method comment.
*/
public java.util.Vector getAll() {

 System.out.println("getAll() in
     JdbcCustomerPortfolioPersistencyManager:");
 if (selectStatement == null){
 prepareSqlStatement();
 };

 java.util.Vector tempCust = new java.util.Vector();

 try{
 java.sql.ResultSet rs = selectStatement.executeQuery();
 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");
 tempCust.addElement(array);
 };
 }
 catch (java.sql.SQLException e) {e.printStackTrace();}
 customerArray = tempCust;
 return tempCust;
}
```

As in the previous examples, we also show the code for the getAt() method.

Since, in our implementation, the customer file consists of only a few records, we can use an easy approach to retrieve the correct customer. We simply call the getAll() method first and scan sequentially through the vector returned, comparing each element key with the key value passed in as the method argument. In a real life application, we can use prompters, so the user can search for a customer using different search criteria.

When you look at the signature of this method, there are two points that need to be explained:

- Use a String[] as an argument, because we may also support compound keys. Using this approach, we can handle these keys as well.
- Use a vector as a return value, because we are not sure if our request will return zero, one, or even more elements.

In our case, we have one key value. As soon as we get a customer with the corresponding key, we add it to the vector and return.

```
/**
* getAt method.
*/
public java.util.Vector getAt(java.lang.String[] keys) {
 System.out.println("getAt() in
    JdbcCustomerPortfolioPersistencyManager:");
 if (customerArray.isEmpty()) getAll();
 for (java.util.Enumeration e = customerArray.elements() ;
  e.hasMoreElements();) {
 String[] aCustomer = (String[])e.nextElement();
 if (Integer.parseInt(aCustomer[0]) == Integer.parseInt(keys[0])){
 java.util.Vector returnValue = new java.util.Vector(1);
 returnValue.addElement((Object)aCustomer);
 return returnValue;
 };
 }
 return new java.util.Vector();
}
```

What have we done so far? We defined the interface to the domain object CustomerPortfolio (that is the methods getCustomers() and getAt()) and implemented the corresponding Persistency Manager, JdbcCustomerPortfolioPersistencyManager. In the next section, we try to use these newly developed classes without a user interface (by using the VisualAge for Java Scrapbook).

10.3.3 Using the Domain Object Interface

Our intention is to create an object model that is capable of running on its own. You can use it even without any user interfaces. The advantage of this approach is that you can use your object model in any context you want. For example, you can use a GUI on top of your object model when you want to create a stand-alone Java application. However, you can use exactly the same model when you have to develop a servlet application, for example. In that case, all you have to do is to replace your GUI front end with a layer that is capable of generating HTML. Or, you may want to implement a distributed application where some parts of your object model run on a server.

As you can see, separating the layers properly and using a clean and well-defined interface between the layers is important if you want to take advantage of object-orientation.

In Chapter 11, “Graphical User Interface Considerations Using Java” on page 287, we cover designing GUIs when using Java. In that chapter, we design a new GUI for our Order Entry application. Our new object-oriented implementation of the application, where the user interface is independent of the rest of the application, makes it easier to implement the new user interface into our application.

Now, we can start putting it all together. First, we try to "run" our object model just using scripts in the VisualAge for Java Scrapbook. We create a new instance of CustomerPortfolio, set the Persistency Manager, and ask for a list of customers.

To use the following examples, make sure you load the packages DomainObjects, PersistencyManagers, and Utilities. After the classes are loaded into your Workbench, open the Scrapbook, and try to run (inspect) the following code:

Application Examples

All of the application examples discussed in this redbook are available for downloading from our Web site. Refer to Appendix A, “Example Programs” on page 415, for details on how to access the redbook Web site. If you want to run the code example, follow the instructions in the readme file to install both the client code and the AS/400 libraries.

```
// select all the text and press Ctrl-q or inspect

// create the instance
DomainObjects.CustomerPortfolio customerPortfolio =
 new DomainObjects.CustomerPortfolio();

// set the persistency manager
customerPortfolio.setPersistencyManager(
 new PersistencyManagers.JdbcCustomerPortfolioPersistencyManager());

// get a list of all customers
customerPortfolio.getCustomers();
```

After you log on to the AS/400 system, an inspector window on java.util.Vector should appear. Try to determine if the correct data is retrieved by expanding the Object[] elementData. You should see the elements in the vector, each representing a customer.

Now, try to retrieve a customer based on a key. Again, in the Scrapbook, select the code and inspect it:

```
// select all the text and press Ctrl-q or inspect

// create the instances
DomainObjects.CustomerPortfolio customerPortfolio =
 new DomainObjects.CustomerPortfolio();
DomainObjects.Customer customer = null;

// set the persistency manager
customerPortfolio.setPersistencyManager(
 new PersistencyManagers.JdbcCustomerPortfolioPersistencyManager());

// get customer with CID = 5
customer = customerPortfolio.getAt("5");
```

There should be a single instance of customer back with Customer ID = 5.

We have already discussed the fact that when using a proper design, you can switch from one Persistency Manager to another without altering the application itself. Suppose, you want to use a stored procedure that calls an AS/400 program to retrieve the data. You simply need to implement the new behavior in a new Persistency Manager (for example, StoredProcedureCustomerPortfolioPersistencyManager) and reconfigure the application, so it uses the new functionality.

In the next example, we use the StoredProcedureCustomerPortfolioPersistencyManager to access the AS/400 database through a stored procedure. As in the previous examples, select the code in the Scrapbook and inspect it:

```
// select all the text and press Ctrl-q or inspect

// create the instance
DomainObjects.CustomerPortfolio customerPortfolio =
 new DomainObjects.CustomerPortfolio();

// set the new StoredProcedureCustomerPortfolioPersistencyManager
customerPortfolio.setPersistencyManager(
 new PersistencyManagers.
 StoredProcedureCustomerPortfolioPersistencyManager());

// get all customers
customerPortfolio.getCustomers();
```

After these examples, we are ready to change our initial GUI (the classes OrderEntryWdw2, SltCustWdw, and SltItemWdw) to use our new object model. The next section shows you steps that are needed to change the window classes.

10.3.4 Integrating with Window Classes

This section covers the steps that are necessary to use the new object model. The new window classes are contained in the ModifiedViews package. To load and run the new Order Entry application, make sure these packages are loaded into your workspace: DomainObjects, ModifiedViews, PersistencyManagers, and Utilities. They are all contained in the OrderEntry project, as shown in Figure 218 on page 269.

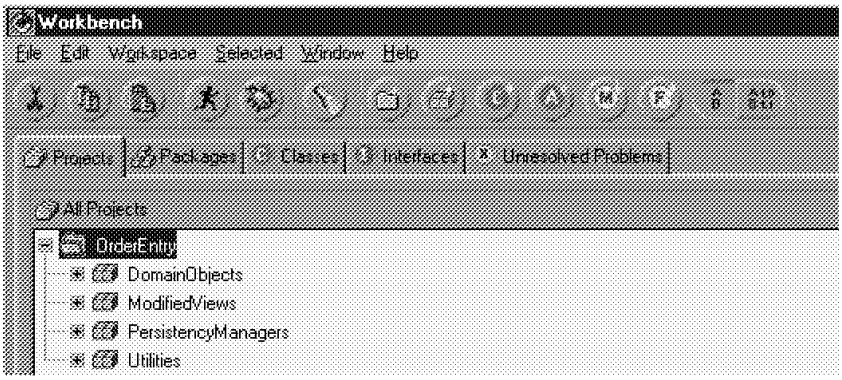


Figure 218. The OrderEntry Package

Note: We take you through the steps that are necessary to convert the original application, as described in Chapter 7, “Moving the Server Application to Java” on page 133, to the new object model. The complete application is in the OrderEntry package.

To use our new object model, we start to change the customer selection window, SltCustWdw. After that, we have to modify the item selection window SltItemWdw. Finally, we have to modify the main window, OrderEntryWdw2, as well. We assume a working knowledge of the basic concepts and the different browsers available in VisualAge for Java.

10.3.4.1 SltCustWdw

In this window, we want to integrate the functionality that was already tested in the Scrapbook (see previous section). To get access to the methods provided by the CustomerPortfolio class, we place a new bean to the free form surface of the Visual Composition Editor (VCE). We name the bean CustomerPortfolio. Next, we create an event-to-method connection by connecting the windowOpened() event to the Get all Customers method of the CustomerPortfolio bean, as shown in Figure 219 on page 270.

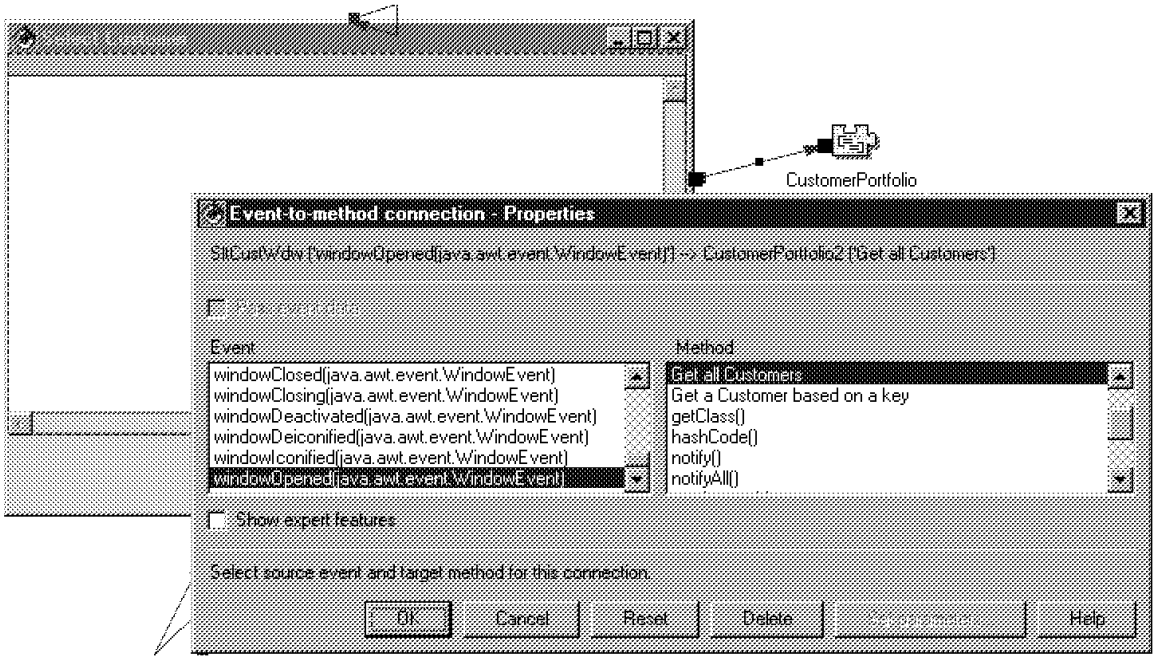


Figure 219. Creating a Connection

Should we use the getCustomer() method to retrieve the list of customers? We have introduced another possibility of retrieving data from a bean: the possibility of using events. With events, you can decouple your GUI even more from the object model. Look at the example provided, Open the CustomerPortfolio BeanInfo page. By selecting the method feature *Get all Customers*, you can see that the actual method executed is getAll(), as shown in Figure 220 on page 271. Maybe you remember that in case the CustomerPortfolio instance variable customers is null, the method getAll() runs. Within this method, we are setting the variable *customers* using its setter method setCustomers(java.util.Vector).

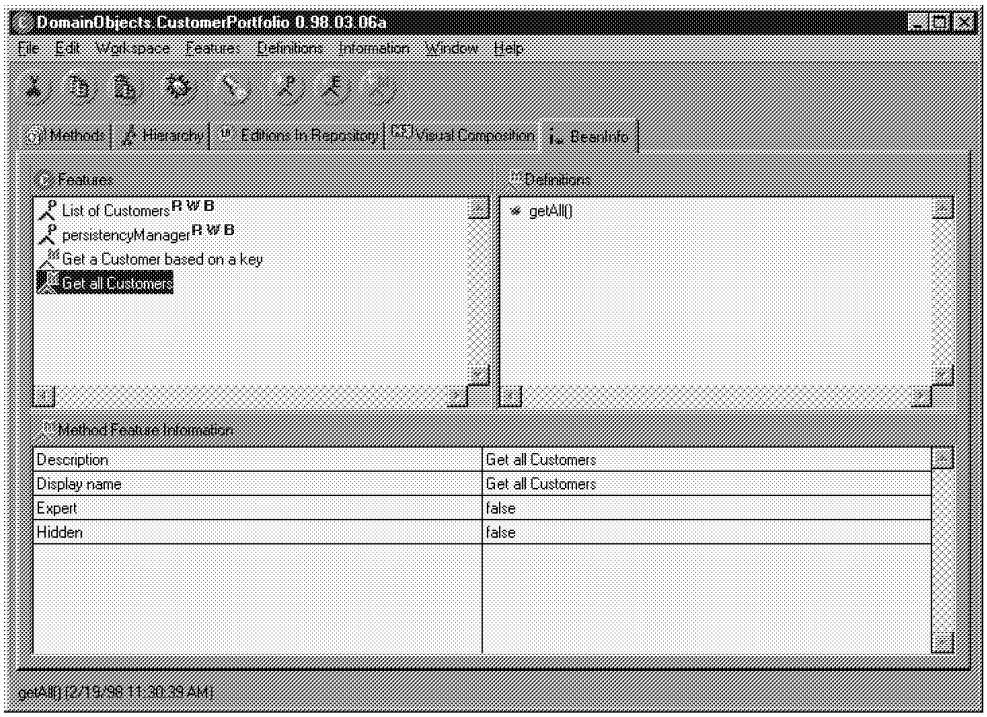


Figure 220. CustomerPortfolio BeanInfo

Now, look at the property feature List of Customers that is provided by CustomerPortfolio, as shown in Figure 221. The methods used are getCustomers() and setCustomers().

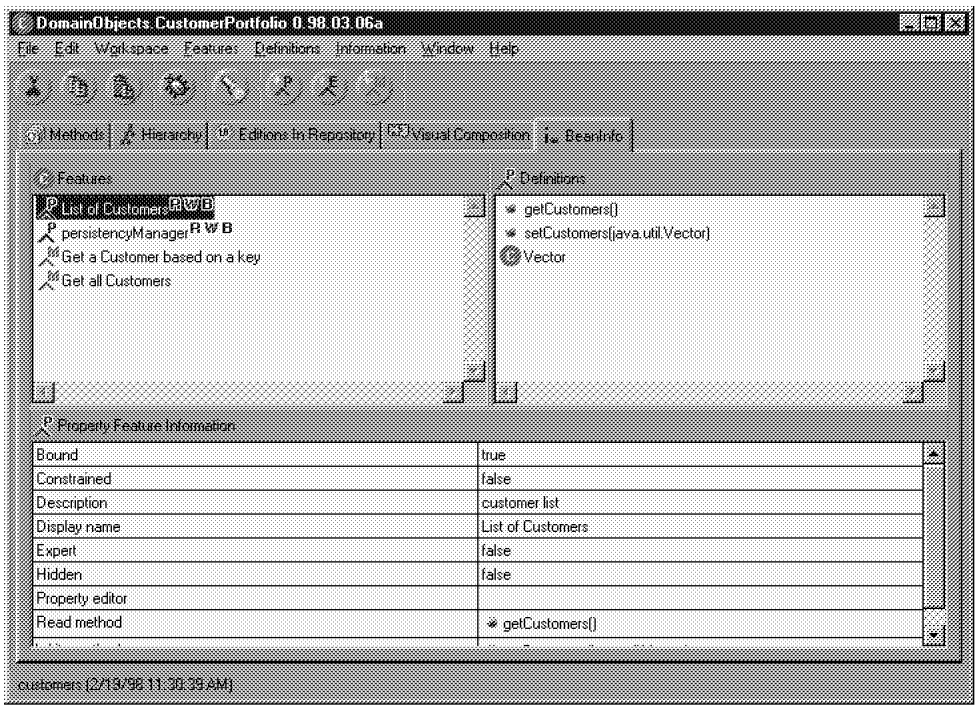


Figure 221. CustomerPortfolio BeanInfo

Since this is a bound property, whenever the value changes, an event is fired. And, this event populates the customer list box in the SltCustWdw window. The advantage of this event-driven concept is that you can use threads, for example, to run long running tasks in the background. As soon as the task has finished its work and has changed a property value, you are notified by an event. Without using events, you either have to poll the background task or wait until it finishes (that is, you block the application). We continue with our modification on the SltCustWdw window.

Next, we must create an event-to-script connection from the bean part List of Customers event to the script populateCustBox(). This causes the populateCustBox() method to run when the List of Customers event is fired. The populateCustbox() method actually populates and displays the customers in a multi-column list box, as shown in Figure 222.

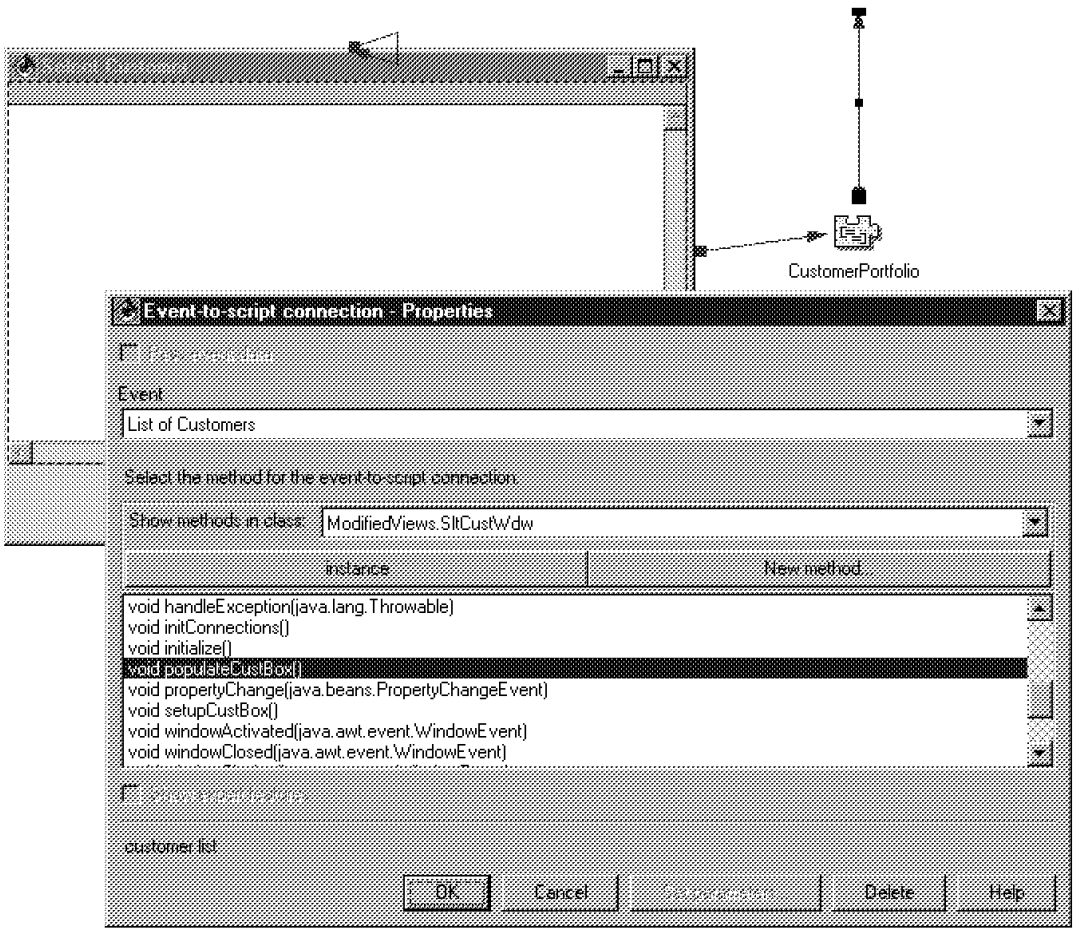


Figure 222. CustomerPortfolio Event to Script

The other connections remain the same as in the old version. The visual composition window should appear similar to the one shown in Figure 223 on page 273.

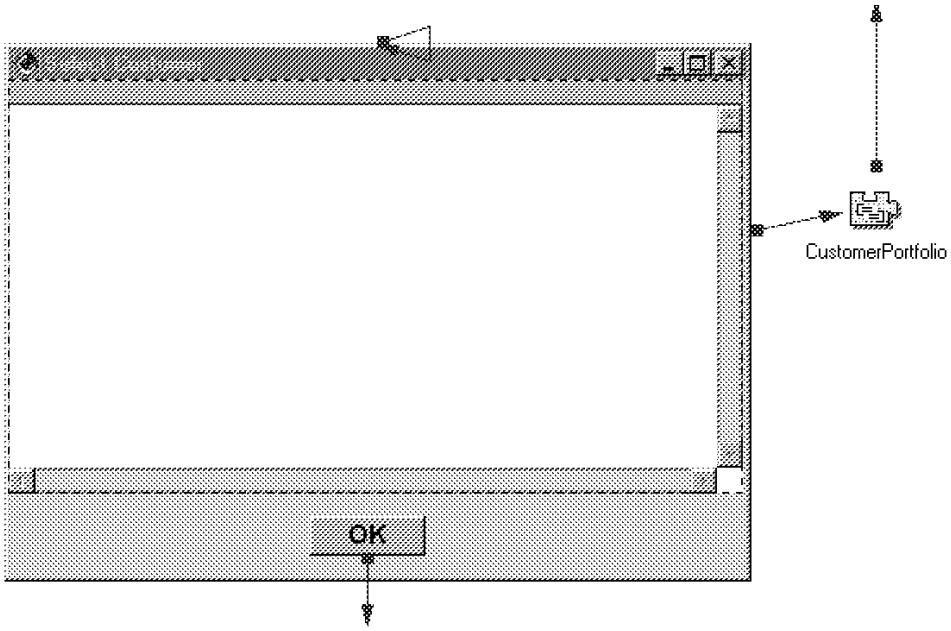


Figure 223. SltCustWdw Visual Composition Editor

Now, we save the window bean by pressing CTRL+F2. We switch to the methods page and select the method populateCustBox(). To access the customer list, we use the CustomerPortfolio getCustomers() method. This method returns a Vector of Customer instances. We convert them into a format that is suitable for our list box. The final populateCustBox() is shown here:

```
/**
* This method was created by a SmartGuide.
*/
private void populateCustBox() {
 orderWindow.updateStatus("Retrieving customer list...");
String[] array = new String[9];
java.util.Vector customerList = getCustomerPortfolio().getCustomers();
 for (java.util.Enumeration e = customerList.elements() ;
    e.hasMoreElements();) {
 DomainObjects.Customer aCustomer =
      (DomainObjects.Customer)e.nextElement();
 array[0] = aCustomer.getId().toString();
 array[1] = aCustomer.getLastName();
 array[2] = aCustomer.getFirstName();
 array[3] = aCustomer.getInit();
 array[4] = aCustomer.getAddress();
 array[5] = aCustomer.getAddress();
 array[6] = aCustomer.getCity();
 array[7] = aCustomer.getState();
 array[8] = aCustomer.getPostCode();
 getCustMLB().addRow(array);
 }
 this.repaint();
 orderWindow.updateStatus("Customer list retrieved");
 return;
}
```

Since we are using an event-driven approach now, we remove the populateCustBox() call within the constructor method public SltCustWdw (OrderEntryWdw2 orderWdw, Connection dbConnect).

Note: We have removed all code that is bound to a specific access technique. If we switch from one access method to another, we do not have to modify this method at all. This leads us to the next question.

How does the window know which access method is used to retrieve the data from the AS/400 system? As in the Scrapbook examples, we specify the correct Persistency Manager. In this case, we set the Persistency Manager in the initialize() method. As you can see, we are using the JdbcCustomerPortfolioPersistencyManager to access the server resources:

```
/**
* Initialize the class.
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void initialize() {
// user code begin {1}
// user code end
setName("SltCustWdw");
setName("SltCustWdw");
setTitle("Select Customer");
setLayout(null);
setBackground(java.awt.Color.cyan);
setSize(425, 285);
setResizable(false);
add(getCustMLB(), getCustMLB().getName());
add(getSltCustBTN(), getSltCustBTN().getName());
initConnections();
// user code begin {2}
getCustomerPortfolio().setPersistencyManager(
 new PersistencyManagers.JdbcCustomerPortfolioPersistencyManager());
// user code end
}
```

Finally, we change the custSelected() method. It looks like this after modification:

```
/**
* This method was created by a SmartGuide.
* This method constructs a Customer object based upon the
 select row in the customer list box.
*/
private void custSelected() {
 Object[] selectedRow = getCustMLB().getSelectedRow();
DomainObjects.Customer custSelected =
getCustomerPortfolio().getAt(selectedRow[0].toString());
orderWindow.setSelectedCust(custSelected);
// close down
 this.dispose();
 return;
}
```

We use the CustomerPortfolio to construct a new Customer based on the key that we get from the list box. Before closing the window, we store the Customer instance in the parent window by using the setSelectedCust() method.

Configuring the CustomerPortfolio Visually

In the previous example, we used scripts to specify the correct PersistencyManager for the CustomerPortfolio. Since we are using a visual development environment, we can also use the VCE to configure the Domain Object Managers. To use a visual approach, we follow these steps:

1. Open SltCustWdw in the VCE.
2. Add two other beans:
   - The JdbcCustomerPortfolioPersistencyManager bean (JdbcCustomerPortfolioPM)
   - The StoredProcedureCustomerPortfolioPersistencyManager bean (StoredProcedureCustomerPortfolioPM)
3. Tell the CustomerPortfolio which Persistency Manager it has to use.

For example, if we want to use the JdbcCustomerPortfolioPersistencyManager to access the database, we connect the event windowOpened from the SltItemWdw to the CustomerPortfolio persistencyManager method. Then, we visually set the connections value parameter to the Persistency Manager that we want to use. In this case, we use the JdbcCustomerPortfolioPersistencyManager (JdbcCustomerPortfolioPM).

Note: The dialog that pops up when you drop the connection can be ignored. After we finish the connections, the VCE looks similar to Figure 224.

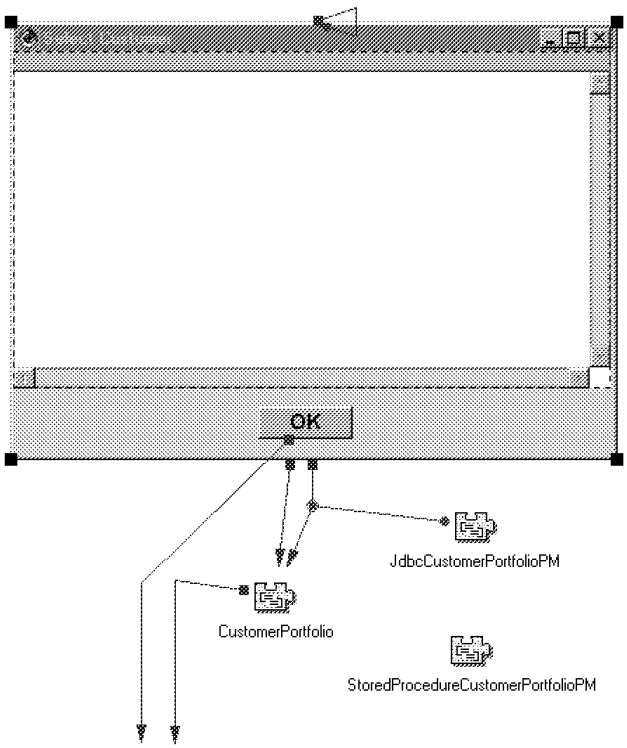


Figure 224. SltCustWdw Visual Composition Editor

Before saving the bean, we have to check the order that the connections are fired in. This is important, because we use the same event (windowOpened) to run two actions sequentially. We encounter an error if we try to call the getAll() method

before we set the PersistencyManager properly. To check the sequence of the methods that are running, we open the pop-up menu over the SltCustWdw part and select the Reorder Connection From option, as shown in Figure 225.

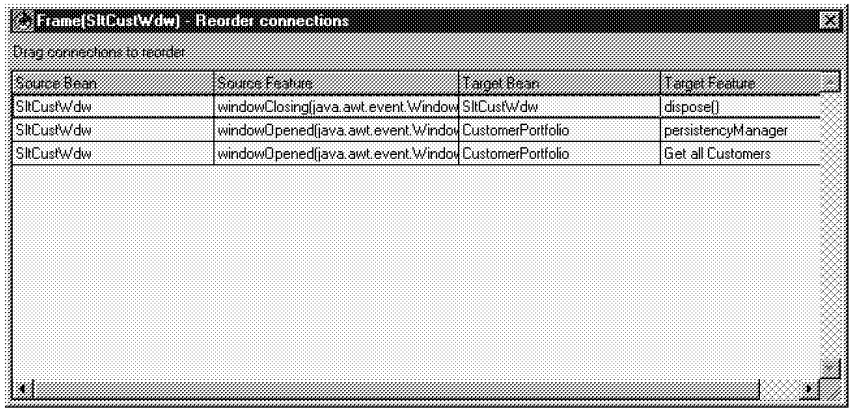


Figure 225. SltCustWdw Reorder Connections

Make sure that the method persistencyManager runs before the Get all Customers (see column Target Feature). If you have to change the sequence, just select and drag a row to the new location.

Keep in mind that when you use the visual approach, you do not have to modify the initialize() method at all. If you decide to use a different Persistency Manager, you just have to switch the parameter-from-property connection to the new Persistency Manager and save the bean.

Now, we are done with the modifications on the SltCustWdw class. The next step is to change the other select window, SltItemWdw. These modifications are described in the next section.

10.3.4.2 SltItemWdw

The modifications in this window are similar to those described for the SltCustWdw window. For that reason, we do not discuss all the changes in depth again. We simply list the steps and provide the code snippets where appropriate:

1. We open the SltCustWdw class.
2. In the VCE, place a new Catalog bean on the free form surface. We name the bean Catalog as well.
3. Connect the event windowOpened from the SltItemWdw to the Catalog getAll() method.
4. Connect the event items from the Catalog bean to the Script populateItemBox().

The window appears similar to the one shown in Figure 226.

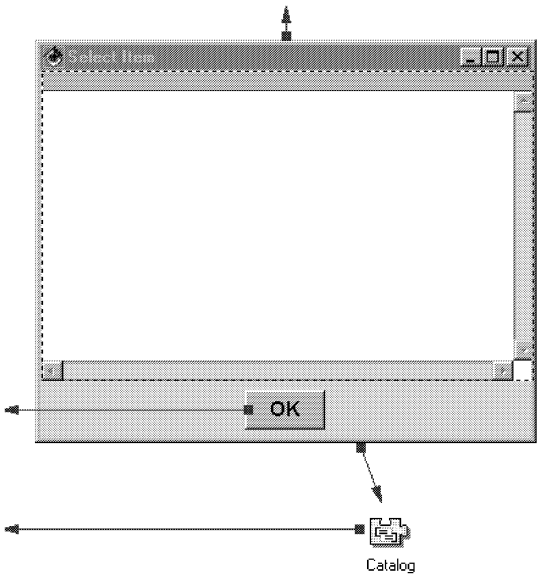


Figure 226. SltItemwdw Window

5. Save the SltItemWdw bean by pressing **Ctrl+F2** and switch to the methods page.
6. Select the method **populateItemBox()** and enter this code:

```
private void populateItemBox()
{
 // The result set that is returned represents records that
 // have 4 fields of data. These fields will be stored in
 // an array of strings
String[] array = new String[4];
java.util.Vector allItems = getCatalog().getItems();
for (java.util.Enumeration itemsE = allItems.elements();
   itemsE.hasMoreElements();) {
 DomainObjects.Item anItem = (DomainObjects.Item)itemsE.nextElement();
 array[0] = anItem.getId();
 array[1] = anItem.getName();
 array[2] = "$" + anItem.getPrice().toString();
 array[3] = Integer.toString(anItem.getQuantity());
 getItemMLB().addRow(array);
}
orderWindow.updateStatus("Item list retrieved");
return;
}
```

7. Save the method.
8. Select the method **itemSelected()** and enter the following code.

   Note: We are using a different approach for creating a new instance of Item. We use the information from the item list box and instantiate a new item using this information.

```
private void itemSelected() {
Object[] selectedRow = getItemMLB().getSelectedRow();
// declare an array of String the same size as the row
```

```
String[] itemInf = new String[selectedRow.length];
// retrieve each item in the selected row, convert to
// String and put it into the String array
for(int i=0;iselectedRow.length;i++){
 itemInf[i] = selectedRow[i].toString();}
 // instantiate an Item object and pass it to the
 // setSelectedItem() method of the order window
DomainObjects.Item itemSelected = new DomainObjects.Item(itemInf);
orderWindow.setSelectedItem(itemSelected);
// close down the list window
 this.dispose();
 return;
}
```

9. Save the method.
10. Select the constructor method **SltItemWdw()** with the two arguments. Comment out the line where the method populateItemBox() is called.
11. Save the method.
12. Select the **initialize()** method and enter this code after the line:

```
// user code begin {2}.
getCatalog().setPersistencyManager(
 new PersistencyManagers.
   StoredProcedureCatalogPersistencyManager());
```

13. Save the method.

Now, we have finished the modification necessary in the class SltItemWdw to use the new object model.

Similar to the SltCustwdw window in the previous section, we use the VCE to visually control the Persistency Manager. Using visual connections, the window appears similar to the one in Figure 227.

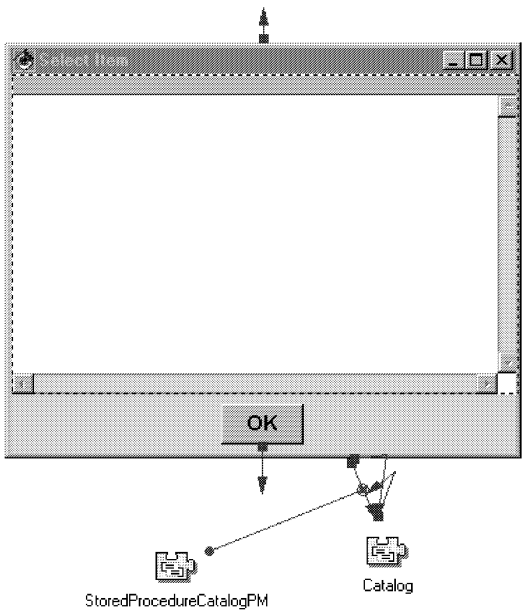


Figure 227. SltItemwdw Window

The next step in this process is to change the parent window of the two selection windows, the class OrderEntryWdw2.

10.3.4.3 OrderEntryWdw2

This section shows you how to introduce a totally new design without changing the entire application. For that reason, some of the "legacy code" is still in place (for example, the import statement for java.sql.* and the method getSQLConnection()). Now, we open the OrderEntryWdw2 class.

First, we change the class definition itself. We add these instance variables:

```
private DomainObjects.Catalog catalog = new DomainObjects.Catalog();
private DomainObjects.Customer customer = null;
private DomainObjects.CustomerPortfolio customerPortfolio =
 new DomainObjects.CustomerPortfolio();
private DomainObjects.Item item = null;
private DomainObjects.Order order = null;
private DomainObjects.OrderDetail orderDetail = null;
private DomainObjects.OrderPortfolio orderPortfolio =
 new DomainObjects.OrderPortfolio();
private DomainObjects.Customer selectedCustomer = null;
```

We open the window in the VCE. We select the connection from the Submit button that runs the script retrieveOrderInfo() and open the properties dialog. Here, we press the **New method...** button, and in the next dialog enter `void submitOrder()` in the Method Name entry field, as shown in Figure 228 on page 280. Then, we click **Finish** and click **OK** to confirm. Later, we add the code to this method to do the actual order entry processing.

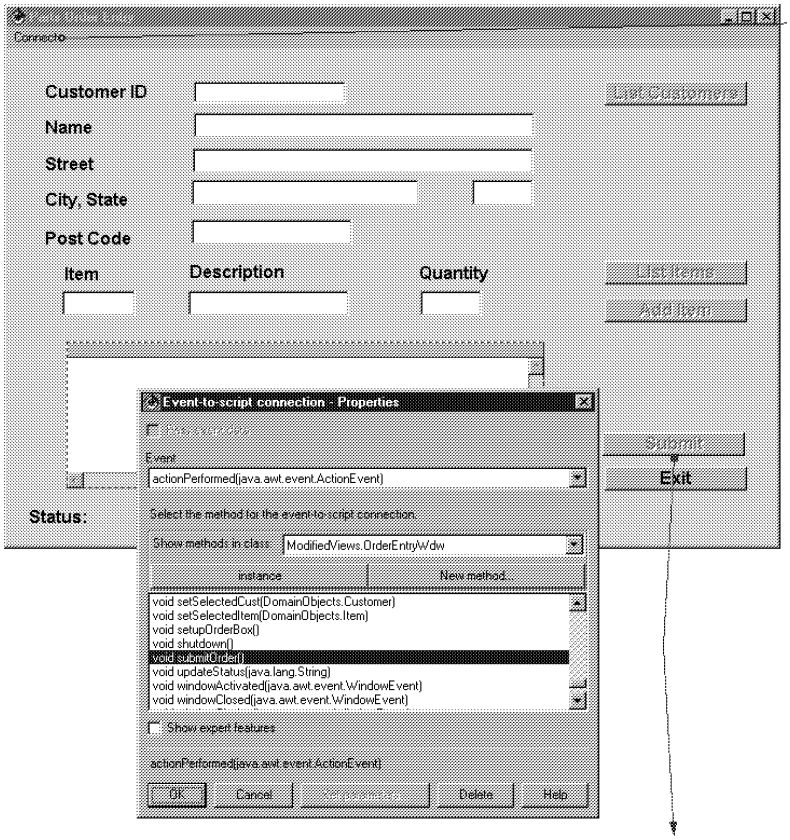


Figure 228. OrderEntry Window

The next method is the addOrderItem() method. This method runs when you select an item, add the quantity you want to order, and press the Add Item button. This method is responsible for checking to see if you have entered a valid item ID. In the old version, this is done using direct DDM access to the item file. If the item ID is valid, a new Item is created and added to the order list box.

Who is actually responsible for checking to see if an item ID is valid? What do we, as humans do, to check in a catalog to see if we can find a corresponding item that matches a certain identifier? We do it the same way. We can ask our Catalog class for a certain Item based on a key (that is, the identifier). If the Catalog has such an Item, it returns it. Otherwise, null is returned. Here is how it works:

1. Ask the catalog for an item with a certain key. If the catalog finds such an item, it returns it and we store it in the variable itemToAdd. If this variable is null, we know that the item ID is not valid and we issue a message describing the fact. If the item ID is valid, we create:
   - An **Object[]** to display the item and its quantity in the order list box
   - A new **OrderDetail** instance that contains the Item and quantity ordered
2. Add the OrderDetail instance to the Order (order.add(orderDetail)). Later, we see that adding the OrderDetail to the Order at this time makes the processing of the order entry much easier.

3. Now, we have to add the following code to the method addOrderItem():

```
private void addOrderItem(String key) {
updateStatus("Verifying order item...");
DomainObjects.Item itemToAdd = catalog.getAt(key);
if (itemToAdd == null) {
 updateStatus("Invalid item...");
 return;
};
 Object[] orderRow = new Object[4];
 orderRow[0] = itemToAdd.getId();
 orderRow[1] = itemToAdd.getName();
 orderRow[2] = itemToAdd.getPrice();
 orderRow[3] = getQtyTF().getText();
 getOrderMLB().addRow(orderRow);
 getOrderMLB().repaint();
 getSubmitBTN().setEnabled(true);
 orderDetail = new DomainObjects.OrderDetail(itemToAdd,
Integer.parseInt(getQtyTF().getText()));
 order.add(orderDetail);
 updateStatus("Item added: please add more or submit");
return;
}
```

4. Select the **connectToDB()** method and replace it with this code:

```
private void connectToDB(String systemName, String userid,
    String password) {
 updateStatus("Connecting to " + systemName + " ...");
 this.systemName = systemName;
 this.userid = userid;
 this.password = password;
try{
 Utilities.AS400ConnectionManager.getAS400(systemName, userid,
     password);
 dbConnect = Utilities.AS400ConnectionManager.getDbConnect();
}
catch(Exception e){
 updateStatus("Connect failed");
 handleException(e);
 return;
}
updateStatus("Connected to " + systemName);
 return;
}
```

5. Modify the connectToDB() method to use as much of the "old" code as possible. That way, that we can still use some of the concepts that actually do not fit into this context. The variable dbConnect is one of those concepts. But, since we pass this variable to the selection windows SltCustWdw and SltItemWdw we keep it "as is". Of course, in a proper object-oriented application, this dbConnect functionality is hidden in a Persistency Manager. But as stated earlier, our intention is to introduce the new object model without modifying the entire GUI.

6. Replace the next method, disconnectFromDb(), with the following code:

```
private void disconnectFromDb() {
if(dbConnect != null) {
 try {
 dbConnect.close();
 dbConnect = null;
 }
 catch(SQLException e){
 updateStatus("Error closing SQL Connection");
 handleException(e);
 return;}
 }
 }
updateStatus("Disconnected: Choose Connect on Menu");
return;
}
```

7. Change the initialize() method, as done in the two previous windows: SltCustWdw window and SltItemWdw window. In this method, we tell the Domain Object Managers (for example, in this case, the Catalog, CustomerPortfolio, and OrderPortfolio) which Persistency Managers to use.

```
//user code {2}:
try {
 catalog.setPersistencyManager(
 new PersistencyManagers.StoredProcedureCatalogPersistencyManager());
 customerPortfolio.setPersistencyManager(
 new PersistencyManagers.JdbcCustomerPortfolioPersistencyManager());
 orderPortfolio.setPersistencyManager(
 new PersistencyManagers.OrderEntryJdbcPersistencyManager());
}
catch(Exception e) {e.printStackTrace();};
```

8. Delete the methods named openItemFile(), prepareOrder(), remoteSubmit(), retrieveOrderInfo(), and submitOrder().

9. Update the methods named setSelectedCust() and setSelectedItem(). To update them, add the following two lines right after the method definition:

```
public void setSelectedCust(Customer selectedCust) {
customer = selectedCust;
order = orderPortfolio.newOrderFor(customer);
....
```

10. Select Customer in the SltCustWdw window and press the OK button. Now the setSelectedCust() method runs.

11. Create a new Order object for the selected Customer. The OrderPortfolio is responsible for creating a new Order instance. We can use the method newOrderFor(), implemented by the OrderPortfolio to create this new instance. We store the newly created instance in the variable called *order*. Before saving the method, we change the arguments of this method to (DomainObjects.Customer selectedCust).

12. Change only the method arguments in the next method, setSelectedItem(). Specify the package name where the class Item can be found (DomainObjects.Item selectedItem).

13. Change the submitOrder() method. Since we completed our order object when we selected a new customer or added a new item to the list, processing the order is simple. Add this code to the method submitOrder():

```
updateStatus("Processing order ...");
orderPortfolio.submitOrder(order);
```

As you can see, we delegate the order entry process to the order portfolio. This class takes care of the correct processing. We explain how this processing is actually done in the following section.

10.3.5 Processing the Order

For our explanation of the order processing, we start with the submitOrder() method in the OrderPortfolio class. When you look at this method, you can see that we use the associated Persistency Manager to perform the actual processing. You may remember that we configured our application to allow OrderPortfolio to use the OrderEntryJdbcPersistencyManager to access the AS/400 system. Let us examine the submitOrder() method implemented by this class.

This method looks familiar, doesn't it? It is actually almost the same code found in the original OrderEntryJDBC method commitOrder():

```
public void submitOrder(DomainObjects.Order anOrder)
 {
 try {
 System.out.println("commitOrder: Started processing part order");

 // Extract the customer number and count of lines
 String customerNumber = anOrder.getCustomer().getId().toString();
 BigDecimal orderLineCount = new BigDecimal(
    anOrder.getOrderDetails().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 (Utilities.AS400ConnectionManager.getDbConnect().
      getTransactionIsolation()
      != java.sql.Connection.TRANSACTION_NONE){
 Utilities.AS400ConnectionManager.getDbConnect().commit();}

 // Initiate order printing
 writeDataQueue(customerNumber, orderNumber);
 }
catch(Exception e) {e.printStackTrace();}
return;}
```

Since we implemented the application using an object-oriented design in which we have separated the application into distinct layers, we can easily change the way we actually do the order processing. We currently have our application configured to use the OrderEntryJdbcPersistencyManager to do the order processing. In this implementation, most of the work is done on the client through JDBC calls. Suppose we want to change the order processing so that it actually runs on the AS/400 system. We can do this by changing our application to use the OrderEntryRMIPersistencyManager.

This Persistency Manager uses the Remote Method Invocation (RMI) interface to call a Java program that runs on the AS/400 system. That program then handles the order processing using JDBC, but it actually runs on the AS/400 system. This is similar to the application described in Chapter 7, “Moving the Server Application to Java” on page 133. To make this change, we change one line of code in the initialize() method of the OrderEntryWdw2 class:

```
//user code {2}:
try {
      catalog.setPersistencyManager(
 new PersistencyManagers.StoredProcedureCatalogPersistencyManager());
 customerPortfolio.setPersistencyManager(
 new PersistencyManagers.JdbcCustomerPortfolioPersistencyManager());
 orderPortfolio.setPersistencyManager(
 new PersistencyManagers.OrderEntryRMIPersistencyManager());
      }
catch(Exception e) {e.printStackTrace();};
```

Now, the code that runs when we call the submitOrder() method uses the RMI interface:

```
public void submitOrder(DomainObjects.Order anOrder) {

 // set an RMISecurityManager - if we have none
 if(System.getSecurityManager() == null){
  System.out.println("creating new RMISecurityManager...");
  System.setSecurityManager(new RMISecurityManager());
 };

 // obtain reference to the remote OrderEntryDDM object
 if(orderHandler == null){
  try{
   System.out.println("Lookup server " + hostServer + " on port "
   + port);
   serverURL = "//"+hostServer+":"+port+
     "/OrderEntryJdbcRMIPersistenyManager";
   orderHandler = (OrderEntryRMII)Naming.lookup(serverURL);
   System.out.println("Lookup OK");

  }
      e.printStackTrace();
       return;
    }
   }
   try {
  System.out.println("Submitting the order to the remote server...");
  orderHandler.submitOrder(anOrder);
  System.out.println("Order successfully processed!");
```

```
    }
    catch (RemoteException re) {re.printStackTrace();}

  return ;
}
```

By using our new object-oriented application, we can easily change the order processing from a "fat" client implementation to a "thin" client implementation. We just have to set the orderPortfolio Persistency Manager to use the desired concrete Persistency Manager.

As mentioned at the beginning of this chapter, we want to show you one possible approach to object-orientation. As you can see in Figure 214 on page 256, there are some parts missing in our object model (for example, the classes *Stock* or *District*). These classes can be used in the previous submitOrder() method, for example. With a proper object-oriented implementation in place, you can now introduce an entire new set of classes without interfering with code that was already developed (that is, the user interface).

Maybe you have seen it already, but we have broken our rule that the Domain Object Managers have to use the Array Level Interface when they want to communicate with the Persistency Managers (see Figure 215 on page 258). The reason for this inconsistency is that we tried to reuse as much code as possible and simply wrap the old code in a new class.

What is the solution? One possible way is to let the OrderPortfolio do most of the work instead of simply delegating the whole order entry processing to its Persistency Manager. To achieve that, it is necessary to introduce some new classes previously described. For example, the class District can implement a method that allows us to get the next valid order number. Or, the class Stock can implement a method that we can use to decrease the stock according to the quantity of a certain item ordered. In addition, the class CustomerPortfolio can implement a method to set the new balance for a specific customer.

As you can see from this list, there are still many things to do. However, keep in mind that with this design, you can introduce new concepts without the need of modifying already existing and working application parts (as long as the interface remains the same, of course).

We invite and encourage you, as the reader, to develop these classes and try to integrate them into the already existing parts. Also, try to apply the same concept that we just described by properly separating the different layers. With a proper design, it is possible to write code that is easy to use, extend, and most important to maintain.

Chapter 11. Graphical User Interface Considerations Using Java

This chapter shows you how to create graphical user interfaces (GUIs) with Java. If this is your first introduction to building GUIs, this chapter is a good starting point. Before you undertake any serious development work, you need a more complete understanding of GUI analysis and design, along with a good background in object design. This chapter provides you with the necessary knowledge to build prototype GUIs in the Java environment, so you can make informed decisions on the direction that you want your application development to take. We start with an overview of some basic GUI principles, and then look at what Java provides for us to construct GUIs. At the end of this chapter, see the section that contains references for current information on these topics.

11.1 General GUI Guidelines

This section provides a brief glimpse of important considerations when creating a GUI, regardless of the language or toolset that you use.

11.1.1 What Makes a Good GUI

A good GUI is one that allows the application user to perform the task that they are trying to complete quickly with the least amount of effort and minimal initial instruction. Some characteristics of a good GUI are:

- You feel in control:

  When you are presented with a window, you feel confident about what to do because it relates to the work that you are trying to do and are apparent to you.

- Guides and interacts with you:

  Moves the cursor or focus to the appropriate control based on a user action.

- Prevents or forgives user mistakes:

  Disables any button that you should not push.

- Provides feedback and status:

  If a task takes longer than you think is normal, it provides a mechanism to show progress and estimated remaining time.

11.1.2 GUI Terms and Concepts

This section offers a reference for you to scan for terminology and to refer to when reading other sections:

Abstract Windowing Toolkit (AWT)
: Class library that is provided by Sun Microsystems, Inc. for constructing user interfaces.

Component
: Pieces that build GUIs (that is, buttons, text fields, containers, labels, and so on).

Control
: Another term for component.

Event
: A signal that something has happened. Examples of this are the user clicking the mouse, a window opening, or an important value that changes in the application.

Focus A control is selected programmatically or by the user.

Integrated Development Environment (IDE)
A toolset that makes it easier to write, compile, debug, and package application code (for example, VisualAge for Java).

JavaBean Reusable software component that follows a specific design pattern.

Java Development Kit (JDK)
All of the necessary software for creating Java applications.

Java Foundation Classes (JFC)
Extension to the AWT.

11.1.3 Steps to Build the GUI

This section gives you a feel for the steps that you need to follow when designing a GUI for your application. This is not a specific methodology, nor does it include all of the detail necessary to do an actual design. As you have seen in the other chapters of this book, the approach used for building this Java application is that the application consists of three separate layers:

- **Business model layer**: Business rules and logic resides
- **Persistency layer**: Function of saving and retrieving an object state is handled
- **User interface layer**

We focus on the user interface layer now. Depending on the size of the project and the development team size, some of the steps listed here may be done in conjunction with the investigation and design of the other layers. Here are some points to consider when you create your user interface:

- **Determine what will make the GUI successful**

  Work with your users, project sponsors, and design team to make sure that the factors to make this project a success are spelled out, so you have specific targets. This includes asking such questions as:

  – What tasks must the user accomplish with the GUI?

  – What computer skills does your target audience have (for example, a computer professional or no computer experience)?

  – Is the user group fairly stable or is there a lot of turnover?

  – What environment will this application run in (that is, Web application, kiosk, mobile user, in the office, all of the above, and so on)?

  – What platform will it run on (operating system, memory, processor speed, DASD, and so on)?

  – What is an acceptable response time?

  – How much resource do you have (user group's time, developers, equipment, funding, and so on)?

  – If working with new technology, what are the training needs for the development staff?

  – Are there any time frames already established?

- How will the project be tracked or managed?
- Does your company have a set of standards for naming, look and feel, and interoperability with other applications? If not, you need to include the time in this project to establish them.

- **User analysis**

  Start with the users or a user representative that is involved in the initial project startup and do these tasks:

  - Identify who your users are and find them.
  - Watch the users work.
  - If different user abilities or types are involved, make sure you visit a fair representation of all of them.
  - Listen to the terms they use and the concepts they have.
  - Identify the tasks and processes that the users perform.
  - Discuss any new function that is being added and gather user input.

- **Task analysis**

  Use the tasks that you documented in the user analysis to:

  - Provide, at a high level, a description of all the tasks that the users must perform.
  - Provide descriptions of these tasks at a level of detail, so you know who is doing the task and when it is successfully accomplished.
  - Break these tasks into specific steps that the user performs to complete the task.
  - Review and verify the tasks and steps with the users.

- **Concept analysis**

  Using the tasks and concepts that the user has, begin to translate or map them to what can be presented on a computer display. During this step, try to see where the computer can really enhance the productivity for the user. This does not provide the details of how you are going to code the user interface, but it shows what can be done and what approach to use:

  - Finalize your product, platform (size, speed, storage, capabilities, and so on), cost, and limitations.
  - What tasks may be interrelated?
  - What are the reuse possibilities?
  - How can you best represent the steps within the task in the GUI?
  - If your users have had limited or no exposure to GUIs before, you may need to demonstrate some of the capabilities to let them see the possible benefits and give them ideas.
  - Which icons or graphics have meaning for the users?
  - Are all of the tasks and objects that the user interacts with available in the business model?
  - Resolve any business model or GUI interface issues that surface.
  - Begin iterations of prototyping the GUI and review it with the users.

- **Detail design**

  Now, you design the actual layouts and content of the displays. Remember these points:

  – Do not forget to focus on what will make the project successful.

  – Be especially wary of function creep. Keep the project scoped within your time, skill, and budgetary constraints.

  – There are many variables that come into play to make a GUI visually appealing and understandable:

    - **Color**
      – Make defaults conservative.
      – Users assume objects of the same color are related.
      – Colors are not seen the same by everyone.
      – Colors mean different things in different cultures.
    - **Alignment**
      – Vertical alignment is usually preferred.
      – Top to bottom flow is most natural.
      – Indent dependent fields.
    - **White space**
      – Adds clarity.
      – Do not crowd controls with each other or the window border.
      – If the application is for international use, take translation space into account.
    - **Fonts**
      – As a general rule, do not mix fonts.
      – Defaults should be common, conservative font.
    - **Component grouping**

      Group related controls together.
    - **Graphics**
      – Make sure graphics or images serve a purpose.
      – Graphics should add clarity, not clutter.
    - **Icons**
      – Make sure the icons make sense to the user, not you the developer.
      – It is difficult to create good icons.
      – For international applications, try to use international symbols.

  – It may be wise to enlist a GUI design consultant if you have not undertaken training in this area.

  – Use appropriate controls for the task.

  – Evaluate third party GUI controls if necessary.

  – Be sure to include these in your design:

    - Error handling and recovery
    - Validation
    - Application flow
    - Keyboard shortcuts
    - Progress indicators or status messages

– Be consistent.

– Follow your company standards.

– If your project and GUIs are more complex, you may want to consider further separation of the GUI from the business model. The Model - View - Controller (MVC) model is one way to do this. In the MVC, Model is the business model, View is the actual windows that the application user sees, and the Controller is an object that sits between them. The View does not need to have knowledge of the business model and, therefore, can be more reusable. If you feel that your GUI windows are fairly stable and have high potential for reuse, you may want to consider this approach.

- **Debug iterations**

  This topic is listed before implementation to make sure that you have thought about how you are going to debug before you begin writing code. You will not get it right the first time through so plan on having some testing phases with the user built into your implementation phase. It is cheaper to catch any problems early rather than at the end when you are rolling out the product. Keep these points in mind:

  – Have periodic reviews with your development team.
  – Establish a set of users to do periodic tests.
  – Sit with the user and watch as they try to perform their tasks with the application.
  – Notice what helps and what confuses.
  – Note reactions to response times and application flow.

- **Implementing**

  If all of the preceding steps have been performed, this step should not present too many surprises unless you have picked some leading edge technology as part of your project. Remember to:

  – Take time for your reviews with the users.
  – Use the same procedures that you use for your business logic programming.

- **Testing**

  This is your final testing before installing for the users. You should:

  – Have good scripts ready to ensure thorough testing.
  – Duplicate the actual production environment.

11.2 Java GUIs Past to Present

Now that some basics of GUI design have been covered, let us look at what Java offers us for creating user interfaces.

11.2.1 The Beginnings — AWT and the Peer Model

The Abstract Windowing Toolkit (AWT) provided the initial components for building Java GUIs. This library of components allows Java developers to build GUIs where a single set of source code could be run on any platform while maintaining the native look and feel of the platform. If the application was running on a Windows system, all of the components looked and responded the same as other Windows applications. If the user was on an OS/2 system, all of the components looked and responded the same as their other OS/2 applications.

The AWT accomplished this through what they called the "peer" model. What this meant is that the AWT components created a native (Windows, OS/2, UNIX) component and wrapped it. This approach brought the AWT to market quickly and met the basic needs of the applets that began to be written, but the limitations of this model quickly became apparent. The use of peer components carried significant overhead in the actual creation of the components on the display and in the number of classes required to construct the window. The event model was also too restrictive, forcing objects to handle the events by subclassing and overriding methods. Complex logic was required to figure out which object initiated the event. This original architecture was a start, but changes needed to be made to allow Java to support the creation of a wide range of applications. The limitations of this original release are addressed in the next release.

Before we move on, the original upgraded AWT library still provides the base set of components for building GUIs, so let's take a look at some of the basic components that you use to construct a GUI. The components reviewed here are the ones you will most likely encounter when building your first GUI with an IDE. To view all of the classes that are available in the AWT, see the Java documentation that is available with the version of JDK that you have on your PC. As you become more advanced in the types of GUIs that you are trying to construct, or if you just want to understand more about how the components work, you also want to look at java.awt.event, java.awt.image, and java.awt.datatransfer.

Components and Their Function

Use this list as a reference for learning about the various components and what they do.

Note: Layout classes (for positioning components in a window) are covered in detail in the next section, and are not listed here.

Button You can click on the mouse to initiate an event. It can display a label.

Checkbox Can either be on (true) or off (false). The state is changed by clicking on it with the mouse. It can display a label.

CheckboxGroup
Groups several check boxes together and allows selection of one at a time.

CheckboxMenuItem
A check box that can be placed in a menu. It has an on or off state.

Choice Pops up a predefined list of strings for selection, of which *one* can be selected. The selected string is displayed.

Container Holds other components. The order in which the components are added to the container may affect where they are displayed.

Cursor Class that encapsulates bitmap representations for the mouse cursor.

Dialog A window that takes input from the application user.

FileDialog A dialog window that lets the user select a file.

Frame A top-level window that contains the window title.

Label Places text in a container. This can be changed by the application, but not by the application user.

List Scrolling list of text items. One or multiple text items can be selected.

Menu A pull-down component of a menu bar that presents the user with some possible selections.

MenuBar A selection of menus placed in a row across the top of a frame that are available to the user.

MenuItem Individual items in the menu. This can be a simple label, check box, or another menu.

MenuShortcut
Represents keyboard accelerator for an item in the menu.

Panel Most basic container for holding other components.

TextArea A multi-line scrollable area for text entry and viewing.

TextField A single line entry for text entry and viewing.

11.2.1.1 Layout Managers

Most developers have not had exposure to layout managers in other environments, so we want to spend more time explaining them.

Layout managers are a set of classes in the Java AWT that control the placement of the GUI components on a window and recompute the layout when the window is re-sized.

You use layout managers in Java to solve one of the problems that GUI developers have had in the past. How do I have one set of source code that looks "right" on multiple platforms? Development environments generally use the screen x,y coordinates to place the GUI components on the window. This works fine as long as the application is used on only one platform (that is, Windows). If you want to use the same GUI code on multiple platforms (that is, Windows, OS/2, AIX, and so on), the code may work, but it probably does not look right. Text fields may shrink or enlarge, controls can overlap, and spacing on the window might not be correct. When developing in Java, the development environments also allow you to create GUIs based on x,y coordinates. However, you can also use layout managers that use relative positioning of the GUI controls, and have portable GUI code.

Layout managers also assist the GUI developer by providing functions, such as re-sizing of windows built into the code. So, should you always use a layout manager? Not if you are absolutely sure that the GUI you are coding is only run on one platform for the duration of its use, and that it is a window that does not need to be re-sized. Otherwise, it is safer to use a layout manager. However, it is often easier to initially construct your GUI using x,y coordinates to place the controls as you iterate through your design to get the look of your GUI correct. Then, when you are satisfied that you have the GUI look correct and the users are happy with the design, it can be restructured using the correct layout managers.

Note: The IDE that you use may generate the layout manager code for you based on input to a property editor or through a wizard. You still need to understand what is occurring, though, to take full advantage of the tools that may help construct the windows using the layout managers.

Layout Managers Provided by AWT

AWT provides a set of layout managers for your use. You can also create your own layout manager, but that is not a topic that we explain in this introductory material. The available managers and a short description are given here. When using layout managers, the order in which you add the components may make a difference.

Flow Layout

Flow layout is the most basic layout. As components are added, they are placed in a single row, left to right, with the row centered by default on the window. If more components are added than fit in a row, it wraps, starting a new row directly below the first and continuing the left to right placement. A common use of this manager is for a row of buttons.

Border Layout

Border layouts use compass directions (north, east, south, west) to position the components. The components are added to the panel around the edges based on the compass direction given and take up the space they need. There is also a center position, and if a component is placed there, it occupies any remaining space.

Grid Layout

Grid layouts allow the developer to divide the window into rows and columns to create cells. Then, one component is placed into each cell starting in the upper left cell and filling out the top row of cells before moving to the next row.

Card Layout

Using a card layout is similar to creating a stack of 3x5 index cards that you can cycle through. Normally, you use the card layout to add a certain number of windows that overlay each other. On each of these windows, another layout manager is used for placing other GUI components.

Using Border, Flow, and Grid Layouts Example

Before seeing the final layout manager, look at this example that uses border, flow, and grid layouts, as shown in Figure 229 on page 295. You can use a combination of these layout managers to achieve the results that you want. Here is an example of a window that was designed to be a Product Order window. The first image shows the completed window.

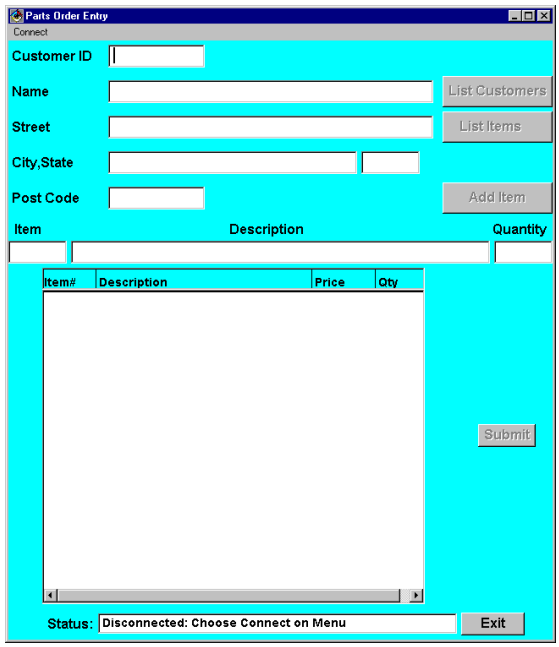


Figure 229. Product Order Window

Due to the number of components on this window, several panels nested inside one another were used to achieve the desired results. Figure 230 shows how the panels were nested and which layout manager was used with each panel. Three panels, which use grid layout managers, were used instead of one large panel, which uses a grid layout manager in the top panel. Therefore, the text fields are larger than the labels or the buttons.

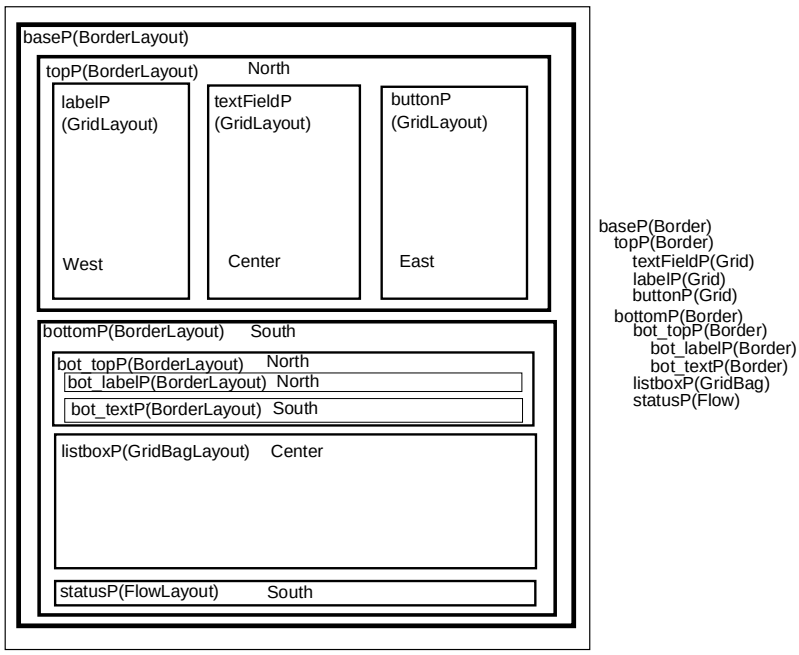


Figure 230. Main Order Entry Panels

Then, the components are placed inside these panels. The placement of the components are controlled by the layout manager. If we take a look at one panel as an example, we can see some of the code that is required. These are just code snippets and do not represent all of the necessary code for creating the window. The panel that we use, as an example, is the labelP panel that holds the labels of Customer ID, Name, Street, City, State, and Post Code. In the code samples, we see the layout managers being set for the panels, the panels getting nested, and the label components being added.

```
 if (ivjtopP == null) {
try {
ivjtopP = new java.awt.Panel();
ivjtopP.setName("topP");
ivjtopP.setLayout(new java.awt.BorderLayout());
gettopP().add(getlabelP(), "West");
gettopP().add(gettextFieldP(), "Center");
gettopP().add(getbuttonP(), "East");
// user code begin {1}
// user code end
} catch (java.lang.Throwable ivjExc) {
// user code begin {2}
// user code end
handleException(ivjExc);
}
 };
 if (ivjlabelP == null) {
try {
ivjlabelP = new java.awt.Panel();
ivjlabelP.setName("labelP");
ivjlabelP.setLayout(getlabelPGridLayout());
getlabelP().add(getLabel1(), getLabel1().getName());
getlabelP().add(getLabel2(), getLabel2().getName());
getlabelP().add(getLabel3(), getLabel3().getName());
getlabelP().add(getLabel4(), getLabel4().getName());
getlabelP().add(getLabel5(), getLabel5().getName());
} catch (java.lang.Throwable ivjExc) {
/handleException(ivjExc);
}
 };
```

Figure 231. topP Panel BorderLayout

Now, we move on to the last layout manager that is provided by the AWT.

Gridbag Layout

The Gridbag Layout is the most powerful, but also the most complex of the provided layout managers. Similar to the Grid layout, the GridBag Layout also supports a rectangular grid of cells. The difference is that the GridBag layout supports components of different sizes. This is accomplished through the use of an additional class called the GridBagConstraints that contains information such as where to place the component, how much space it occupies, how to re-size, and how much space to leave around a component. There are instance variables for GridBagConstraints that provide this control. Some of the values that these variables can be assigned are held by constants on the class. An example of using the fill instance variable if gbc is my instance of a GridBagConstraints may be:

```
gbc.fill = GridBagConstraints.NONE;
```

Here is a listing of the instance variables and their purpose:

Gridx, gridy The layout manager places the component on the grid; the upper left cell is 0,0. The default value is RELATIVE (next location to the last component put into the container).

Gridwidth, gridheight
Specifies the size of the component in number of cells. The default value is 1,1.

Weightx, weighty
Specifies how the layout manager should distribute space if the container expands. Values range from 0.0 - 1.0.

Anchor Tells where the component should be placed if the component is smaller than the area in which it is to be displayed. The values are:

- CENTER (default)
- NORTH
- NORTHEAST
- EAST
- SOUTHEAST
- SOUTH
- SOUTHWEST
- WEST
- NORTHWEST

Fill Lets the layout manager know how to re-size components when the display area is larger than the component. The values are:

- NONE — Does not fill the area.
- HORIZONTAL — Fills the area horizontally.
- VERTICAL — Fills the area vertically.
- BOTH — Fills the area in both directions.

Insets Minimum border between the component and the display area in which it is located. This must be an instance of the class Insets. Normally, the constructor Insets (ini, int, int, int) is used as the value. The int values are top, left, bottom, and right.

Ipadx, ipady Enlarges the minimum size of the component by adding the number of pixels specified to each side (left and right for x, top and bottom for y).

Each component can have an associated GridBagConstraint. As you can see, this gives you a great deal of control in placing components, but it requires some thought and practice to become skilled with this layout manager. Again, depending on the IDE that you use, this code may be generated for you through the use of a wizard or property editor. However, we show you an example using just the classes that the AWT provides. While the IDE makes this easier, they may do it with the use of their own classes. This does not affect the portability of the application, but can make the sharing of code more difficult if all of the developers in your company are not using the same IDE. The example in Figure 232 on page 298 shows the use of the Gridbag Layout.

Figure 232. Select Items (Gridbag Layout)

In the example shown in Figure 232, the frame contains a base panel that holds the three controls (the multi-column list box, the OK button and the cancel button). The base panel (baseP) has a GridBagLayout as a layout manager. The three components have a GridBagConstraints associated with each of them. In the code snippet in Figure 233 on page 299, you can see the settings of the GridBagConstraints for the components.

```
 ivjPanel1.setLayout(new java.awt.GridBagLayout());
constraintsIMulticolumnListbox1.gridx = 0;
constraintsIMulticolumnListbox1.gridy = 0;
constraintsIMulticolumnListbox1.gridwidth = 7;
constraintsIMulticolumnListbox1.gridheight = 4;
constraintsIMulticolumnListbox1.fill = java.awt.GridBagConstraints.BOTH;
constraintsIMulticolumnListbox1.anchor = java.awt.GridBagConstraints.NORTHWEST;
constraintsIMulticolumnListbox1.weightx = 1.0;
constraintsIMulticolumnListbox1.weighty = 1.0;
constraintsIMulticolumnListbox1.insets = new java.awt.Insets(5, 5, 5, 5);

getPanel1().add(getIMulticolumnListbox1(), constraintsIMulticolumnListbox1);
constraintsButton1.gridx = 1;
constraintsButton1.gridy = 4;
constraintsButton1.gridwidth = 1;
constraintsButton1.gridheight = 1;
constraintsButton1.anchor = java.awt.GridBagConstraints.SOUTHWEST;
constraintsButton1.weightx = 0.0;
constraintsButton1.weighty = 0.0;
constraintsButton1.insets = new java.awt.Insets(5, 5, 5, 5);

getPanel1().add(getButton1(), constraintsButton1);
constraintsButton2.gridx = 6;
constraintsButton2.gridy = 4;
constraintsButton2.gridwidth = 1;
constraintsButton2.gridheight = 1;
constraintsButton2.anchor = java.awt.GridBagConstraints.SOUTHEAST;
constraintsButton2.weightx = 0.0;
constraintsButton2.weighty = 0.0;
constraintsButton2.insets = new java.awt.Insets(5, 5, 5, 5);
getPanel1().add(getButton2(), constraintsButton2);
```

Figure 233. Select Item GridBag Layout

11.2.2 Java Foundation Classes (JFC), JavaBeans, and the JDK 1.1 Event Model

The introduction of the AWT in JDK 1.0 was just a first step. It allowed programmers to begin writing user interfaces for their Java applications, particularly Web based applets. However, it was apparent that a more robust package was needed to create the types of GUIs that were already being created in other development environments. Sun Microsystems, Inc., Netscape, and IBM joined forces to create the Java Foundation Classes (JFC), which were built on top of the AWT and integrated Netscape Internet Foundation Classes and the best of new Sun Microsystems, Inc. technologies. Several new advances for the development of GUIs debuted with the release of JDK 1.1, such as:

- **A lightweight UI framework to replace the peer model** — Allows GUIs to be completely written in Java, and not have the overhead of wrapping a native "peer" component.
- **A new event model** — Discussed in a later section.
- **JavaBeans compliance** — Discussed in a later section.
- **Printing support** — In JDK 1.0, printing support needed to be written in the native platform. Now in JDK1.1, the printing of text and graphics can be written in Java to keep applications portable.
- **Data transfer/clipboard support** — Capabilities to transfer objects within and across applications.

- **Desktop colors integration** — Supports desktop color schemes to enable the Java application to match the native platforms color changes.
- **Graphics and image enhancements** — Added capabilities to clip, subset, and flip the graphics image to support applications requiring more advanced graphics capabilities such as engineering applications.
- **Mouseless operation** — Allows navigation through the application through use of the keyboard only if desired by the user. This fills a need for specific users such as data entry and allows short cuts to be incorporated for power users.
- **PopupMenu** — Menu that can be dynamically opened at a specified position in a container.
- **ScrollPane container** — In JDK 1.0, the developer had to manage any scrolling needed to view additional items in a component. The ScrollPane was provided in JDK 1.1 to relieve this burden from the developer and provide better performance and consistency across platforms.

11.2.2.1 The Delegation Event Model

In JDK 1.1, a new event model, called the Delegation Event Model, was introduced. This new model aids in the development of distributed applications and matches well with the visual development environment that is used in the major IDEs. In this model, an event "source" sends an event to a "listener" or a group of listeners. The source object accomplishes this by starting a method on the listening object. This event model allows only the interested objects to have to deal with the event instead of all of the objects in a hierarchy, such as the original model.

For this communication to happen, both the event source object and the listener object must implement some behavior. For an object to become an event source, it must include the ability for another object to register with the source object to receive the event when it occurs. To have this behavior, the source object needs to implement methods to enable either a single object to register if an event is single-cast, or multiple objects to register if the event is multi-cast. If an object wants to register as a listener, it needs to implement the event listener interface of the specific event type. To enable this, especially in IDEs where a lot of this code can be generated for the developer by the IDE, some other concepts are used. Inner classes were added in JDK 1.1. This allows one class to be encapsulated within the scope of another class. Since the scope is shared, it allows the inner class access to the containing classes data and functions. This allows an IDE to create adapter classes for the specific events. We start by showing an example of how a class can implement a listener interface.

Remember, if an object wants to register as a listener, it needs to implement the event listener interface of the *specific* event type. An example of a specific event type is an ActionEvent. This is the event that occurs when the application user clicks the mouse on a button, selects an option on a menu bar, or if the user double-clicks an item in a list. In this case, if the object wanted to register as a listener for the ActionEvent, that object has to implement the ActionEventListener interface. By implementing this interface, the object can use the source objects addActionListener() method to register as a listener. A listing of the interfaces, such as ActionEventListener, can be found in the java.awt.event package.

Similar to the layout managers we discussed, an IDE can generate much of this code for you. To get a better understanding of the code that will appear in your class, take a look at the example shown in Figure 234. Here is an example of the SltItemWdw class that implements the ActionListener interface. We use the Action event to control what happens when the user clicks on the OK button or the cancel button.

In the initConnections method, we add actionListeners for both buttons. If an ActionEvent is generated, the actionPerformed method is called with the event as an input parameter. We use the getSource method of the event to determine the source of the event. We then add the application specific code that we want to execute.

```
public class SltItemWdw extends java.awt.Frame implements java.awt.event.ActionListener,
java.awt.event.ItemListener, java.awt.event.WindowListener, java.beans.PropertyChangeListener {

private void initConnections() {
/getB_OK().addActionListener(this);
getB_Cancel().addActionListener(this);
}

public void actionPerformed(java.awt.event.ActionEvent e) {
if ((e.getSource() == getB_OK()) ) {
// code to execute when the OK button is clicked}
if ((e.getSource() == getB_Cancel()) ) {
// code to execure when the cancel button is clicked}
}
```

Figure 234. SltItemWdw Class

Next, we show an example of the methods that enable an object to become an event source. The example is for the AWT multi-column list box in the SltItemWdw class. It allows the object to register to receive notification when something has happened to an item in the list (that is, an item is selected). In Figure 235, we add a ItemListener for the list box.

```
getIMulticolumnListbox1().addItemListener(this);
```

Figure 235. Adding an ItemListener

Figure 236 shows the method where you put the code to perform when the action occurs. This method resides in the SltItemWdw class. The code was generated by an IDE. The developer only has to add code where the comments are in this method.

```
public void itemStateChanged(java.awt.event.ItemEvent e) {
/if ((e.getSource() == getIMulticolumnListbox1()) ) {
// code to handle event
}
```

Figure 236. Handling the itemStateChanged Event

If you want more details on events and the JDK 1.1 event model, look in the JavaBeans book referenced at the end of the chapter. You can also see this code in the SltItemWdw class which is available for download from the redbook Web site.

11.2.2.2 JavaBeans

The purpose of the JavaBeans specification is to provide a model for Java classes to follow so that the classes can have a predictable way to interact with each other and be easily used in an IDE. What that means is that for your Java class to be considered a bean, it must follow the JavaBean design patterns, or provide another class called a BeanInfo object that can be used to determine the important information (property and event information) about a class. Properties are basically the attributes of an object, and events are signals that indicate that something has happened to the object. JavaBean patterns provide:

- **Properties**
  - Describes access methods for the property so they follow a signature pattern.
  - Allows a property editor to be used in builder tools.
  - Allows access to its object type.
  - Indicates whether the source object can change the property value or must ask permission from interested (registered) objects.
- **Events**
  - How listeners register with the source.
  - What type of listener this event is fired to.
  - What method calls are made to the listener when the event fires.
  - Whether the source event is unicast or multicast.
  - The Java Beans book referenced at the end of the chapter provides much more detail on Java Beans and their development.

11.2.3 New Additions to JFC

The newest release of the JFC again builds on the foundation of previous releases. While JDK 1.1 strengthened the architecture for building Java GUIs with JavaBeans and the delegation event model, the newest additions greatly improve the functionality that is available to the Java developer. These added capabilities bring the Java environment closer to maturity from a GUI perspective and greatly add to the developer's ability to deliver commercial grade applications. These capabilities are being phased in as they become available. For example, at the time of this writing, you can get many of the new components in the latest JDK 1.1.x version. However, the drag-and-drop capabilities are not available yet. An overview of the newest features are shown here:

- **Drag-and-Drop** — Allows the user to drag-and-drop objects from within an application and between applications. This includes Java and non-Java applications.
- **2D API** — Provides new graphical capabilities such as more complex shapes and better control of the graphics rendering process. This provides better performing and more sophisticated graphics in the application.

- **New components** —Allow the creation of more professional and higher quality applications without the developer having to create their own controls or purchasing third-party components. These include:
  - Tree view
  - List view
  - Table view
  - Toolbar
  - Pane splitter
  - Tabbed folder
  - Multi-column List
  - Progress bar
  - Slider
  - Styled text
  - Font chooser
  - Color chooser
  - File chooser
  - Custom cursors
  - Tool tips
  - Image support for buttons and menuItems
  - Status bar
  - Spin box
  - Combo box
  - Drop down combo box
  - Drop down list box
  - Composable button
  - Multimedia controls
- **Pluggable look and feel** — Provides a mechanism to individualize an application's visual look to allow you to select a look and feel with which you are comfortable.
- **Accessibility features for the physically challenged** — Provides a screen reader API to facilitate text to speech or a Braille terminal, and a screen magnifier API to allow you to adjust screen magnification up to 16 times the normal size.

11.3 Summary

Hopefully, this information gives you an awareness of what Java has in terms of capabilities for creating GUIs and some of the issues that need to be addressed when designing GUIs. The next step is to look at some of the resources available (a starting place is the following section), and try creating some prototype GUIs of your own.

11.4 Information Sources

Please consult the following sources for additional information:

- **Web Sites**
  - General GUI
    - http://usableweb.com/hcivl/
    - http://www.acm.org/sigchi/
    - http://www.ibm.com/IBM/HCI/guidelines/guidelines.html
    - http://www.cooper.com/
  - Java
    - http://www.ibm.com/java
    - http://www.alphaWorks.ibm.com - Java-related downloads
    - http://www.javasoft.com
- **Books**
  - General GUI
    - Cooper, Alan. *About Face*. Foster City, CA: IDG Books Worldwide.
    - Preece, Jenny. *Human Computer Interaction*. Reading, Massachusetts: Addison-Wesley, 1994.
    - Mayhew, Deborah J. *Principles and Guidelines in Software User Interface Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.
    - Galitz, Wilbert O. *User-Interface Screen Design*. New York, New York: Wiley-QED, 1993.
    - *Object-Oriented Interface Design* (IBM Common User Access™ Guidelines). Carmel, IN: Que, 1992.
  - Java
    - Englander, Robert. *Developing JavaBeans*. Sebastopol, CA: O'Reilly Associates, Inc., 1997.
    - Vanderburg, Glenn. *Tricks of the Java Programming Gurus*. Indianapolis, IN: Sams.net Publishing, 1996.

Chapter 12. Debugging Java Programs on the AS/400 System

In this chapter, we cover debugging Java programs that run on the AS/400 system. There are two different ways that you can debug Java programs on the AS/400 system.

- You can use the OS/400 system debugger, which is described in Section 12.2, “Using the OS/400 System Debugger” on page 306.
- You can use the Cooperative Debugger that comes as a part of VisualAge for Java - AS/400 Feature, which is described in Section 12.3, “Using the VisualAge for Java Cooperative Debugger” on page 314.

We explain how to use both debug environments in this chapter. Both of these debuggers allow you to debug Java programs that run only on the AS/400 system. You cannot debug client or workstation Java programs with these debuggers. If you want to debug an AS/400 Java client/server program, use the VisualAge for Java debugger to work with the client code and one of the debuggers that are described in this chapter to work with the AS/400 Java code.

12.1 Getting Ready to Debug

To debug Java programs on the AS/400 system, 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 debugging a Java program on the AS/400 system, you need to use the **javac** command to compile the .java file with the -g option to get the debug information into the code, for example:

```
javac -g MyPackage\MyClass.java
```

If you do not compile with the -g option, you cannot debug on the AS/400 system. Debugging with the Cooperative Debugger is limited.

You may receive the message when starting the OS/400 debugger: `(Source not available.) Class file cannot be debugged.` Or, you may receive a message similar to the one shown in Figure 237 from the Cooperative Debugger. If you receive this message, you probably have not compiled your class with the -g option. Compile it and try again.

Figure 237. Debugger Message

12.1.2 Setting the Optimization Level

You may have already compiled the program with the -g option and then used the CRTJVAPGM command or the Compile AS/400 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. Or, delete the program on the AS/400 system using the DLTJVAPGM command. If you use the DLTJVAPGM command, it erases the hidden compiled version of the Java program. When you run the program, a version with optimization level 10 is automatically generated.

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 JDBCRmi package:

```
JAVA CLASS(JDBCRmi.JDBCRmi) OPTION(*DEBUG)
```

The Display Module Source display is shown in Figure 238.

Important

To debug a Java program on the AS/400 system, 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 Chapter 2, “Java Overview and AS/400 Implementation” on page 5.

```
Class file name:   JDBCRmi.JDBCRmi
     1
     2  package JDBCRmi;
     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 keys
```

Figure 238. 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 this example, we set a breakpoint at line 106.

```
                          Display Module Source

 Class file name:   JDBCRmi.JDBCRmi
106  Item theItem = new Item(anItem);
    107
    108   try
    109   {
    110    System.out.println("getItem: Started processing");
    111    java.sql.ResultSet rs = null;
    112
    113
    114
    115   psSingleRecord.setInt(1, Integer.parseInt(anItem));
    116   rs = psSingleRecord.executeQuery();
    117
    118   if (rs.next()) {
    119    theItem.setItemDesc(rs.getString("PARTDS"));
    120    theItem.setItemQty(rs.getInt("PARTQY"));

 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 106.
```

Figure 239. Setting a Breakpoint

When we start the program, it stops at line 106 before running it (Figure 240 on page 308).

```
                          Display Module Source
 Current thread:   000000EA     Stopped thread:   000000EA
 Class file name:   JDBCRmi.JDBCRmi
    102   */
    103  public Item getItem (String anItem) throws RemoteException
    104  //public String getItem (Item anItem) throws RemoteException
    105  {
106  Item theItem = new Item(anItem);
    107
    108   try
    109   {
    110    System.out.println("getItem: Started processing");
    111    java.sql.ResultSet rs = null;
    112
    113
    114
    115   psSingleRecord.setInt(1, Integer.parseInt(anItem));
    116   rs = psSingleRecord.executeQuery();

 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 106 in thread 000000EA
```

Figure 240. Stopping at a Breakpoint

12.2.2 Displaying Variables

The debugger allows you to display program variables. To display a variable, you enter EVAL VARIABLENAME on the command line. Figure 241 shows how to evaluate a String variable called anItem.

```
                          Display Module Source
 Current thread:   000000EA     Stopped thread:   000000EA
 Class file name:   JDBCRmi.JDBCRmi
    102   */
    103  public Item getItem (String anItem) throws RemoteException
    104  //public String getItem (Item anItem) throws RemoteException
    105  {
    106  Item theItem = new Item(anItem);
    107
    108   try
    109   {
    110    System.out.println("getItem: Started processing");
    111    java.sql.ResultSet rs = null;
    112
    113
    114
    115   psSingleRecord.setInt(1, Integer.parseInt(anItem));
    116   rs = psSingleRecord.executeQuery();

 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 = 12301
```

Figure 241. 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 242, we evaluate an object called theItem.

```
                          Display Module Source
Current thread:   000000EA     Stopped thread:   000000EA
Class file name:   JDBCRmi.JDBCRmi
   114
   115   psSingleRecord.setInt(1, Integer.parseInt(anItem));
   116   rs = psSingleRecord.executeQuery();
   117
   118   if (rs.next()) {
   119    theItem.setItemDesc(rs.getString("PARTDS"));
   120    theItem.setItemQty(rs.getInt("PARTQY"));
   121    theItem.setItemDate(rs.getDate("PARTDT").toString());
   122    theItem.setItemPrice(rs.getBigDecimal("PARTPR", 2));
   123
   124   }
   125   else {
   126
   127       return(null);
   128  }

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 242. Using the EVAL Function on an Object

Figure 243 shows the object Evaluate Expression display.

```
                              Evaluate Expression

Previous debug expressions

  ----> THREAD 000000E2 <----
  ----> THREAD 000000EA <----
> EVAL theItem
  theItem = JDBCRmi/Item:ECE12F34EB0DB700
  theItem.ItemId = java/lang/StringBuffer:ECE12F34EB0DB980
  theItem.ItemDesc = Quad speed CD ROM Drive
  theItem.ItemPrice = java/math/BigDecimal:ECE12F34EB056080
  theItem.ItemDate = 1996-01-12
  theItem.entryArray = ARR:ECE12F34EB289580
  theItem.ItemQuantity = 44
  theItem.index = -1
  theItem.cursor = 0

Debug . . .


F3=Exit   F9=Retrieve   F12=Cancel   F16=Repeat find   F19=Left   F20=Right
F21=Command entry       F23=Display output
```

Figure 243. Evaluate Expression for an Object

You can use the EVAL function to set variables or to test objects for equality. Here we show 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, the same as if we were writing Java code.

```
                              Evaluate Expression

Previous debug expressions

  theItem.ItemPrice = null
  theItem.ItemDate = 1996-01-12
  theItem.entryArray = ARR:C96F2B19C70BE880
  theItem.ItemQuantity = 44
  theItem.index = -1
  theItem.cursor = 0
> EVAL theItem.ItemQuantity == 43
  theItem.ItemQuantity == 43 = FALSE
> EVAL theItem.ItemQuantity == 44
  theItem.ItemQuantity == 44 = TRUE
> EVAL theItem.ItemQuantity = 43
  theItem.ItemQuantity = 43 = 43
> EVAL theItem.ItemQuantity == 43
  theItem.ItemQuantity == 43 = TRUE
EVAL theItem
theItem = JDBCRmi/Item:C96F2B19C70BE680
theItem.ItemId = java/lang/StringBuffer:C96F2B19C70BE700
theItem.ItemDesc = Quad speed CD ROM Drive
theItem.ItemPrice = null
theItem.ItemDate = 1996-01-12
theItem.entryArray = ARR:C96F2B19C70BE880
theItem.ItemQuantity = 43
theItem.index = -1
theItem.cursor = 0

Debug . . .


F3=Exit   F9=Retrieve   F12=Cancel   F16=Repeat find   F19=Left   F20=Right
F21=Command entry       F23=Display output
```

Figure 244. 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 245 on page 312.

```
                          Work with Module List
                                                                 System:
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 245. Work with Module List

12.2.4 Debugging from Another Terminal Session

You may choose to debug the Java program from an AS/400 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 get into its source with the debugger and set the appropriate breakpoints that you want.
2. Sign on to an AS/400 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 of debugging a Java program on AS/400 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 job QJVACMDSRV for user MAATTA. We use option 5 to work with the job (Figure 246 on page 313).

```
                              Work with Active Jobs
                                                                  03/17/98
 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
        DSP04        WBL         INT      .0  PGM-QCMD         DSPW
5       QJVACMDSRV   MAATTA      BCI      .0                   TIMW
        QPADEV0007   MAATTA      INT      .0  CMD-JAVA         DEQW
        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                   TIMW
      QSERVER        QSYS        SBS      .0                   DEQW
        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 246. Work with Active Jobs

Then, the Work with Job display (Figure 247) is shown. 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 247. Work with Job

Use the STRSRVJOB command prompt display to start a service job for the Java program. See Figure 248 on page 314.

```
                        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=Cance
F13=How to use this display        F24=More keys
```

Figure 248. 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 this 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 249.

```
                            Display Module Source

 Class file name:   JDBCRmi.JDBCRmi
      1
      2  package JDBCRmi;
      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 variab
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More key
```

Figure 249. Display Module Source

12.3 Using the VisualAge for Java Cooperative Debugger

VisualAge for Java Enterprise Edition includes a cooperative debugger. The cooperative debugger allows you to debug a Java program running on the AS/400 system from the VisualAge for Java IDE running on a workstation. Before you can use the debugger, the debug server must be started on the AS/400

system. To start the debug server, enter the Start Debug Server (STRDBGSVR) command on an AS/400 command line and press **Enter**.

> **Attention**
>
> The debug server needs to be started only once for the AS/400 system 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 STRDBGSVR command.

To debug an AS/400 Java program, you need to compile it using the -g option. If you compiled the program and used a higher optimization level higher than 10, recompile with level 10 for best results. Use:

```
CRTJVAPGM CLSF(xxxxxx) OPTIMIZE(10)
```

12.3.1 Debugging an AS/400 Java Program

This section shows how to debug a Java program running on the AS/400 system from within the VisualAge for Java IDE. The program that we debug is the RMI example that we created in Section 7.2, “Order Entry Using JDBC” on page 151. To start the remote debugger, we highlight the program that we want to debug and select Debug from the ET/400 menu (Figure 250 on page 316).

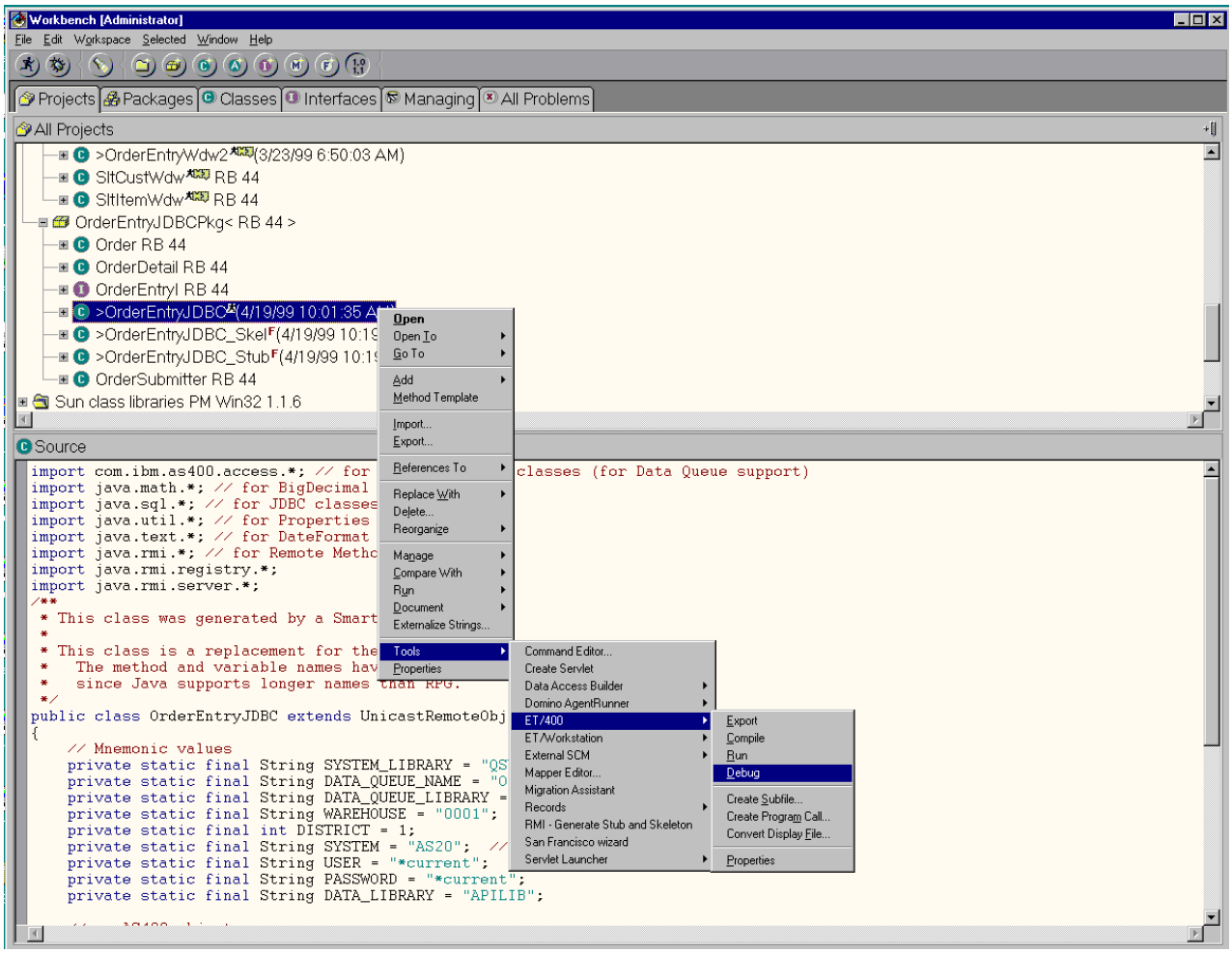


Figure 250. Debugger Start Up

The Debugger Logon prompt shown in Figure 251 appears. Enter your user ID, password, and AS/400 name or IP address of the AS/400 system that you want to debug the Java application on and press **OK**.

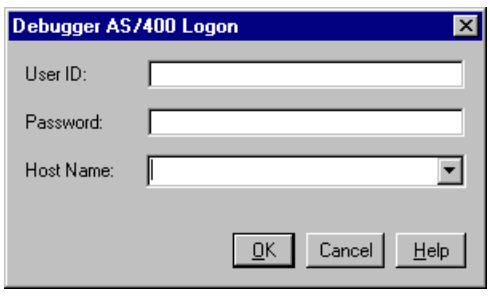


Figure 251. Debugger AS/400 Logon

After signing on the AS/400 system, the debugger starts running the program on the AS/400 system. We see the source Java code for the program as shown in Figure 256 on page 320.

After signing on to the AS/400 system successfully, the Startup information window is shown (Figure 252).

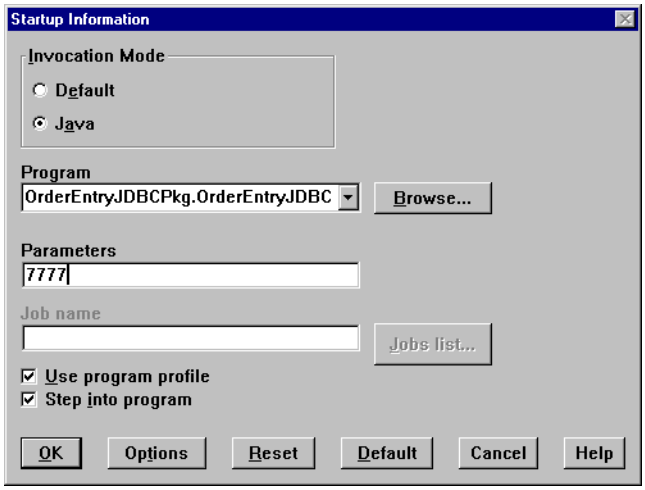


Figure 252. Debug Startup

Select Java as the Invocation method and enter the name of your class in the program entry field. Do not worry about CLASSPATH yet. You must enter the name as PackageName.ClassName as if you are running the application from the command line.

If this is not the first time you are debugging this class, chances are that the name of the class is displayed in the Program Textbox drop-down list. It holds the latest debugged class name.

You can enter any required program parameters. For example, if this program uses the Remote Method Invocation (RMI) interface, you may be using a parameter to enter the TCP port number that is being used.

Next, press the **Options** button to display the Debugger Environment window (Figure 253).

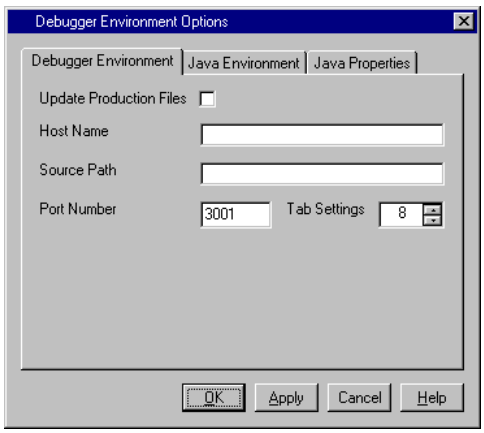


Figure 253. Debugger Environment

As shipped, the AS/400 debug server is set up to listen for connection requests on TCP port 3001. If port 3001 is being used by another application, you can designate and use another port instead. To do this, complete the following steps:

1. Use the Work Server Table Entry (WRKSRVTBLE) command on the AS/400 system and change the port named QDBGSVR to the new value.

   Note: Before changing the port, end the debug server by using the End Debug Server (ENDDBGSVR) command. Then, change the port and start the debug server again.

2. To sign on to the AS/400 debug host for the first time, specify the new port in the Host Name entry field of the Debugger AS/400 Logon dialog as HOSTNAME:PORTNUMBER. Here, HOSTNAME is the name of the AS/400 system on which you want to run the application to be debugged. PORTNUMBER is the new value of the QDBGSVR port.

3. When the Startup Information dialog appears, press the **Options** push button.

4. Change the Port Number setting in the Debugger Environment page of the Debugger Environment Options notebook to match the port specified on the AS/400 WRKSRVTBLE command.

Setting the CLASSPATH

Select the Java Environment tab (Figure 254) to set the CLASSPATH for the Java program.

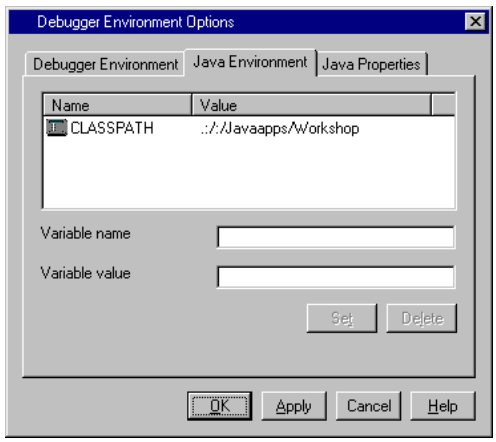


Figure 254. Cooperative Debugger Java Environment

The default is /QJAVA/. If you need to change it, keep in mind that you are debugging on the AS/400 system. Therefore, you need to use forward slashes and colons to separate directories such as this:

```
/mylib:/jt400/lib/jt400.zip
```

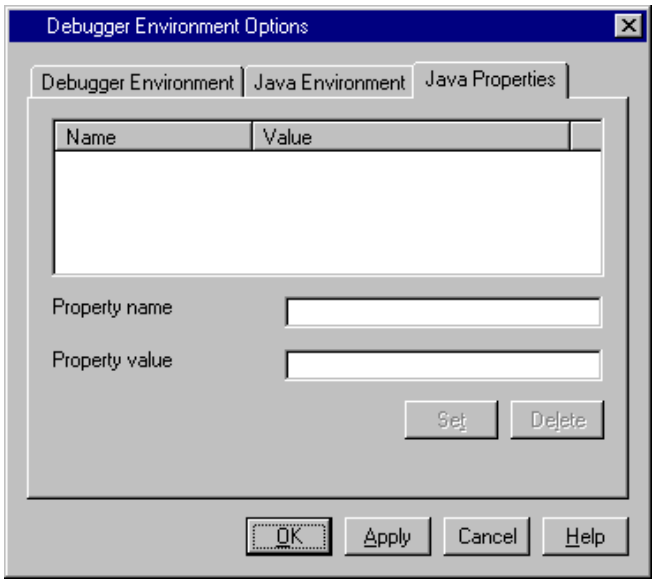


Figure 255. Cooperative Debugger Java Properties

Use the Java Properties tab page (Figure 255) of the Debugger Environment Options notebook to specify property names and values that are passed to the Java Virtual Machine (JVM). These can be retrieved by using a Java program running on the AS/400 system.

Press **OK**. Then, press **OK** again on the Startup Information dialog to start debugging.

12.3.2 Debugging AS/400 Java Programs

The Cooperative Debugger allows you to debug Java programs that run on the AS/400 system. The debugger itself 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. As with the OS/400 debugger, it can only debug Java code running on the AS/400 system. It cannot, for example, debug a Java client/server application at both the client and the AS/400 server.

Figure 256 on page 320 shows the initial debug window.

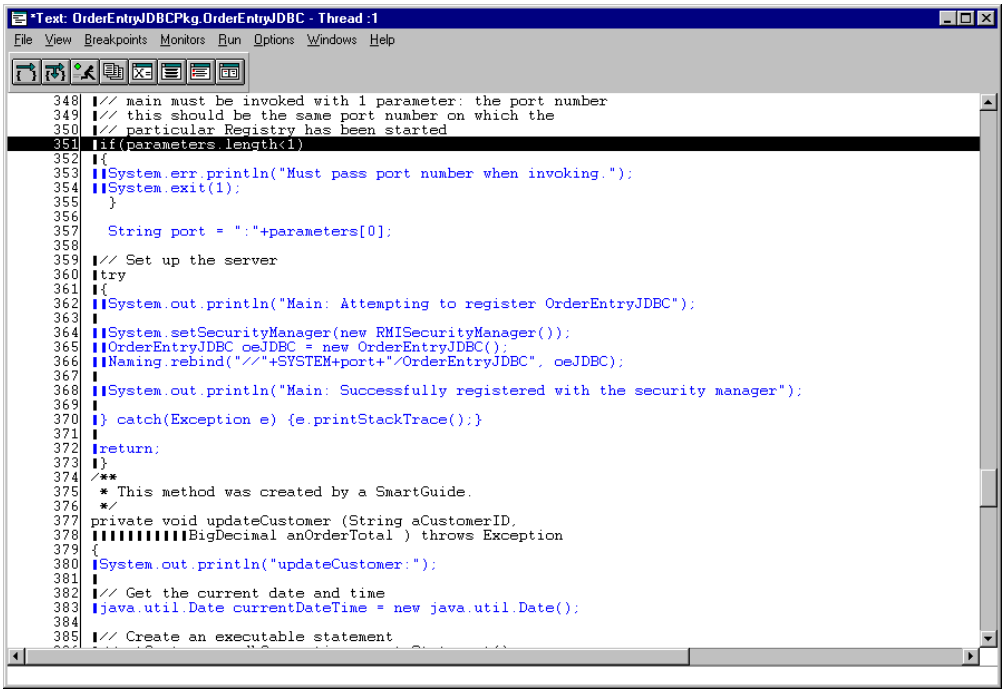


Figure 256. Debug Source Window

You can set breakpoints by double-clicking on the line number where you want the breakpoint.

Start debugging by pressing on one of the step buttons or the run button (or use the keyboard shortcuts O, D, or R). The local variable window changes as you step through the program.

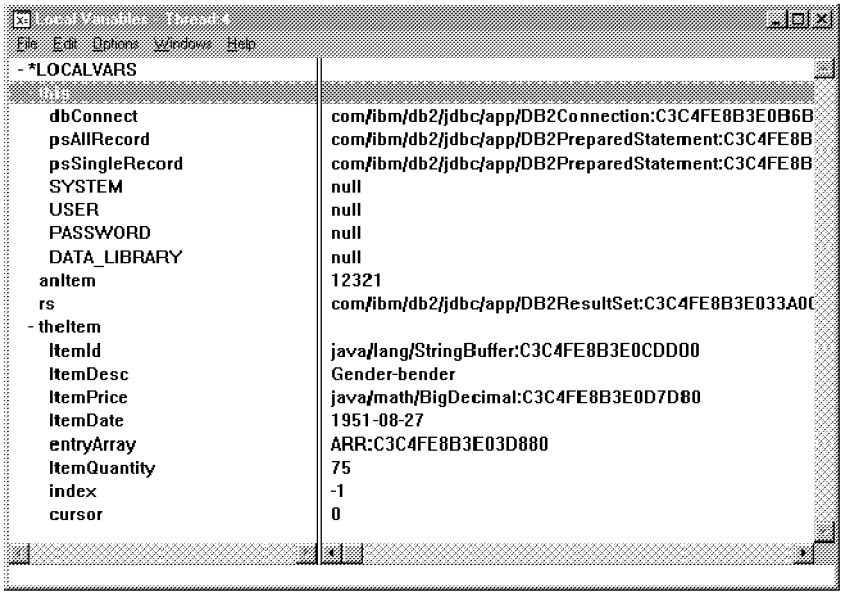


Figure 257. Debugging an Application

If you need to change any of the local variables, simply double-click on the value of the variable to make the change.

12.3.3 Controlling the Cooperative Debugger

You can control a debug session through the use of the Cooperative Debugger icons (Figure 258). These icons cause debugger commands to run. In this section, we describe the control icons and how to use them.

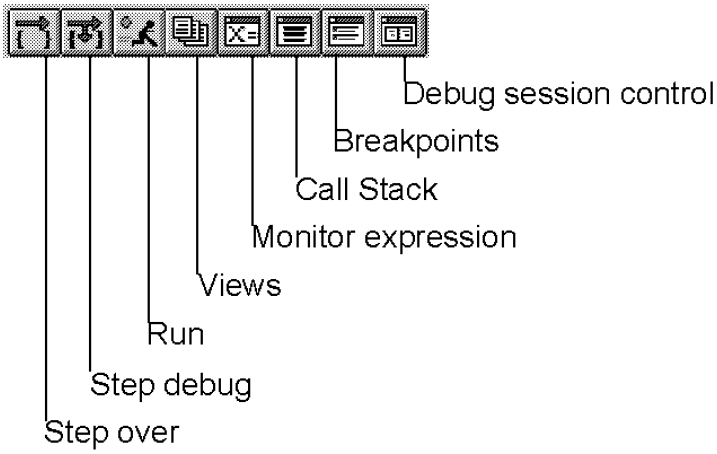


Figure 258. Debugger Control Icons

The Debugger icons include:

- **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 Debug**

  Step Debug 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.

  Note: Because of the more complex analysis that the debugger must do to check for code, which can be debugged during a Step Debug command, the performance of Step Debug may be slower than that of the Step Over command.

- **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, at the beginning of the program. When you select Run, execution continues until one of these conditions occur:

  – The debugger encounters a breakpoint.
  – An exception is encountered that is not handled.
  – The program terminates.
  – You close the debugger.

- **Views**

  Views switches between the Statement View and the Text View. The Text view displays the source code for the program that is being debugged. If this view is not available, it is because the debugger cannot locate the .java file. The Statement View displays the statement number for each executable statement and the name of the procedure to which the statement belongs.

- **Monitor Expression**

  You can use the Monitor Expression dialog to add a variable or expression to one of the following monitors that are listed in the dialog within a radio button group box:

  – *Popup* — Associated with a specific Source window and closes when the associated window closes

  – *Private* — Monitor variables or expressions from a given Source window.

  – *Program* — Collect variables or expressions that you want to monitor. This monitor is not associated with any specific Source window.

- **Call Stack**

  You can use the Call Stack window to view a list of active methods for a thread, and to raise a Source window to view the code for any of these methods.

- **Breakpoints**

  This window lists information on all breakpoints, and lets you enable, disable, set, and delete breakpoints. You can also modify breakpoint characteristics from this window.

- **Debug session control**

  The debug session control switches back and forth to the Debug Session Control window.

Chapter 13. Java Performance and Work Management Overview

Java on the AS/400 system results in a different environment from what most AS/400 developers previously used. First, Java is a truly object-oriented language, unlike RPG and COBOL, which are considered to be procedural languages. Therefore, the development life cycle is also different.

From a work management viewpoint, there are several concepts with which the traditional AS/400 application developer or deployer should be familiar. Among these are the spawned Batch Immediate jobs (within which Java programs run) and built-in kernel thread support, since Java is a multi-threaded language. Several system tuning tips are provided in this chapter, specifically for the new Java environment, which is very different from traditional AS/400 environments.

Within an application, there are several techniques that can be used to improve performance. These include compile options, run parameters, and coding techniques. For the most part, good coding techniques are platform independent, so they apply equally to AS/400 JDK.

Another area that impacts response time and resource utilization is the use of system services, mostly involving database access. JDBC is expected to be the most common database access method because it is easier to switch to a different database server platform. In situations where the database server is always an AS/400, DDM (record level access) and DPC (distributed program call) are viable, high performance options. These techniques for database access are covered in more detail in Chapter 14, “Java Application Design and Host Performance Analysis” on page 355.

The emphasis in this chapter is on Java performance on the AS/400 system. At the time of writing this redbook, significant IBM development resources were being invested to improve Java runtime performance. The information contained in this chapter is based on an early implementation of AS/400 Development Kit for Java (5769-JV1) on Version 4 Release 4 of OS/400.

13.1 Java Implementation

One of the greatest strengths of Java is its portability. An application that adheres to the Java standards will run on any compliant Java Virtual Machine (JVM) on any hardware platform. This portability is achieved through standardized language constructs, where the Java source can be moved from one system to another, and at a lower level through bytecodes.

Bytecodes are machine-independent pseudo-code generated by the Java compiler and executed by the Java interpreter. When a Java source program is compiled using **javac**, a class file containing the bytecodes is created. This class file containing the bytecodes can be executed on the original system's JVM and can also be transported to another platform and executed on a different JVM.

Executing the bytecodes on a JVM involves interpreting the bytecodes into instructions that are specific to that particular operating environment (operating software and hardware). This implies that a JVM is platform-specific because it has to be aware of the underlying system architecture. On the other hand, the

bytecodes are completely portable and isolated from the underlying hardware implementation.

13.1.1 Comparison with Traditional AS/400 Applications

Consider the process that was described in the previous section. The program source was "compiled" into bytecodes, then the bytecodes were "interpreted" by the virtual machine. This is different from other programming models on the AS/400 system. In other languages, "compile" of the program source generates an executable *PGM object under the Original Programming Model. In the Integrated Language Environment (ILE), the program source is compiled into a *MODULE, which is then bound with other modules to create an executable *PGM object.

The AS/400 Developer Kit for Java provides an additional facility to create a persistent, optimized program object as the default. This facility is called the *Java transformer* and is invoked explicitly through the CRTJVAPGM command or automatically when a class file is first loaded. Because the program object consists of 64-bit enabled RISC instructions, runtime performance is improved significantly without affecting the Java portability qualities of the corresponding bytecodes. The traditional ILE environment and the new Java environment are graphically compared in Figure 259.

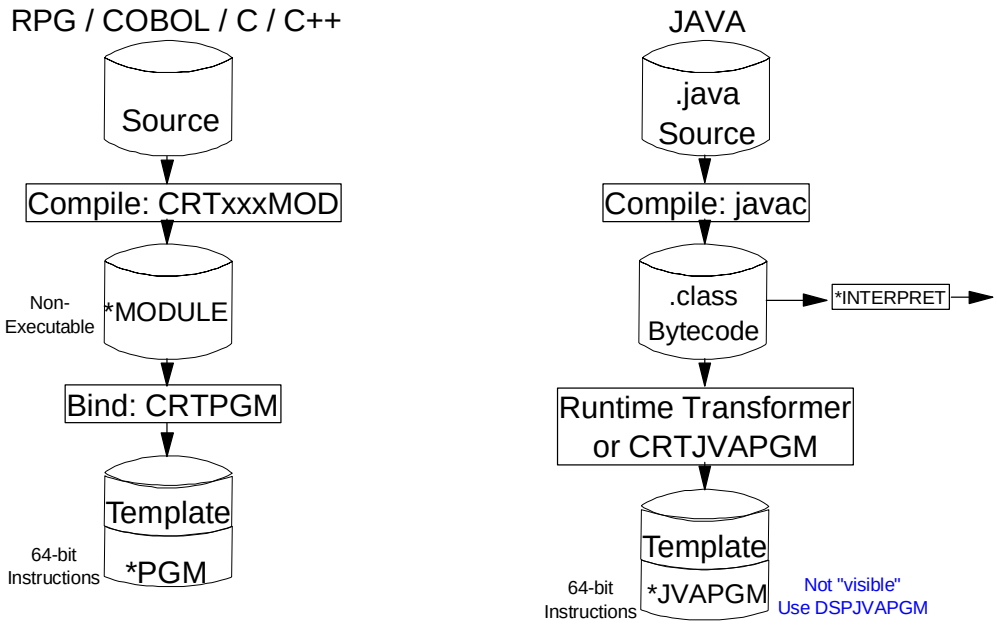


Figure 259. AS/400 Application Development Comparison

13.1.2 The Development Platform versus the Deployment Platform

In the traditional application development environment, an application has to be installed and run on the same platform (hardware and operating system) in which it was developed. However, Java's portability frees the enterprise from such a restriction. An application that is 100% pure Java can run on any compatible JVM.

Most of the Java application development is expected to occur in a platform that provides a graphical user interface (GUI) and can run an Integrated Development

Environment (IDE). Obviously, such a development environment provides large productivity benefits and enables techniques such as Rapid Application Development (RAD). Additional gains can be achieved by having development features that are more integrated with the server, for example, VisualAge for Java and the Enterprise Toolkit for AS/400.

Regardless of where the application is developed, certain components should be deployed where it makes business sense. For example, the mission critical components that handle the business logic and the database serving belong to a scalable, secure, and reliable AS/400 server.

13.2 Industry Concerns about Java Performance

With all of the excitement about Java, there are several concerns regarding its performance. These concerns are not isolated to a specific server's Java implementation, but are valid for all Java incarnations. Often there is a trade-off between performance and many things such as ease of use, portability, levels of abstraction, and so on. Java should not be looked at in isolation, but should be considered along with the goals, objectives, and requirements (current and future) when developing an application system.

13.2.1 Portability and Interpreted Code

From a performance standpoint, these two characteristics of the Java programming model raise red flags. One is the promise of portability. In most, if not all, computing environments, portability is mutually exclusive with performance. The other is the interpreted nature of the bytecodes. When one recalls the inherent slowness of past interpreted languages, there is a tendency to expect the same from Java.

In a way, these two are related. The interpreted bytecodes are needed for Java to be portable. During the creation of the Java class file that contains the bytecodes, the destination system may not be known. Therefore, the compile can only optimize the bytecodes down to the lowest common denominator. It cannot contain any features that are platform specific. The portable bytecodes then may not be able to take advantage of a hardware platform's performance strengths.

As far as the interpreted nature is concerned, interpretation of bytecodes occurs at a much lower level than the source level interpreters with which many programmers are familiar. In addition, the javac compile process also performs a certain level of optimization. It then runs faster than source level interpreted programs, but not as well as those languages that can be compiled to a specific hardware platform (for example, ILE RPG).

In practice, most server Java implementations do not run in interpreted mode. There are large efforts being undertaken within the IT industry to optimize the performance of the portable Java bytecodes by reducing the inherent slowness of interpretation. The following list shows the significant techniques:

- **Just-in-Time Compilers (JIT)**

  This approach involves a monitor that runs separately from the application. It keeps statistics on which methods or statements are being executed most frequently and generates machine instructions to improve their execution. However, the overhead of the JIT compiler itself is very significant. Therefore,

it has a negative impact on the application's performance, particularly in those systems that are not well designed for multi-tasking environments.

Because of the runtime overhead, more sophisticated optimization algorithms cannot be used. Furthermore, the output of the JIT is not persistent and has to be re-created at the next execution.

- **Static Compilation**

  In this approach, Java source or bytecodes are compiled directly into platform specific executable programs. This is done on an application level so that most static compilation schemes cannot handle dynamic class loading, which is necessary for applets and servlets.

  Because of such limitations, most static compilation models cannot pass the Sun compatibility tests. However, because the compilation is done prior to runtime, optimization levels can be higher.

- **AS/400 Transformer**

  The AS/400 system makes use of its Java transformer, which generates 64-bit PowerPC instructions from the bytecodes. This approach retains the bytecodes' portability (allowing them to be served to other platforms) and makes use of an internal Java program to perform the actual execution.

  Since there is an option to run the transformation prior to runtime, more sophisticated optimization algorithms are used. Several optimization levels are available. However, unlike the static compilation approach, dynamic class loading is available with the transformer.

  By default, the transformer is invoked when running a Java application. If a particular class file does not have an associated Java program, one is created for it in the same integrated file system subdirectory. This is known as "implicit transformation."

  However, for operational simplicity and for ultimate runtime performance, we recommend explicit transformation through the CRTJVAPGM command. This is done at application installation or deployment time, rather than relying on the first user to "touch" a class.

13.2.2 The 'Application Layer' Java Virtual Machine (JVM)

A computing platform that wishes to participate in the Java bandwagon needs to provide a JVM that adheres to the specifications. Most of the JVM implementations involve a JVM that runs as an application above the operating system. The Java application, in turn, runs on top of this JVM.

As expected, there can be a significant loss of performance in this type of architecture. For a comparison, note the "application level" JVM that was available for early testing prior to V4R2 of OS/400. At that time, the IBM Hursley lab ported a JVM to run on top of OS/400. Known as a "technology preview," this JVM allowed Java applications to function on the AS/400 system on a test basis. However, the performance was poor.

Consistent with the integrated nature of the AS/400 system, the JVM implementation is fully integrated, all the way down to the SLIC level. This provides very significant performance advantages. Java application portability is not affected. The only disadvantage to this approache is that when a new Java specification becomes available, there may be a short period of time before the

changes can be integrated into the AS/400 system. Such a delay is usually not significant. The reason is that AS/400 customers, who have a heritage of running mission-critical applications, tend to wait for the new Java Development Kit (JDK) technology to mature before implementing it.

13.2.3 Are Object-Oriented (OO) Designs Inherently Slower?

Going beyond the language implementation, a concern that object-oriented (OO) technology pundits have is that an object-oriented design is expected to perform slower than a functionally equivalent structured programming design or even spaghetti code. This is mainly due to method resolution, which occurs when methods from other classes are invoked. The invocation of a method may result in a search for the appropriate method and the invocation of that method.

In a conceptual OO implementation, when a method or procedure from another class needs to be invoked, the method is called and the corresponding parameters are passed to the class. This is part of the overhead of method resolution. There are various ways to reduce this overhead.

Note that Java is a full OO language and is capable of implementing OO concepts. To fully use the capabilities of this language, an application should be developed using the recommended object-oriented analysis (OOA) and object-oriented design (OOD) methodologies. This results in an application that has a well-designed class hierarchy with high levels of maintainability and reusability. But what performance penalty needs to be paid for this?

As in other platforms, there is, indeed, additional overhead associated with an OO design. For example, instantiating an object is relatively expensive. Using a method that resides in another class (for example, the superclass) results in method resolution, which has a higher cost than if the method was in the same class that invoked it. Java also has stricter type checking than a language such as C++.

Method inlining can significantly improve method call performance. This is available for any method that is final (private, static, or protected). To take advantage of this (prior to V4R4), use the javac -O option when creating the Java class from the Java source. One trade-off is that the sizes of the class files, and the sizes of the transformed Java programs that are eventually created, increase. However, the resulting DASD storage penalty is well worth the improvement in runtime performance.

Before discounting OO technology and Java due to the incremental overhead, be aware that the trend in computer science is that software development and maintenance costs are increasing rapidly. In the meantime, hardware price/performance continues to improve.

13.2.4 Early Technology

In comparison with other programming environments that are available on the AS/400 system, Java is fairly new to the platform and to the entire industry. Other languages such as RPG and COBOL have had many years to advance over Java in terms of performance optimization. OS/400 V4R2 contained the first full production version of Java for the AS/400 system. The current V4R4 implementation is significantly faster across the board. In some cases, Java performs as well or even better than other languages. However, there are also

cases when Java is worse than other languages that implement the same function.

The industry's performance objective is to have Java perform similar to C++. This cannot be expected to happen when Java is interpreted. However, with JIT compilers in other platforms and with the AS/400 system's Java transformer, there is a good chance that such an objective can be achieved in the near future (perhaps a few Web years). There has even been talk of processors that are optimized for Java, but that is all for the future.

Java has also drawn interest due to its applicability to the Internet. Java applets can be downloaded from an Internet server such as an AS/400e system to a client workstation or network station where a browser frame is presented to the user. Processing can occur on the client if the browser has a JVM (and most do). The user can then input data and send a response back to another Java program running on the AS/400 system, which can eventually access information stored in DB2/400. If this happens, it fits the profile of a commercial environment, albeit with an electronic commerce flavor.

13.3 Commercial Server Performance Profile

There is a wide range of Java benchmark information that is available on the Web. Most of this information does not apply to the kinds of workloads that run on an AS/400 system. Most of these benchmarks (for example, Towers of Hanoi, CaffeineMark, VolanoMark, and so on) are not commercial server benchmarks. They only provide an indication of how fast a single-user client workstation can display graphical user interfaces (GUI) or perform scientific or engineering calculations or serve up a chat room. This is not the environment for which the AS/400 is designed.

In addition, most of these benchmarks contain single-thread workloads (an exception is Volano, the chat server benchmark, which contains many), and are sometimes so small that they fit entirely into a computer’s memory or even its L2 cache. Therefore, they do not apply to the AS/400 system.

The AS/400 system has inherent architectural advantages in running commercial workloads with a huge number of threads. It is not uncommon to have this platform support thousands of users and jobs. In several comparison tests, the AS/400 system has shown significantly more scalability than other platforms, some of which exhibited significantly diminishing returns after only two threads.

A recently audited and certified benchmark is the Business Object Benchmark for Java (jBOB), which runs a workload based on the business model used by TPC Benchmark C, Standard Specification, Revision 3.3, April 8, 1997. In accordance with the Transaction Processing Council's (TPC) fair use policy, we must note that jBOB deviates from the TPC-C specification and is not comparable to any official TPC result.

jBOB is a benchmark that runs a large number of threads and includes database access. More information regarding the benchmark is available from the AS/400 Web site at: http://www.as400.ibm.com/whpapr/jbob400.htm

You can also perform a search on jBOB.

Traditionally, the AS/400 system has been an excellent performer within the commercial environment. Typical RPG, COBOL, or C commercial applications have performance profiles, where fewer resources are spent processing high-level language program instructions than system services such as database processing (this is represented graphically in Figure 260).

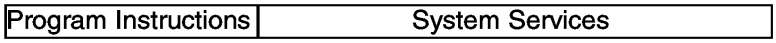


Figure 260. Traditional Commercial Application

A commercial Java application is expected to have a similar profile. However, currently, the Program Instructions portion is proportionately higher. This implies that more time is spent executing machine instructions compared to performing system activities such as database I/Os.

Maximum performance payback is then achieved by implementing similar techniques that worked so well in traditional languages (for example, database tuning, keeping files open, logical blocking, and so on).

13.4 AS/400 Java Execution Steps

The following major events occur when a Java program is run on any platform. While the Java application is portable, there may be different ways of tuning specific platforms to improve performance.

- Starting the Java environment (JVM)
- Searching, verifying, and loading the class files
- Actual program interpretation or instruction execution
- Accessing system services

These events are further explained in Figure 261.

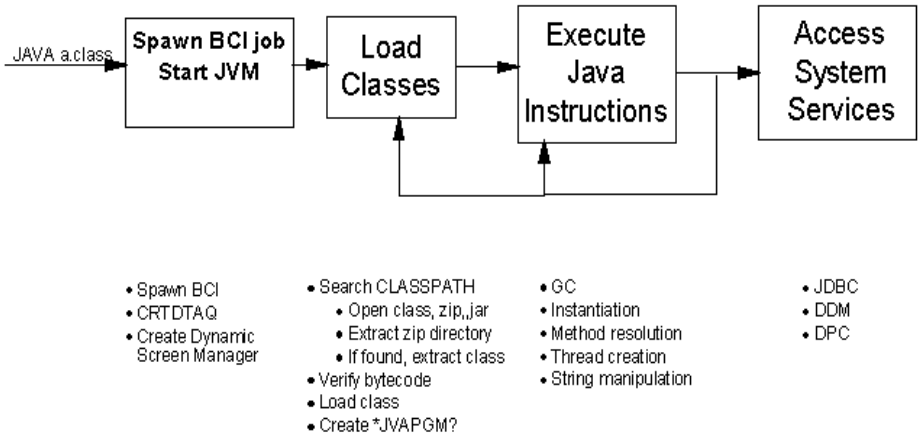


Figure 261. AS/400 Java Execution Steps

In all platforms, there is a noticeable delay when a class is first executed using the JAVA command. This includes time taken to set up the JVM, search through the CLASSPATH for classes, verify the bytecodes, and load the classes.

Therefore, running many short-lived Java programs is not recommended because it may take more time and resources to start up and to terminate than to actually execute the programs.

Upon invocation, the instructions in a Java class are executed. This portion is affected by program coding techniques.

The Java application generally has to access system services to perform functions related to the database, security, and so on. Performance can be affected by techniques such as database tuning, and so on.

These major steps are discussed in more detail later in this chapter.

13.5 Comparison with Main Frame Interactive (MFI)

When discussing performance, it is best to start with the most apparent symptom of a performance problem—slow response time. This term as used here can apply to interactive response time in the case of an online transaction processing (OLTP) environment and also to batch processing. Usually there is a response time objective, either implied or explicitly stated. A performance problem exists if the response time objective is not being met or, based on future projections, the objective will not be met.

Let us start with a typical Main Frame Interactive (MFI) environment, also known as a non-programmable terminal environment. Many AS/400 installations use applications that run in this environment. The application transaction is triggered by the user pressing Enter or a Function key.

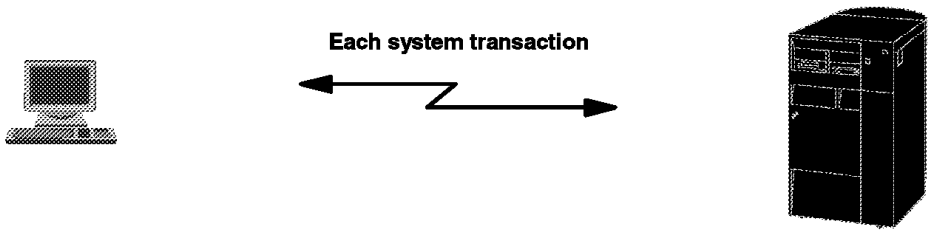


Figure 262. Main Frame Interactive Response

For every transaction, there is usually one conversation between the terminal and the AS/400 system, unless certain 5250 display handling techniques such as RSTDSP(*YES) or DFRWRT(*NO) are used. A large majority of the response times can be attributed to the server AS/400 system with minimal contributions in the network and virtually none by the terminal.

In Figure 263 on page 331, the response time is divided into its individual components.

Note: The actual contribution of each response time component may vary depending on the system environment.

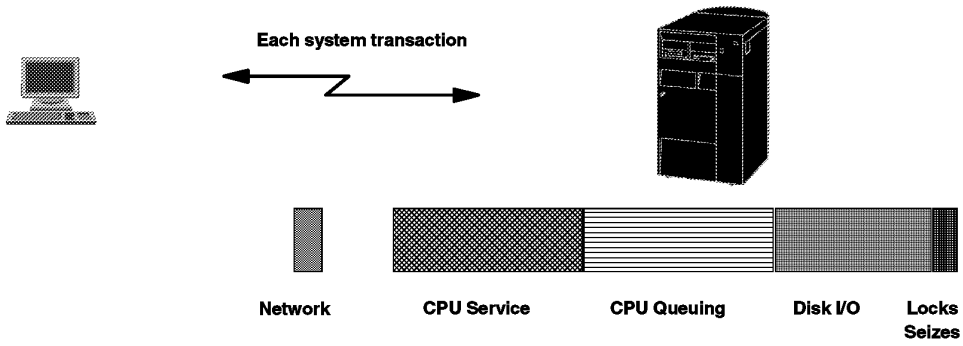


Figure 263. Main Frame Interactive Response Time Components

CPU service is the actual processing time needed to execute all of the instructions in the transaction.

CPU queuing is the time spent waiting for the processor to become available. In a typical multi-user environment, the processor may not be available when the task is ready to execute because other tasks of higher or equal priority are already queued ahead of it. The queuing time becomes more pronounced as the system becomes busier (higher CPU utilization). The effect is exponential and the queueing factor is approximated by:

```
Q(f)= 1 / (1 - (u**n))
```

In this equation, `u` is the cumulative CPU utilization due to equal or higher priority jobs, and `n` is the number of main processors

Since queuing is a function of CPU utilization, reducing the utilization by improving the efficiency of high volume transactions reduces queuing.

Disk I/O time is due to the physical I/O operations that need to be done as part of the transaction. There are several different categories of physical I/Os. However, generally, the synchronous ones directly affect performance. Due to the implementation of the AS/400 system's single-level storage, the application does not have direct control over the number of physical I/Os unless file parameters, such as Force Write Ratio (FRCRATIO) on output operations or Number of Records (NBRRCDS) on input processing, are specified. System tuning facilities, such as Expert Cache, Set Object Access (SETOBJACC), pool sizes, and so on, can also affect this number of physical I/O operations. In general, the more complex a transaction is, the more physical I/O operations are required and more CPU resources consumed. Making the transaction less complex tends to reduce both CPU and physical I/O requirements. Individual disk arms can also suffer from excessive I/O requests and exhibit similar delays due to queueing as the CPU. More information can be found on system tuning in the *OS/400 Work Management Guide*, SC41-5306.

Locks and seizes involve waiting for resources to be released by another task. An example of a lock is when a job reads a record with update intent. If another job attempts to read that record with update intent, this second job cannot unless the first one releases the record. Seizes occur in the microcode, and may sometimes occur when a task seizes an object, such as an index, to update it when a record is added, or if a new index needs to be built. DB2/400 has been enhanced to minimize the chance of seizes occurring and if they do, for short times. Excessive

seizes or locks usually indicates poor application design. You can use *AS/400 Performance Tools, 5769-PT1*, to find seizes and locks. More information can be found in *Performance Tools/400*, SC41-5340.

In comparison to the MFI environment, Java on the AS/400 system is expected to play a role in the client server environment, particularly in relation to the Internet. Figure 264 shows a typical client/server transaction.

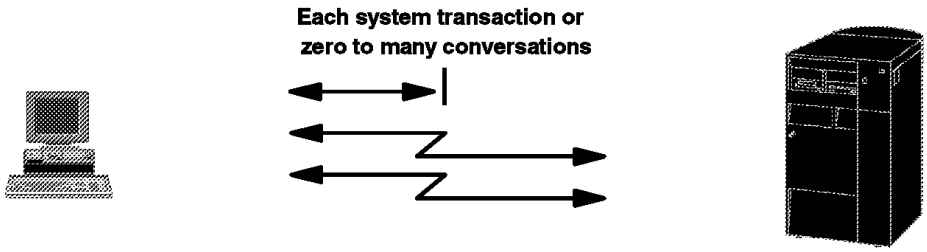


Figure 264. Client/Server Response

Every transaction (even a user clicking on a control button) can result in zero to many conversations with the server AS/400 system. With zero conversations, the total response time is due to the client. Every additional conversation adds to the network component, including turnaround time. The server overhead depends on the amount of processing that is required, typically performing database accesses.

It is important to understand that when a performance problem is experienced, it can be due to a combination of three major areas:

- Client
- Server (AS/400 system)
- Network

The response time components of each of these areas are shown in Figure 265 on page 333.

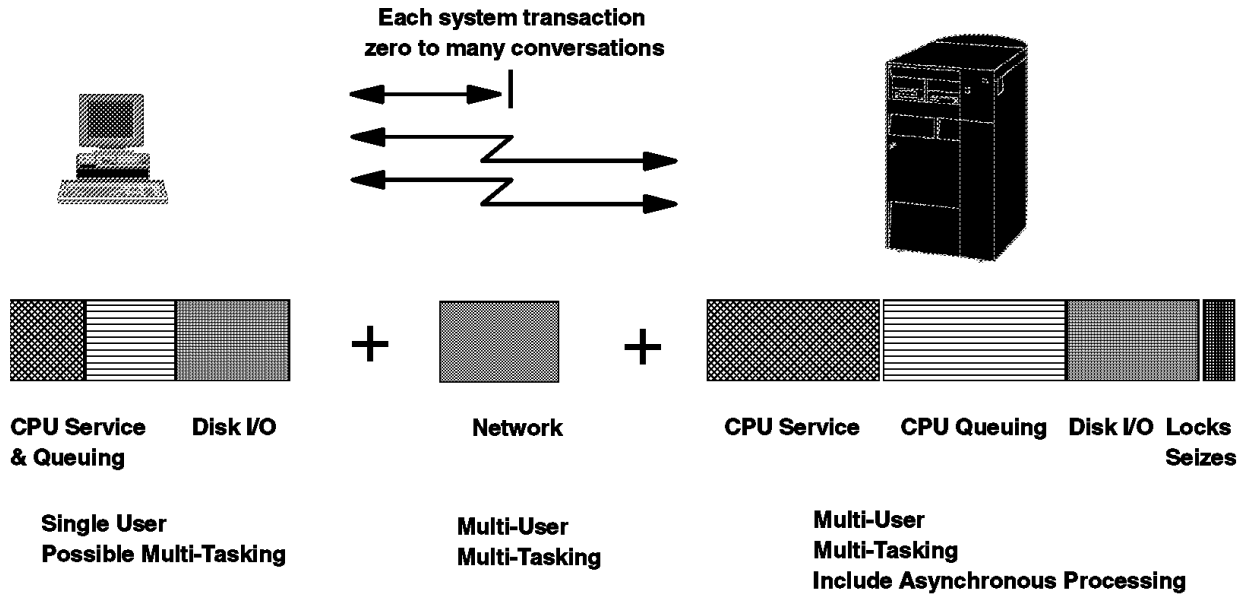


Figure 265. Client/Server Response Components

The remainder of this chapter addresses the server component of the response time. In the client/server environment previously described, a performance problem can manifest itself through a slow response time at the clients, a network congestion problem, or excessive use of the network or both, or through capacity constraints at the server. Ensure that the appropriate diagnostic tools are available to determine whether the problem is due to client, network, or server constraints.

There are actually several scenarios that fit the client/server model. Figure 266 on page 334 shows some that are more commonly used methods, including some AS/400 access methods overlayed.

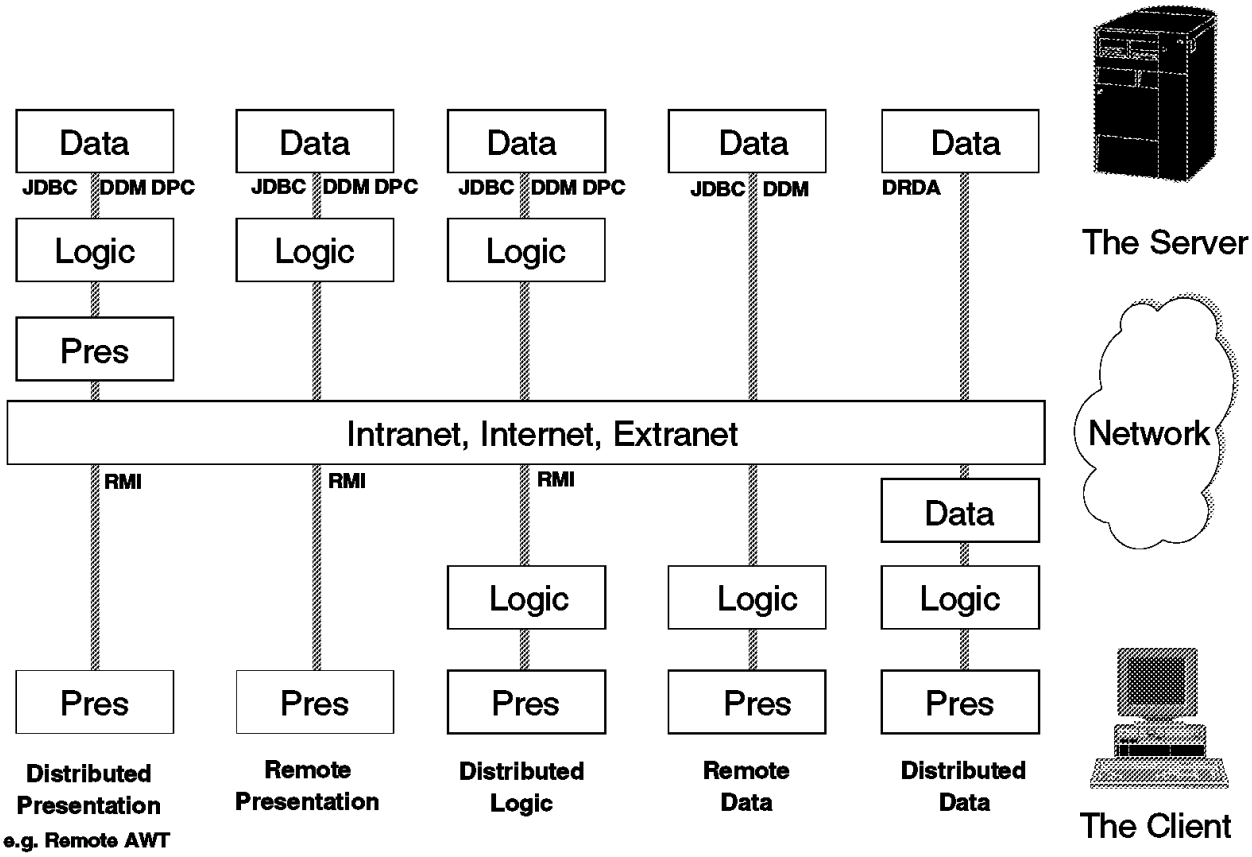


Figure 266. AS/400 Java and the Client/Server Model

In all five scenarios, there are two physical tiers composed of the client hardware and the server hardware. The application is organized according to three logical tiers:

- **Presentation:** Encompasses the user interface, generally pertaining to Abstract Windowing Toolkit (AWT), Swing classes, or HTML or XML code
- **Logic:** Involves business logic such as processing and enforcement of business rules
- **Data:** Access to the database and persistence

The three logical tier design is the recommended approach in implementing Java applications. Because of its portability, any of the logical tiers can be implemented within any of the physical tiers, depending on which hardware platform is most appropriate.

Because this redbook covers Java on the AS/400 system, the performance discussion only includes the three scenarios on the left, that is, Distributed Presentation, Remote Presentation, and Distributed Logic. These scenarios use tiers that allow business logic on the server, which is assumed to be written in Java. The other two scenarios are addressed in other books. The Remote Data scenario is covered in the redbook *Building AS/400 Client/Server Applications with Java*, SG24-2152. The Distributed Data scenario is covered by many books. However, a good reference is *Distributed Database Programming*, SC41-5702.

In these three client/server Java scenarios, user interaction can be done through a GUI and is not based on the 5250 data stream. All of these Java scenarios are excellent candidates for AS/400 e-servers, which provide the best price per performance for non-5250 applications. With the consolidation of the various models into the 7xx series, these application scenarios can now be run on AS/400 models that have much lower interactive features.

Under Distributed Presentation, all three logical tiers of the Java application run on the server. However, because the Abstract Windowing Toolkit (AWT) or Swing support for Java cannot run on the AS/400 system, any method calls to AWT are actually executed on the client. The Remote AWT support within the AS/400 system was originally built over Java's Remote Method Invocation (RMI) standard. However, it has been modified to use direct sockets connections for improved performance.

For Remote Presentation, the Java business logic and the data reside on the server with the presentation on the client. A common example is a Web page containing an applet that has been downloaded from the server. This applet handles the presentation logic, and then invokes remote methods on the server that perform the business logic. RMI is also the expected way of invoking these remote Java methods.

Distributed Logic has a portion of the business logic on both physical tiers. The client performs some of the simpler business rules (for example, validation of input) prior to invoking remote Java methods on the server. Also, because the business logic on the server is written in Java, RMI may be the facility for invoking remote methods.

13.6 Addressing Performance

All performance situations can be addressed in four major ways (see Figure 267). Any of these can provide a significant performance improvement. Conversely, if any of these ways are not properly implemented, performance can be negatively affected. For example, the best hardware can be installed, but if the system is not tuned properly, there is a good chance that performance will suffer.

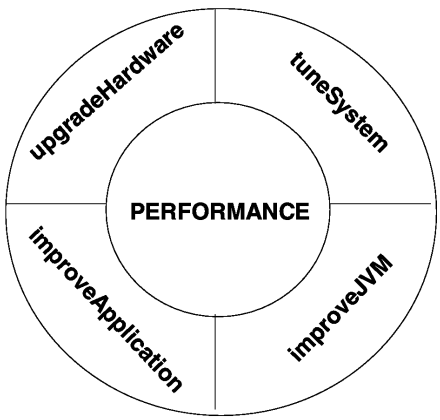


Figure 267. Performance Factors (Donut)

The criteria generally used to determine the most appropriate approach include:

- The availability of the option
- The cost of implementation versus expected improvement
- The time it takes to implement

Hardware upgrades are not discussed in detail. It is a rational assumption that if the components of response time on the AS/400 system are largely due to CPU service time and CPU queuing, a processor upgrade should provide improvements. The latest generation of processor technology can provide performance advantages over its predecessors, even if they have similar CPW ratings. Similarly, if disk I/O service time is excessive, adding more disk arms or faster disk arms (and memory) reduces that response component.

Tuning the system is discussed because the implementation of the JVM on the AS/400 system is new. Work management concepts (including the startup of the JVM, Qshell, and thread support) are discussed. In many environments (Java or other environments), system tuning can result in noticeable improvements within a short period of time.

Application design and coding recommendations are also made. This is usually the area where the most significant, long term performance improvements can be attained.

Significant improvements to the Java runtime support can be expected in future updates. These can be attributed to ongoing performance enhancements to both AS/400 Java support and Sun JDK support.

13.7 Work Management and Tuning

As mentioned previously, system tuning can provide noticeable performance improvements with the least time and with minimal cost. Some of the significant differences, as compared to traditional RPG or COBOL environments, are the use of a Batch Immediate (BCI) job for the JVM and the inclusion of native thread support.

13.7.1 Ways of Running Server Java Applications

There are four major ways of running Java programs on the AS/400 system. These include:

- Through the JAVA or RUNJVA command, which uses the BCI job QJVACMDSRV.
- Through the Qshell Interpreter, where the JAVA built-in utility can be used to run a JAVA program or a script (text) file can be used to run several JAVA programs. QSHELL uses the QZSHSH BCI job.
- Through the AS/400 Operations Navigator, by double-clicking or right-clicking on a class file. The Java application actually runs within a QZSHSH BCI job under subsystem QSYSWRK.
- As a servlet within the WebSphere Application Server. A JVM is actually started within the batch job representing the HTTP server instance, under subsystem QHTTPSVR.

Various types of Java programs may be observed, for example, running Enterprise Java Beans (EJB) under an Enterprise Java Server, but these make use of the standard JVM support. Java applications still run inside the BCI jobs that are spawned.

For those execution environments that have a command line interface available, there are significant advantages in running the Java applications in batch mode, for example, not interactively through a 5250 session. These advantages include:

- The interactive or 5250 workload is minimized. An AS/400 system with a lower interactive feature can be used, which provides a much lower hardware cost.
- A 5250 session is not used up to execute a long running application. Batch mode eliminates the security risks of keeping a 5250 session active.
- From a work management standpoint, it is easier to control the attributes of a batch job or a set of batch jobs. In a mixed mode environment where both Java applications and traditional 5250 applications are running, it is easier to segregate the batch Java applications.
- The creation of a temporary data queue and its associated Dynamic Screen Manager for every Java session is eliminated. These facilities are used for passing messages, which includes standard input and standard output between the Java application and the 5250 session.

13.7.2 Major Runtime Steps

Each Java application undergoes four major execution steps. These steps include:

1. Spawn the Batch Immediate job and start the JVM.
2. Search, verify, and load classes.
3. Execute Java instructions.
4. Access system services.

13.7.3 Required: The Latest Software

As expected, the latest software provides the best performance. The V4R3 implementation of JDK 1.1.6 performs much better than the V4R2 implementation of JDK 1.1.4. Initial comparisons show that JDK 1.1.7 available with V4R4 has even larger improvements. JDK1.1.7 is available for V4R3 systems through a group PTF SF99066.

Most of the performance improvements can be attributed to enhancements to the AS/400 JVM, as opposed to improvements in the JDK.

In addition to performance enhancements, having the latest operating system software and the JDK software generally fixes prior problems and provides more functionality.

The latest PTFs should also be installed to take advantage of any performance improvements and fixes. This applies to microcode and program product PTFs. Java-related PTFs are available as group PTFs and can be ordered using the standard Electronic Customer Support (ECS) facilities.

Information about these PTFs is available from the following Web sites:

- AS/400 Developer Kit for Java (5769-JV1) at the site http://www.as400.ibm.com/developer/java/devkit.html
- AS/400 Toolbox for Java (5769-JC1) at the site http://www.as400.ibm.com/toolbox

13.8 Starting the Java Environment

When a Java application is started, a Batch Immediate (BCI) job is spawned by the job that ran the command. Spawning a BCI job is not as costly as a full job initiation, such as that generated by a Submit Job (SBMJOB) command. The reason is that a BCI job uses the same environment as the job that spawned it (that is, security, work management parameters, and so on). A BCI job does not go through a JOBQ and uses fewer objects than a full job initiation.

The JVM is then started within the BCI job. One variation is when dealing with Java servlets and Java Server Pages (JSP) within the WebSphere Application Server, which is described here.

Frequent initiations should be minimized for reasons that will become clearer as the startup details are described. Spawning a BCI job is more expensive than starting a new thread. Java classes should then only be called or used if they are relatively long running. For example, do not run a Java class from an RPG program to frequently perform a Java date conversion routine. Doing this can result in frequent BCI job spawning.

However, this does not limit the use of Java to a batch environment. It fits very well into the interactive client/server scenarios previously described. Regardless of the transport mechanism, for example, RMI, sockets, and so on, the server classes can be started and remain active in a "Never-ending" mode.

13.8.1 Using the JAVA or RUNJVA Commands = QJVACMDSRV BCI Job

Examine the details of starting a Java program with the RUNJVA or JAVA commands:

- A BCI job named QJVACMDSRV is spawned under the same work management environment, that is, the same subsystem and routing entry as the job that ran the JAVA command. This is the job that actually loads the classes and runs the Java program.
- If the JAVA command was run interactively (that is, through a 5250 session), a temporary data queue is created by the job from which the command was run. The following message is shown in the interactive job log: `CPC9801: Object QP0ZTRML type *DTAQ created in library QTEMP`.

  This *DTAQ is used to communicate events between the BCI job and the 5250 session. This data queue is deleted when the Java program ends and the BCI job is terminated.
- If JAVA was run interactively, an internal Dynamic Screen Manager (DSM) session is created. This checks the 5250 device attributes (for example, screen size and double-byte character set capability).

These steps are shown graphically in Figure 268 on page 339.

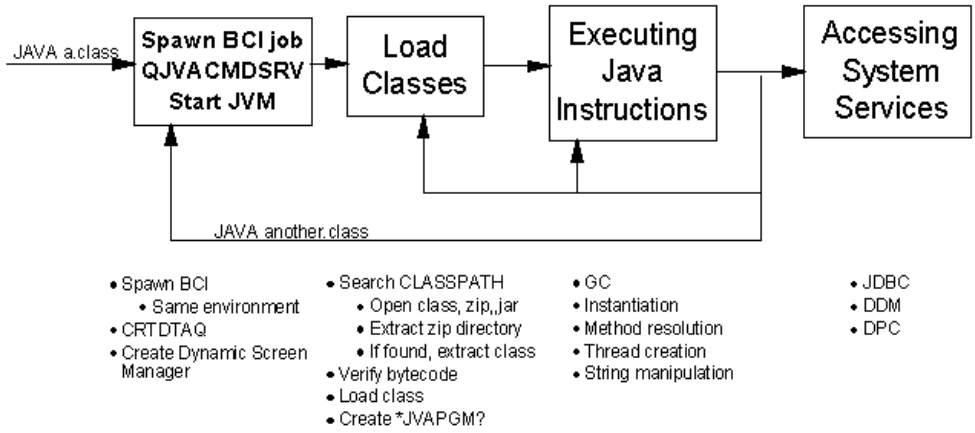


Figure 268. Running Java with the JAVA or RUNJVA Command

Figure 269 shows a Work with Active Jobs display where interactive job QPADEV0003 ran a Java program. Note its entry under the Function column, CMD-JAVA. This program was initiated using the following JAVA command:

```
JAVA CLASS(SlowDatabaseServer) CLASSPATH('/llames/SlowFast')
```

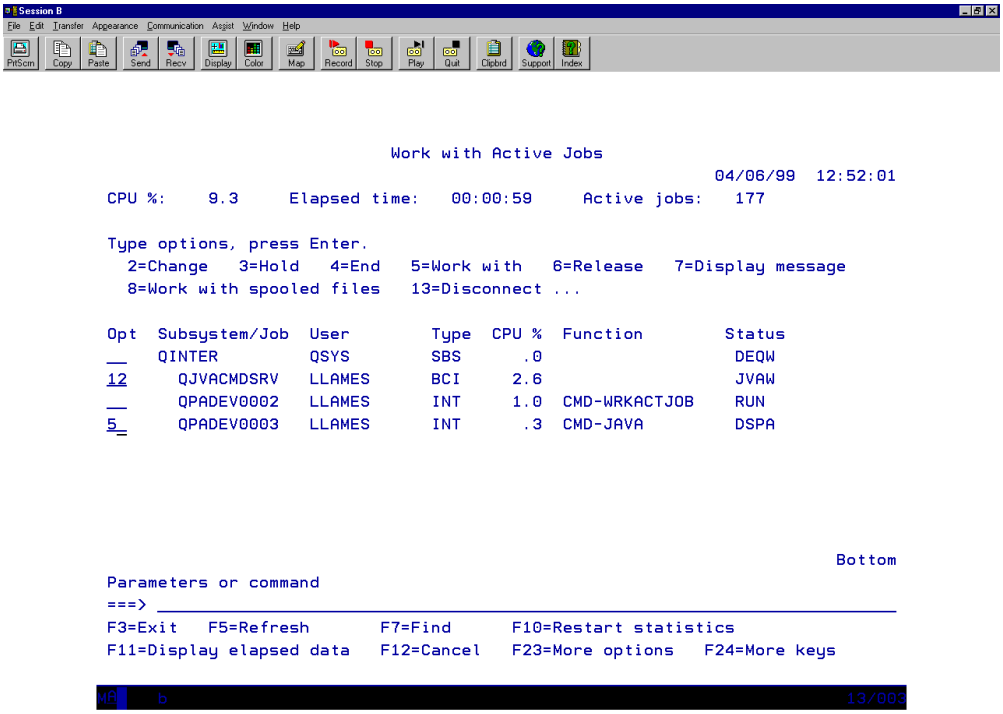


Figure 269. WRKACTJOB Showing INT and BCI Jobs from the JAVA Command

This action spawned the BCI job QJVACMDSRV under the same environment (that is, subsystem, job description, and class). A Batch Immediate job is similar to a UNIX process being spawned and has similar characteristics as the job that spawned it.

The initial job log entry in the QJVACMDSRV job is similar to that of the interactive job that spawned it:

```
Job 050076/LLAMES/QJVACMDSRV started on 04/06/99 at 13:02:01 in subsystem
  QINTER in QSYS. Job entered system on 04/06/99 at 13:02:00.
Public write authority on "/llames/SlowFast".
Job 049373/QUSER/QSQSRVR used for SQL server mode processing.
```

You can minimize the relative costs of starting up the Java environment (that is, job initiation and termination, creation and deletion of the temporary data queue in 5250 mode, and the Dynamic Screen Manager). To do so, Java classes should only be called or used when they are relatively long running.

13.8.2 Running Java from QSHELL = QZSHSH BCI Job

The second method of running Java classes is through the Qshell Interpreter. This is invoked through the **qsh** or Start Qshell (STRQSH) commands and provides a POSIX- and X/Open-based command interpreter from which the JAVA command can be run.

This interface provides additional efficiencies when starting several Java programs because a SHELL script can be stored in a text file and used to run several commands in a row, including starting up several Java programs. With a script, several Java programs can be run under the same BCI job instead of having to spawn a BCI job for each one. This advantage is also available when issuing several Java runs interactively within the same QSHELL session.

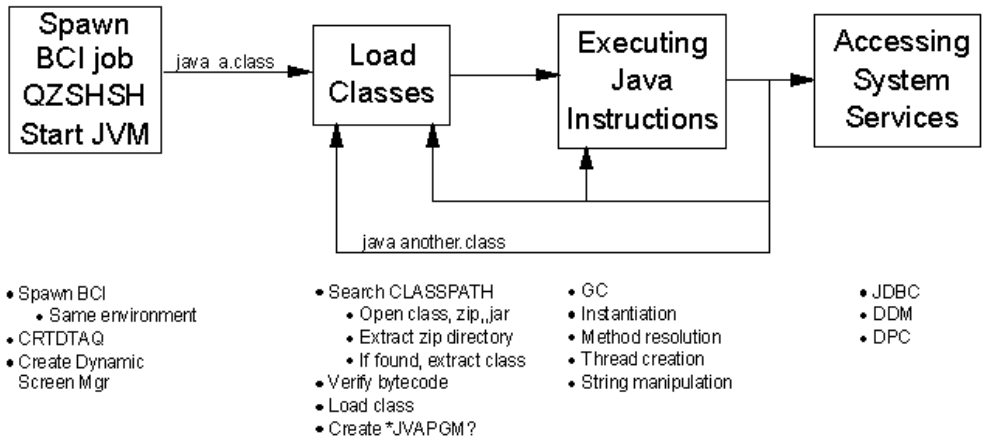


Figure 270. Running the Java Utility within QSHELL

The spawning of a BCI job QZSHSH is similar to that of QJVACMDSRV. In Figure 271 on page 341, you can see the BCI job that was spawned from the interactive job running CMD-QSH.

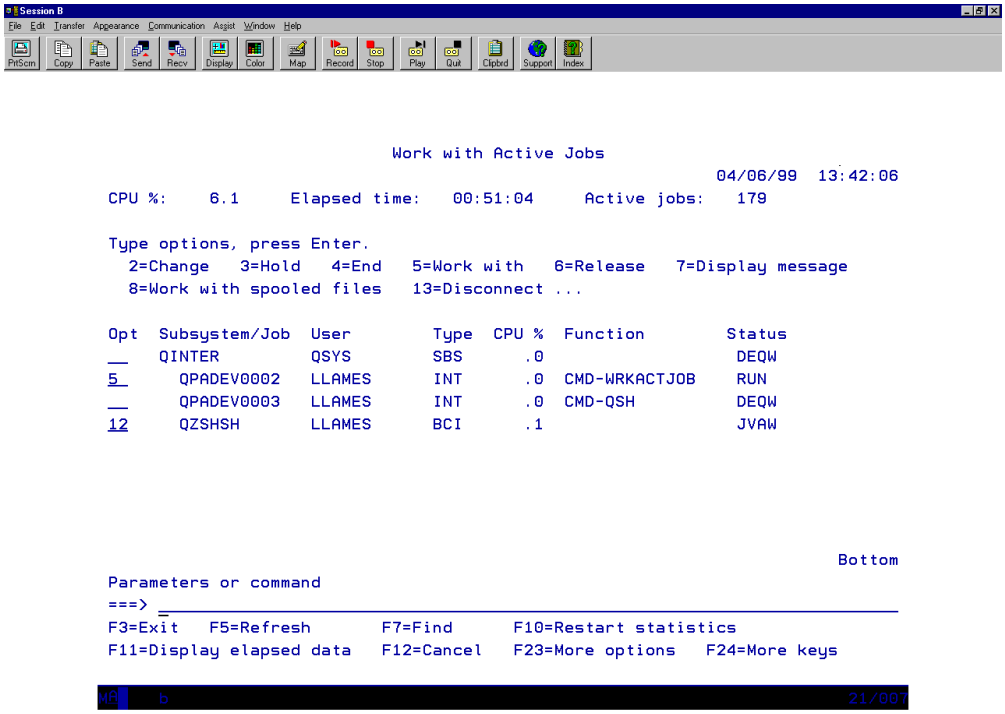


Figure 271. WRKACTJOB Showing INT and BCI Jobs from the QSHELL Interpreter

To run a QSHELL session with a QSHELL script in the form of a text file, use the QSH or STRQSH command with the parameter CMD(textfile_in_IFS_syntax). If the script needs to run a Java program, the following environment variable should be in effect:

```
QIBM_MULTI_THREADED=Y
```

When using a QSHELL script, a separate process is spawned to run the Java application. This is another BCI job named QP0ZSPWT.

For more information about QSHELL, refer to the *Qshell Interpreter Reference*, which can be accessed starting with the AS/400 Web site at:
http://www.as400.ibm.com/library

Follow the links to **AS/400 manuals & Redbooks**. Select a language under **Information Center**. Then, follow the link to **Java**.

13.8.3 Operations Navigator = QZSHSH BCI Job

To run a Java application from Operations Navigator, navigate to the directory that contains the class file. This is illustrated in Figure 272 on page 342.

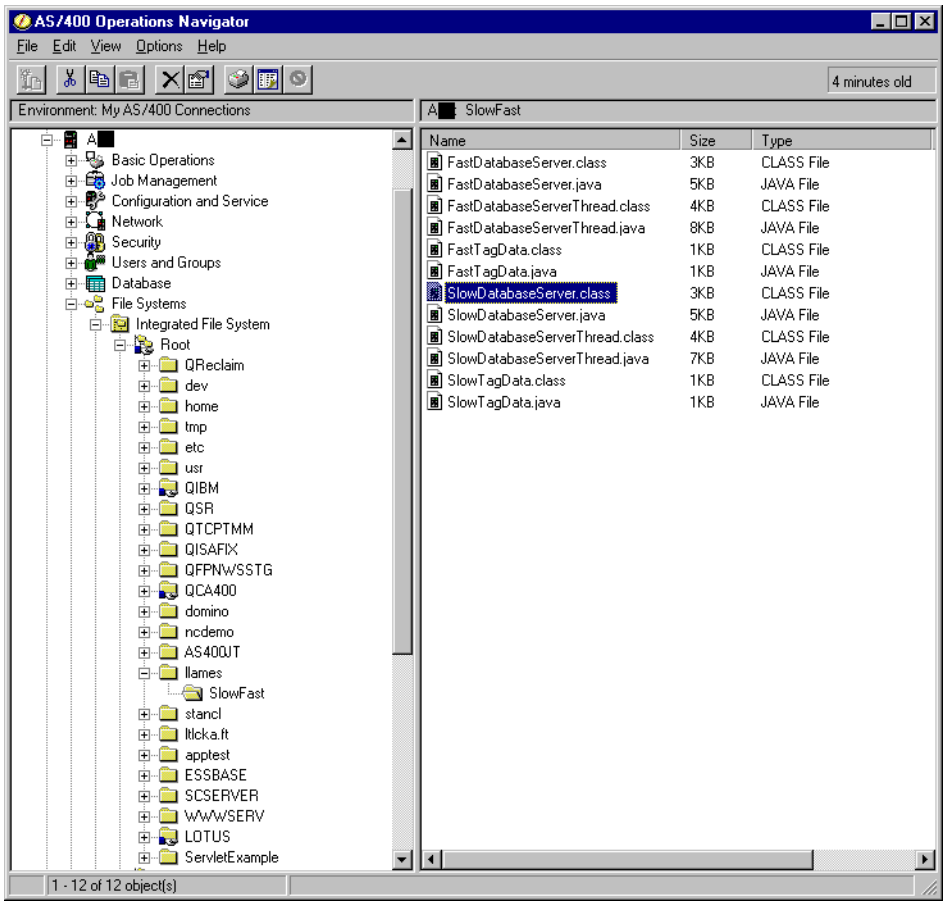


Figure 272. Operations Navigator — List of Java Related Files in Integrated File System

By double-clicking or right-clicking on the main class file, you can navigate to a window with the various parameters for running an AS/400 Java application. Any input arguments that are required to run the application should be enclosed in double quotation marks.

The Java application will run within a JVM inside a BCI job named QZSHSH. This job is under subsystem QSYSWRK. The user profile for this job will show up as QUSER. However, the user profile that is actually in effect is the same one used to establish the session. When looking for a particular QZSHSH job that is running a Java application from Operations Navigator, use the command:

```
WRKOBJLCK OBJ(user profile used to establish session) OBJTYPE(*USRPRF)
```

13.8.4 WebSphere Application Server

When running Java servlets under the WebSphere application server, the characteristics are a little different. The HTTP server instance that is configured to run Java servlets and Java Server Pages (JSP) contains the Java Virtual Machine.

The JVM is started inside the batch job representing the HTTP server instance. In the example in Figure 273 on page 343, the HTTP server instance is LLAMES. The Java programs will run inside the batch (BCH) job with the same name as the server instance. Note the program name and the subsystem.

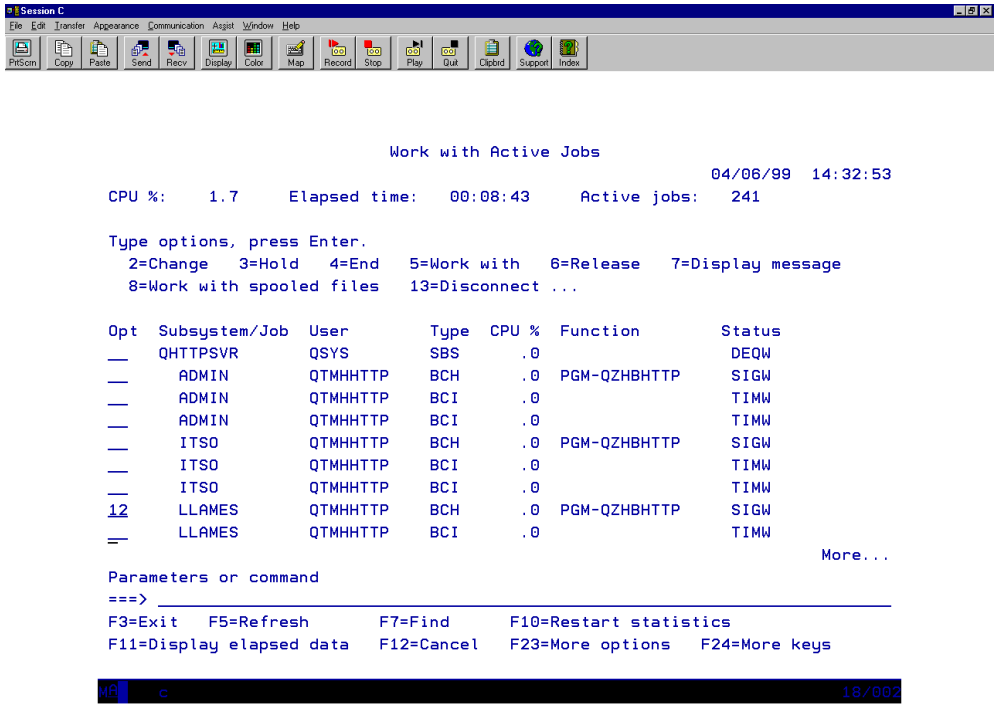


Figure 273. WRKACTJOB Showing the BCH Job from the WebSphere Application Server

13.9 Searching and Loading Classes

When running Java under OPTION(*VERBOSE), many messages that relate to class loading are generated. These messages include:

```
Loading class: java/lang/Object.class
Loading class: java/lang/Class.class
Loading class: java/lang/String.class
Loading class: java/io/Serializable.class
Loading class: com/ibm/as400/system/MIPtr.class
Loading class: java/lang/Cloneable.class
Loading class: java/lang/Void.class
Loading class: java/lang/Byte.class
Loading class: java/lang/Number.class
Loading class: java/lang/Character.class
Loading class: java/lang/Double.class
Loading class: java/lang/Float.class
Loading class: java/lang/Integer.class
Loading class: java/lang/Long.class
Loading class: java/lang/Short.class
Loading class: java/lang/Boolean.class
Loading class: a970107e/RMIExample/HelloImpl.class
Loading class: . . . . . .
```

There are several "system" classes that are loaded before the actual application class (HelloImpl.class in this example) is loaded and starts executing. After this, other required classes are loaded in a "lazy load" mode or as needed.

When classes are loaded, the system checks if each class has an associated Java program unless the runtime parameter specifies that the application be run in interpreted mode. If no Java program exists, the JVM invokes the transformer that creates the Java program. The default optimization level for the Java program is 10.

One main difference between CRTJVAPGM . . . OPTIMIZE(30) and OPTIMIZE(40) in the Direct Execution environment is that the latter is meant for fastest execution. Therefore, more classes are pre-loaded. OPTIMIZE(30) uses the standard Java lazy resolution scheme for class loading, where classes are loaded as they are encountered. Under OPTIMIZE(40), a more active AS/400 pre-resolution scheme is used such that all external references in a method are resolved when the method is first invoked. This may result in more classes being pre-loaded so that execution is faster.

At times, a particular class load seems to take several seconds while most of the other classes seem to load much faster. In most cases, the delay is due to searching for a class rather than in actually loading the class. Class size is not a primary reason for such delays. Another reason can be that the transformer is being invoked and the internal Java program is being created.

Classes are searched for in the order at which directories are specified in the classpath.

There are several techniques that can be implemented to improve the class search and load times. These techniques are described in the following sections.

13.9.1 Create Java Program (CRTJVAPGM) for jar, zip, and class Files

Jar and zip files are treated the same way (that is, same compression and decompression algorithms) on the AS/400 system. One difference is that a jar file includes a manifest for authorities and library maintenance.

Make sure that the Java jar, zip, and class files have been processed through the transformer. They should have the "hidden" Java program created, particularly for the sun.zip, java.zip, and rawt_classes.zip files. These can be verified using the Display Java Program (DSPJVAPGM) command as shown here:

```
DSPJVAPGM CLSF('/QIBM/Proddata/Java400/lib/sun.zip')
DSPJVAPGM CLSF('/QIBM/Proddata/Java400/lib/java.zip')
DSPJVAPGM CLSF('/QIBM/Proddata/Java400/lib/rawt_classes.zip')
```

The zip files associated with the AS/400 Developer Kit for Java are installed with Java programs. If, for some strange reason, the Java programs for the JDK files disappear, the licensed program product (5769-JV1) must be re-installed.

The same applies to any jar, zip, or class files that are part of the application that run on the AS/400 system. Recall that to run a class in the direct execution (non-interpreted) environment, the Create Java Program (CRTJVAPGM) command should be used either explicitly or implicitly to create the Java program. Using the OPTIMIZE(40) parameter enhances execution performance but reduces debug capabilities such as changing the program variables.

Jar or zip files without the Java programs will exhibit apparently long class search and load times. When a class has to be loaded from the zip file, the zip file is opened, its directory is searched for the class, and a Java program has to be

created implicitly for the class that was just loaded. The default optimization level for the new Java program is OPTIMIZE(10) unless that parameter was modified in the command. Therefore, not only is the class loading slow, but the class execution may also be slow because of the low degree of default optimization.

Prior to V4R4, the resulting Java program is not persistent when implicitly done for classes that are loaded from jar or zip files. This implicit transformation has to be performed every time a JVM is started. In V4R4, the Java programs that result from the transformation for jar and zip files is persistent.

However, explicit transformation of jar and zip files through CRTJVAPGM should still be done even in a V4R4 environment. Relying on implicit transformation results in more linkages between the individual Java programs and the classes, and more storage overhead for the individual Java programs. It also can result in a possible loss of inter-class optimization and of method inlining opportunities.

Also prior to V4R4, the AS/400 Toolbox for Java classes stored in jt400.jar and jt400.zip were not shipped with the Java program. Using AS/400 Toolbox for Java classes on a server Java application resulted in slow startup times because of the implicit transformation occurring. Starting with V4R4, the Toolbox jar and zip files are shipped with their respective Java programs.

A class that is not stored in a zip or jar file also needs its Java program to run well. If it has not been put through CRTJVAPGM, a persistent Java program is created internally the first time that the class is run.

13.9.2 Minimizing the Directory Search

Searching through directories for a Java class is similar to searching through a library list for a non-qualified program name. Then, similar recommendations apply. This implies that unnecessary directories should be removed from the CLASSPATH so that the directory search is no deeper than necessary.

In addition, the most frequently used directories should be placed at the beginning of the CLASSPATH. Otherwise, searches of the starting entries of the CLASSPATH may result in many misses.

13.9.3 Storing Related Java Classes in jar or zip Files with Java Programs

The best scenario for searching and loading classes is to have them in a jar or zip file that has a Java program associated with it (that is, created through the CRTJVAPGM command). Opening a zip file and scanning through its directory is faster than looking for an individual class within a directory.

This technique may not be viable during a development environment because it can result in frequent changes to the zip or jar files. It is generally applied when the classes are relatively static. In addition, debugging is easier for individual class files.

In V4R4, individual linkages are available between a Java program and its associated class stored inside a jar or zip file. This reduces the maintenance because when a class changes, only its own Java program is affected.

In order of class search and load performance, with the fastest performer first, the following common environments are encountered:

1. Zip or jar file with existing Java program
2. Regular class directory search
3. Zip or jar file without Java program (worst)

As mentioned in Section 13.9.1, “Create Java Program (CRTJVAPGM) for jar, zip, and class Files” on page 344, the results of implicit transformation for jar and zip files in V4R4 are now persistent. This means that jar and zip files now fall into the first (and fastest) category.

13.10 Threads and Tuning

A thread, or thread of control, is the path taken by a program during execution. Thread support is available in the Java language. Every AS/400 job or process has at least one thread. This thread can start or spawn others that can run independently of one another, although there are ways of synchronizing threads within the Java language.

The AS/400 thread support is implemented through kernel threads, which lets the machine schedule the execution of threads and can take advantage of symmetric multiprocessor (SMP) hardware.

For more information about threads and threaded applications, go to the AS/400 Information Center, and click on **Programming**.

Thread support actually provides a way of reducing response times by multi-tasking within a single process. On the AS/400 system, a process is represented by a job. For example, if an order processing program needs to print a report, it can create another thread to do the printing asynchronously. Creating a new thread (considered a lightweight operation) is far less expensive than initiating a new job or process.

Some ways to observe a job's threads are by using WRKACTJOB option 12 or WRKJOB option 20. There is also thread identification information in the job log to indicate which thread generated a particular message.

Figure 274 on page 347 shows the threads of a QJVACMDSRV BCI job.

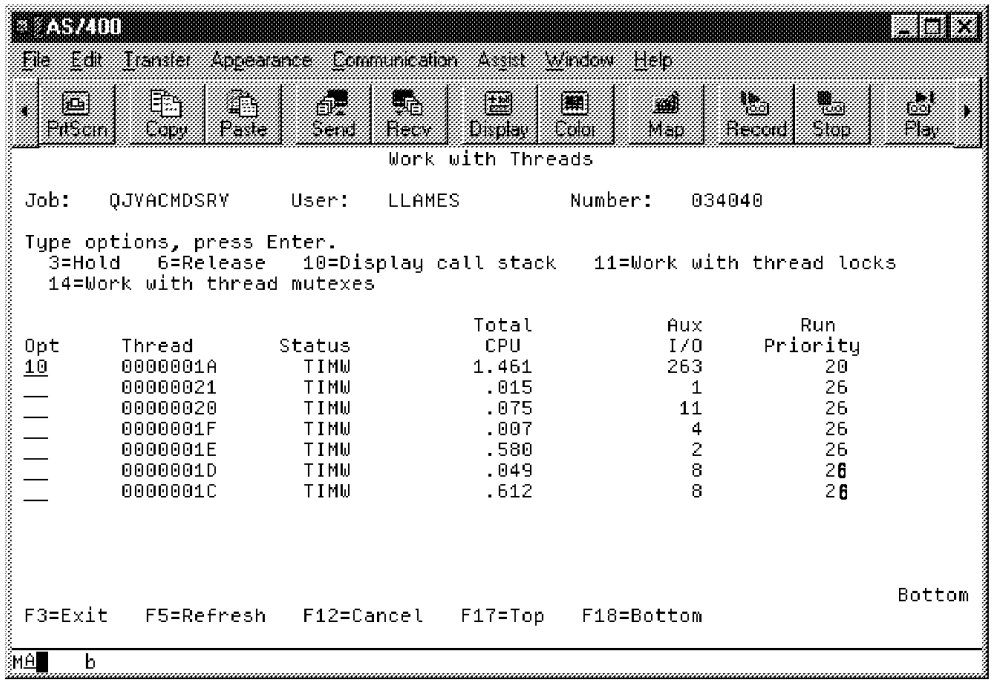


Figure 274. Work with Threads Display

Note

Although you may already be familiar with system tuning, the introduction of native thread support adds new information of which you need to know. Therefore, it is important that you read the following sections.

13.10.1 Initial Thread

In the V4R2 Java environment, an initial thread is created to perform the initial service functions related to starting up the JVM. It also handles interrupt requests when the BCI job that it is running under is ended and then performs cleanup functions. When this thread is doing nothing, it sits at TIMW status. This thread has the lowest thread number assignment since it is the first one that is started.

Starting with V4R3, this initial thread is no longer used and can no longer be observed. The first thread that shows up, usually with the lowest thread assignment, is the first Java application thread.

13.10.2 Run Priorities

The initial thread in V4R2 runs at the same run priority as the BCI job that created it, which runs at the same priority as the job that spawned it. When the actual Java main method starts, the initial thread starts the actual Java application thread whose run priority is lower by a relative amount, usually 5. As mentioned in Section 13.10.1, “Initial Thread” on page 347, the initial thread is no longer used in V4R3 and subsequent releases.

Java has 10 priorities ranging from 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY). The higher the integer value used in the java.lang.thread setPriority() method is,

the higher the run priority is. The default priority of 5 is the same as NORM_PRIORITY.

When converting Java priorities to AS/400 run priorities, the equation used is:

```
AS/400 Thread Priority = (BCI job's run priority + (11 - Java priority))
```

For example, when a BCI job is at priority 20 and a thread has a Java *NORM priority of 5, the resulting AS/400 run priority for the thread is 26. This is derived from the following equation:

```
AS/400 Thread Priority = (20 + (11 - 5))
```

If the JAVA command is run in batch mode so that, by default, the BCI job is running at priority 50 and a thread is set to the maximum Java priority of 10, the thread's AS/400 priority is 51, as shown in this equation:

```
AS/400 Thread Priority = (50 + (11 - 10))
```

When a BCI job's run priority is changed (for example, from 20 down to 30), the thread priorities are also changed by the same relative amount (for example, from 26 to 36). However, the change does not occur immediately. A thread's priority is changed only the next time it hits a long or short wait, or when it reaches MI time slice (external time slice) as specified in the AS/400 *CLS object. Its priority does not change if it has been sitting at a prolonged wait state or TIMW status. Reaching the system internal time slice does not cause the thread's priority to change either.

To compete on the same level as other jobs on the AS/400 system, the BCI job's run priority should then be set accordingly. This applies in a mixed mode environment, for example, Java applications and traditional 5250 interactive applications running in the same system. For example, if the Java application is used for online transaction processing and there are interactive RPG 5250 applications running at priority 20, the BCI job's priority should be set to 14. This allows its threads to also run at priority 20 and compete at the same level as the interactive RPG applications.

The effects of dynamic priority scheduler for Java classes whose BCI job runs at priority 20 are not expected to be any different. In such a case, the work then runs at a default priority of 26. This means that the threads that consume most of the CPU run within the delay cost range of priority 23 to 35. The Dynamic Priority Scheduler is established by the system value QDYNPTYSCD and defaults to a value of 1 (On). The standard priority ranges are:

- Priority 10-16
- Priority 17-22
- Priority 23-35
- Priority 36-46
- Priority 47-51
- Priority 52-89

Two of the threads are garbage collection threads. They usually have the next lowest thread number assignments to the initial thread.

When running a Java class from the JAVA or RUNJVA commands in a V4R2 system, there is a **GCPTY** parameter that is used to set the garbage collection priority. Its valid values are 10 (lowest), 20 (default), and 30 (highest).

With a GCPTY value of 20, the threads' run priority will be the same as a Java NORM_PRIORITY, or 5. In this example, they should run at an AS/400 run priority of 26.

With GCPTY(10), the garbage collection threads run at an equivalent Java priority of 0, even though no such Java priority exists. Recall that the lowest Java run priority is 1. However, a Java priority of 0 must be used when calculating the the resulting AS/400 GC thread priority using the equation previously described. Therefore, when the BCI job is at priority 20, the garbage collection threads run at an AS/400 run priority of 31.

GCPTY(30) translates to the highest Java priority of 10. With the BCI job at priority 20, the GC threads run at an AS/400 run priority of 21.

Starting with V4R3, this parameter is no longer effective. It is only included as a command parameter for compatibility with Java applications deployed in V4R2. In V4R3, the garbage collection threads are assigned the same run priorities as the BCI job.

13.10.3 Activity Level

Each active thread occupies an activity level, even though several threads can be running under one job. That is because each thread is an independently dispatchable task. Remember that with the kernel thread support, AS/400 threads can take advantage of SMP hardware. Therefore, the initial activity level setting for a Java pool should be higher, based on the expected number of active threads, not on the expected number of active jobs. Observations within peak periods should then be used to set the activity level based on ineligible rates and standard pool faulting guidelines.

13.10.4 Time Slice

Because threads are separately dispatchable tasks, they have their own separate counters for the amount of CPU that they have individually used. Each thread's counter is compared to the TIMESLICE parameter that it is running under as well as to the internal time slice value for the system.

Each thread can independently hit its time slice limit. This limit is derived from the parent job's time slice value.

The total CPU consumed by all threads in a job is accumulated to keep the job from exceeding the maximum CPU time limit if specified in the Work Management class (*CLS) object. This limit is established with the CPUTIME parameter on the CRTCLS command or the CHGCLS command.

13.10.5 PURGE

If the job is multi-threaded (a BCI job is capable of being multi-threaded) and one of its threads goes into a long wait, the other threads are checked for any activity. If none of the others are active, the BCI job's Process Access Group (PAG) becomes a candidate for purging. It then uses the standard storage management algorithm for PURGE.

The only difference between the Java environment and RPG or COBOL environments is that there are now other threads to consider.

In V4R4, this is all moot because the effect of the PURGE has been removed, and the Process Access Group (PAG) structure as we know it is no longer exists.

13.10.6 Main Storage Pool Segregation

The standard recommendation for main storage pool assignments is to keep jobs or processes with common page requirements in the same pool. The reason for this is to minimize the probability that processes will steal pages from one another. Java is no different from other applications from this standpoint.

A Java application requires much more main storage than an equivalent application coded in a traditional language such as RPG or COBOL.

13.11 Java Instruction Execution

The execution of Java program instructions is the third major area where a Java application incurs overhead. This overhead can be reduced by using certain compilation and deployment options. From a coding standpoint, there are certain recommendations to which you should adhere. Most of the coding recommendations are also applicable to other platforms.

13.11.1 Compile Options for javac

Java source is compiled into bytecodes using the **javac** command. One way is to do the compilation on a client, and then move the generated bytecodes to the AS/400 Integrated File System. Another way is to run the command on the AS/400 inside Qshell. Consider these options:

- **javac -O:**

  An object-oriented design is generally more expensive than a functionally equivalent procedural design because of method resolution. When a method is requested, it has to be searched for then invoked.

  The -O option on the AS/400 system does method inlining. Any method that is final (private, static, or protected) is a candidate. Runtime performance is improved because method call overhead is reduced.

  Because a final method cannot be sub-classed or overridden, the method that will be used is known. A copy of that method's bytecodes are then included in the caller (inlining) so that the method execution is much faster. The trade-off is that the class size will increase.

  In V4R4, the transformer was enhanced so that at OPTIMIZE(40), method inlining is automatically performed. The use of the -O option is no longer necessary under such circumstances.

- **javac -g:**

  This option is needed so that when debugging the class, variables can be displayed or changed.

13.11.2 Explicit Java Transformer CRTJVAPGM (Optional but Recommended)

Using the Create Java Program (CRTJVAPGM) command on a class file results in a "hidden" Java program being created for that class. The program is considered "hidden" because it is not visible through the object commands.

This causes the class to run at a default optimization level of 10 unless a different OPTIMIZE parameter was used. Once an application is deemed stable and has minimal need for debugging, the transformer should be run at OPTIMIZE(40) to improve performance. One exception is the AS/400 Toolbox for Java, as described in Section 13.11.4, “Optimization Levels” on page 352. The program is created by transforming the class file's bytecodes into an optimized program object attached to that class. This program object contains native 64-bit RISC machine instructions.

When the **JAVA classname** command is run, the program object executes with significant performance improvement over the interpreted mode. One exception is when the CRTJVAPGM is run with OPTIMIZE(*INTERPRET). The Java class then runs under bytecode interpretation, a much slower process. V4R4 provides a new command to Change Java Program (CHGJVAPGM).

13.11.3 Automatic Java Transformer

If a class, jar, or zip file does not have the program object created prior to execution, the Java transformer is automatically run to create a program object under the default OPTIMIZE(10) level. Note the difference between the individual class files and the classes that are stored in jar and zip files.

- **class files:**

  If the class file does not have a hidden Java program object associated with it, one is automatically created by the Java transformer the first time it is run. It is possible to use a value different from 10 (for example, a higher optimization level by using a different OPTIMIZE parameter setting in the JAVA or RUNJVA command). This program object is persistent until the class is deleted or the Delete Java Program (DLTJVAPGM) command is run on it.

- **jar and zip files:**

  A performance enhancement in V4R4 allows the Java program created automatically for a class stored in a jar or zip file to be linked directly to the class. That makes the Java program persistent. However, we still recommend that the transformer be explicitly run on all jar and zip files because doing so provides several advantages:

  – Creating a Java program over an entire jar or zip file results in smaller Java programs. Individual links between a Java program and its respective class inside the jar or zip file do not have to be created.

  – The first user does not pay a performance penalty when the application is first touched.

  – Inter-class binding and method inlining is done when an entire jar or zip file is processed with CRTJVAPGM under OPTIMIZE(40).

  Prior to V4R4, performance could be worse with respect to jar and zip files, as explained in the following paragraphs.

  If a jar or zip does not have a program object associated with it, performance is expected to be poor. Class searching and loading takes much longer (see Section 13.9, “Searching and Loading Classes” on page 343).

  In addition, once the class is found in the jar or zip file's directory, a non-persistent program object is created for it. This program object is also run at a default OPTIMIZE(10) level unless a different value is specified in the JAVA or RUNJVA command's OPTIMIZE parameter.

An obvious problem is that the program object is non-persistent and, therefore, has to be created (then destroyed) each time the class is referenced in another JVM. It is imperative, as mentioned in Section 13.9, “Searching and Loading Classes” on page 343, to run the CRTJVAPGM transformer on jar and zip files.

In V4R3, the AS/400 Toolbox for Java (licensed program product 5769-JC1) jar and zip files shipped with their respective Java programs. Prior to this, applications using the AS/400 Toolbox for Java had to repeatedly invoke the transformer, resulting in performance degradation. The solution at that time was to explicitly create the Java program for the files that the applications use. The CRTJVAPGM command accepts a generic, fully qualified /directories/class name, for example, /directory1/directory2/jt400.*. As with most commands, it can also be submitted to the batch environment. We recommend this method since it may take several minutes for the Java programs to be created over large jar and zip files.

13.11.4 Optimization Levels

Optimization is usually a trade-off between performance and size and debug capabilities. For maximum debugging on the AS/400 system, there are three requirements:

- **javac -g** must be used so that debug information is included in the class.
- The java and the class file should be in the same integrated file system directory.
- The class should be running in *INTERPRET mode or at OPTIMIZE(10), although classes at higher optimization levels can still be debugged with less capability.

Otherwise, when the application is stable, it is best to run at the highest optimization level of 40 for best runtime performance, although OPTIMIZE(40) may result in larger Java Programs due to method inlining.

One exception to this guideline is for the AS/400 Toolbox for Java classes for V4R3 alone. Remember this difference between OPTIMIZE(30) and OPTIMIZE(40). OPTIMIZE(30) loads classes as they are first encountered in a JVM. OPTIMIZE(40) loads all external references in a method when that method is called regardless of whether some classes will be bypassed in the logic. Since some AS/400 Toolbox for Java classes contain references to specific browser environments, which may not be available when running on the AS/400 server, "class not found" errors may be encountered under OPTIMIZE(40). Therefore, the AS/400 Toolbox for Java for Java jar and zip files should only be optimized to level 30 in a V4R3 environment. This problem for AS/400 Toolbox for Java classes no longer occurs with V4R4.

13.11.5 Concurrent and Automatic Garbage Collection

Unlike most other JVM implementations, the AS/400 automatic garbage collection is not stop and copy, where other threads stop as garbage collection runs. The AS/400 JVM usually performs garbage collection concurrently or asynchronously without having to stop the other threads. The exception is when the maximum garbage collection heap size is reached, which is described later.

The following garbage collection parameters are specified within the RUNJVA command.

- **Initial size:**

  The JAVA command keyword is GCHINL. The default in V4R2 varies by processor model. The default value starting with V4R3 is 2048.

  This specifies the size that the garbage collection heap will grow before garbage collection starts or restarts. Setting a value too low causes garbage collection to start too frequently on small programs. This is why the default should be used as the minimum size for this parameter.

- **Maximum size:**

  The keyword for this parameter is GCHMAX. This specifies the maximum size that the garbage collection heap can grow to prevent runaway programs from consuming all of the available storage.

  GCHMAX is not the same as the -mx parameter on other Java platforms, where -mx controls both virtual and real main storage consumption. On the AS/400, thanks to its Single Level Storage, GCHMAX largely controls virtual storage only. Using a large number for GCHMAX does not mean that more main (real) storage is consumed by the application.

  The default value starting with V4R3 is GCHMAX(*NOMAX). It was system model specific in V4R2.

  In most situations with the latest release, the best performance can be achieved by using GCHMAX(*NOMAX) and running the Java application in a main storage pool containing other Java applications with the same page requirements. Allocate sufficient main storage so that pool faulting rates adhere to the published guidelines.

  When using a specific value for GCHMAX (other than *NOMAX), and the memory allocation is reached, garbage collection, which normally runs asynchronously, stops all threads while garbage collection takes place. The effect of such an occurrence is noticeable.

- **Frequency:**

  The parameter keyword is GCFRQ. The default is 50. However, this parameter does not have an effect in V4R2 or subsequent releases of the AS/400 JVM.

- **Priority:**

  The parameter keyword is GCPTY. This is described in Section 13.10.2, “Run Priorities” on page 347. The priority parameter is no longer in effect starting with V4R3. Note the various threads in the example in Figure 275 on page 354.

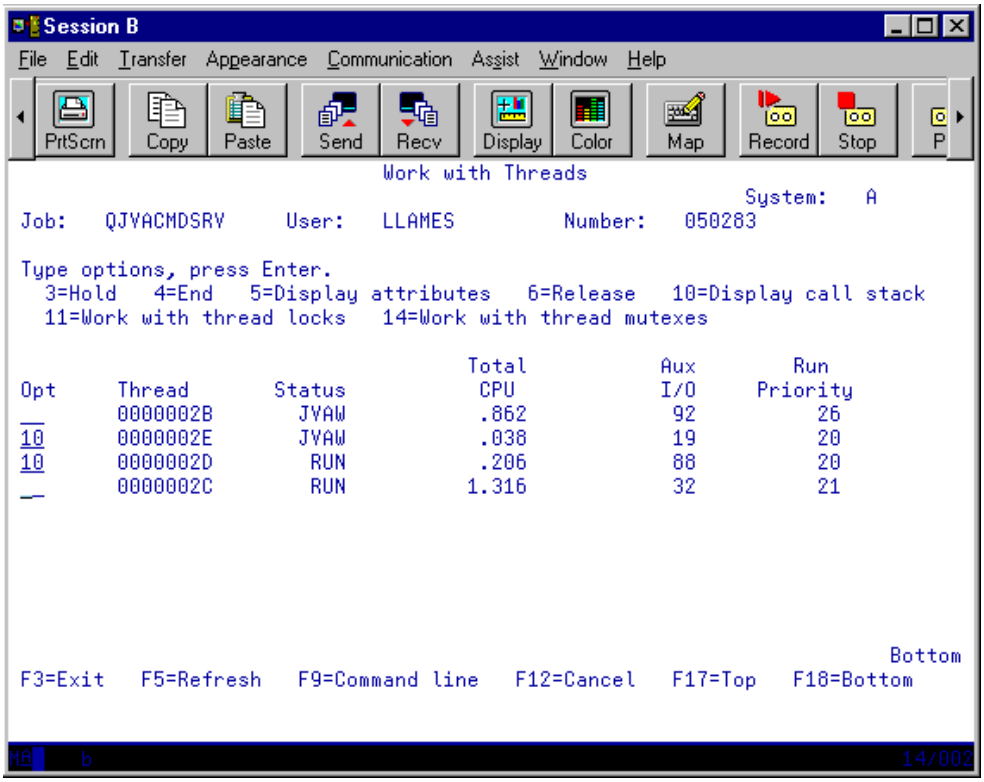


Figure 275. Work with Threads Display

The first thread, with thread number 0000002B is the first Java application thread that was started. It is executing at a run priority of 26. Refer to Section 13.10.2, “Run Priorities” on page 347, for a description of how this priority was obtained.

The next thread assignment, 0000002C represents another thread that this Java application started. For this thread, the setPriority method was invoked to have it run at MAX_PRIORITY, or 10. Note the resulting AS/400 run priority.

There are two threads associated with garbage collection. The example in Figure 275 has them marked with option 10. These threads and their functions are, at a high level:

- **Collector Thread**: Locates objects that are no longer referenced.
- **Finalizer Thread**: When an object collected by the garbage collection, this garbage collection thread invokes the FINALIZE method for the object, if such a method was defined.

Under V4R2, these GC threads are started immediately after the initial thread described in Section 13.10.1, “Initial Thread” on page 347. Their sequential thread assignments reflected that order.

From V4R3, not only is the initial thread removed, but the garbage collection threads are not started right away. The Collector Thread is started when the GCHINL parameter is reached. The Finalizer Thread is started if and when a finalize method is invoked.

This explains the difference in the thread assignment order between a Java application running in V4R2 and an application running in subsequent releases.

Chapter 14. Java Application Design and Host Performance Analysis

In Chapter 13, “Java Performance and Work Management Overview” on page 323, several techniques to improve Java runtime performance were presented. These applied more towards work management and system tuning, or steps that can be completed when installing and running a Java application on the AS/400 system.

There is another area where performance can be significantly improved. This falls into the realm of application design. This chapter explains the relationship between Java application design and host performance.

14.1 Java Design

Because Java is a portable language, you hope that the coding techniques that work well on one platform are also beneficial on other platforms. In many ways, this is true. Even before the advent of Java, seemingly dissimilar systems had certain similarities in terms of what operations tended to be very expensive. These specifically pertain to any operations that required the creation and destruction of anything. For example, the creation and deletion of a file or table, the creation and destruction (initiation and termination) of a job, the opening and closing of a file (results in creation and destruction of an Open Data Path), are all relatively expensive. In this chapter, we discuss certain techniques that are available in the Java environment that tend to result in the instantiation and deletion of objects. Fortunately, there are alternatives to minimize object creation and deletion.

Application design primarily affects the areas of Java instruction execution and access to system services. The subject of application design has broad coverage. However, this chapter presents design and coding considerations that apply to Java runtime performance and various recommendations for accessing DB2/400. Note that since Java on the AS/400 system is expected to play a significant role in the e-business environment, access to the database has a much larger significance than in non-commercial applications.

14.2 Performance Measurement

Another portion of this chapter covers one way of analyzing Java applications on the AS/400 system. This method of analysis is specific to the AS/400 system and does not apply to other platforms. Typically, analysis of a Java application involves the use of the java -prof[:<file>] option, where data involving Java object and method usage is sent to a file. The file contents can then be analyzed with visual analysis tools, which highlight the most expensive portions of a Java application. These tools are available from various sources. One such tool that is available is Jinsight, which currently can be downloaded from the Web at: http://www.alphaworks.ibm.com

However, the -prof option is not supported on the AS/400 system. When an application that uses the -prof option is deployed on the AS/400 system, for example, to run in QSHELL, the option is merely ignored. Where it is available on other platforms, the -prof option does not provide data on the performance

characteristics of system services such as database access because it only handles Java instruction execution.

It is possible to use visual analysis tools for Java applications that are running on the AS/400 system. There are two of such facilities that are described in Chapter 15, “Application Performance Analysis with GUI” on page 395.

On the AS/400, performance measurements of a Java application can be done using the Performance Explorer (PEX) facility. All AS/400 RISC systems have the measurement facilities available through the base operating system. Reports can be produced with the Performance Tools program product, 5769-PT1. However, the existing Performance Tools reports provide limited capabilities for analyzing AS/400 Java application execution. They are also text-based and, therefore, are not as user friendly.

14.3 Instruction Execution — Coding Considerations

There are certain programming techniques that result in high amounts of overhead. These techniques appear to be costly because they result in object creation and deletion, as well as other overhead that is inherent to an object-oriented design. Techniques such as these are finally showing up as more Java applications are built. Since we are in the relatively early stages of Java application development, coding is generally done with functionality in mind. Performance is generally an afterthought. However, certain programming techniques are being discovered to have more overhead than desired and, therefore, should be minimized. It is also possible to carry something to an extreme, for example, creating tens of thousands of objects when one is sufficient.

14.3.1 Instantiation = Garbage Collection

Instantiation is the creation of new objects. This is expensive because it results in the allocation of heap storage. As expected, instantiation eventually results in garbage collection since the objects that were created eventually have to be cleaned up when they are no longer referenced. The more complex an object is, the more overhead it takes to instantiate it.

Instead of creating a new instance of an object each time with the "new" operation, it may be possible to reuse an object by varying its attributes or instance variables. Obviously, unnecessary object instantiation needs to be eliminated. In the early version of our sample application, one of the reasons behind the slow startup of any RMI related transaction was caused by the repeated and unnecessary instantiation of the RMISecurityManager. The use of the lazy initialization technique to determine if an object needs to be created made it more efficient:

```
if (securityManager == null)
      securityManager = new RMISecurityManager;
```

This way, the securityManager object is created only if it does not already exist.

14.3.2 String Manipulation = Instantiation

The problem with string manipulation is that strings are immutable. This simply means that the Java specifications dictate that strings are not capable of change. Therefore, changes to the value of a string (for example, assignments or concatenation) result in the old string object being discarded and a new string object being created to contain the new value. String manipulation can then result in large amounts of object instantiation and garbage collection.

A technique for reducing this overhead is to use a StringBuffer object instead of a string. The StringBuffer is not immutable and has several methods, one of which is StringBuffer.append. Appending to a StringBuffer is a much better performer than string concatenation. However, being capable of change means that there is a corresponding synchronization overhead associated with this object type. An even faster technique for variable string values is to use an array of characters (char[]) to behave like a fixed-length string.

14.3.3 Method Resolution

Properly designed object-oriented applications are much more modular than procedural implementations. This results in more frequent method resolution, or the invocation of methods that do not exist in the class that performed the request. Instead, the methods can represent behaviors that are inherited from a super class. When the method is called, the program searches for it within the current class. If it is not found, the method is searched for externally. Then, the external method is resolved and invoked, with the expected arguments passed to the external method. Obviously, this process is more expensive than if the method was located in the same class that called it.

One way to reduce this is to specify methods as final, when possible, and then to compile them with the javac -O option. This causes *method inlining*. This means that the method call is replaced with the actual method instructions (bytecode) being copied into the subclass that requests it. This eliminates the need for the application to resolve or search for the method and to access the external method. This can result in substantial runtime improvements if method resolution is a large portion of the application overhead. Note, however, that this will result in larger class sizes for the affected classes because a copy of the method's bytecode is now embedded within the class. In V4R4, creating a Java program under OPTIMIZE(40) performs method resolution without requiring the javac -O option.

14.3.4 Synchronized Methods

This keyword is used to enforce dependencies among threads. It is not necessary if the threads are actually independent of each other. Use of synchronized methods should be minimized since it can cause unnecessary overhead and the locking and unlocking of objects.

However, data integrity is usually more important than pure performance. There will always be situations in a multi-threaded environment when threads have to be synchronized.

14.3.5 Data Conversion

Database servers generally store data using a different encoding scheme from Java. The AS/400 JVM stores string data as 2 byte Unicode. If the database is encoded in EBCDIC, the data will be converted on every database access. This can be avoided by storing the data in DB2 as 2 byte Unicode, using the SQL graphic type with CCSID 13488 or the DDS graphic type with CCSID 13488.

However, in the real world, large amounts of data exist in EBCDIC form. To minimize such conversions, access only the fields or columns that are needed by the application. For example, instead of "SELECT * FROM custmaster. . .", specify the needed columns, as in "SELECT custnumber, custname FROM custmaster. . .."

Store numeric data in DB2 as a float to reduce numeric conversions. Decimal data cannot be represented in Java as a primitive type. Decimal data is converted to the abstract class Java.lang.BigDecimal on every database read or write. This conversion can be avoided by storing numeric data in the database as float or double. Alternatively, you may consider converting the BigDecimal type to float for heavy calculations while leaving the database in decimal form. Note that rounding differences may be introduced with the use of float or double.

14.3.6 Exception Handling

Using exception handling mechanisms such as "throws exception" is relatively expensive. Using exception handling mechanisms for regular processing should not be done. They should be used only in real exception conditions.

Try-catch structures are very efficient. It is far more efficient to catch the occasional exception than to pass return codes and evaluate them.

14.3.7 Java Native Interface (JNI)

The implementation of the JVM is fully integrated into the OS/400 Machine Interface. Frequent accesses of trivial non-Java programs through JNI should be minimized since this requires processing above the Machine Interface and tends to slow the application down.

As of V4R4, a safe JNI call can be made to ILE RPG and ILE COBOL programs since they were made thread-safe. Previously, only C programs were recommended for JNI calls.

However, calls to longer running non-Java programs may be tolerable because the overhead of making the call is small compared to the total amount of productive work done.

14.3.8 Thread Creation

The AS/400 system kernel thread implementation is very efficient. Thread creation is considered a lightweight operation. Threads within a process can re-use heap pages among themselves. However, extreme designs have also been observed, for example, one process with tens of thousands of threads. Although the AS/400 system is more than capable of handling such designs due to its multi-tasking heritage, unnecessary thread creation simply adds up to additional overhead.

One area where threads can really be useful is in reducing the perceived response time of a transaction. Instead of serially processing a long running transaction, other threads can be spawned to multi-task the transaction.

14.3.9 Variable Scope

The scope of variables has different costs. Access to a local variable is the fastest. Direct access to an instance variable or using an inlined accessor method have approximately the same cost. Both are more expensive than local variable access. Using an accessor method (not in-lined) is even more expensive. The most expensive is to use a Synchronized accessor method.

14.3.10 Remote AWT

Because the AS/400 server is not designed to generate and display graphics, method calls to AWT or Swing methods are routed to a graphical client where they belong. This is done through an AS/400 mechanism called Remote AWT.

In V4R2, Remote AWT was implemented internally through Remote Method Invocation (RMI). Significant performance improvements were attained in V4R3 when the RMI layer was removed and a direct sockets connection was used. Additional enhancements were made for V4R4.

Remote AWT is not designed for high-volume transaction processing. Think about the amount of traffic that has to flow through the network, including events such as mouse moves, mouse clicks, and so on. The GUI portion of an application should run on the client where it belongs, not on the server to be handled by Remote AWT.

However, for low-volume transactions, such as installing an off-the-shelf package, the network traffic is not as significant so Remote AWT usage is not a problem.

14.4 Accessing System Services — Database

In Chapter 13, “Java Performance and Work Management Overview” on page 323, we described that in an typical commercial environment, there are two major areas where processing overhead is incurred. One area is in running the Java program instructions, as described in the previous section. The other is in the access of system services. In most cases, the majority of the system services are represented by database operations.

Some of the recommendations implemented herein are specific to the AS/400 server platform and may not apply to other servers. For example, record level access through DDM and Distributed Program Call (DPC) are done through the AS/400 Toolbox for Java and apply only to the AS/400 system. Other recommendations pertaining to Java Database Connectivity (JDBC) may apply to other relational databases.

Figure 276 on page 360 shows various ways of accessing the relational database or existing programs on the AS/400 system.

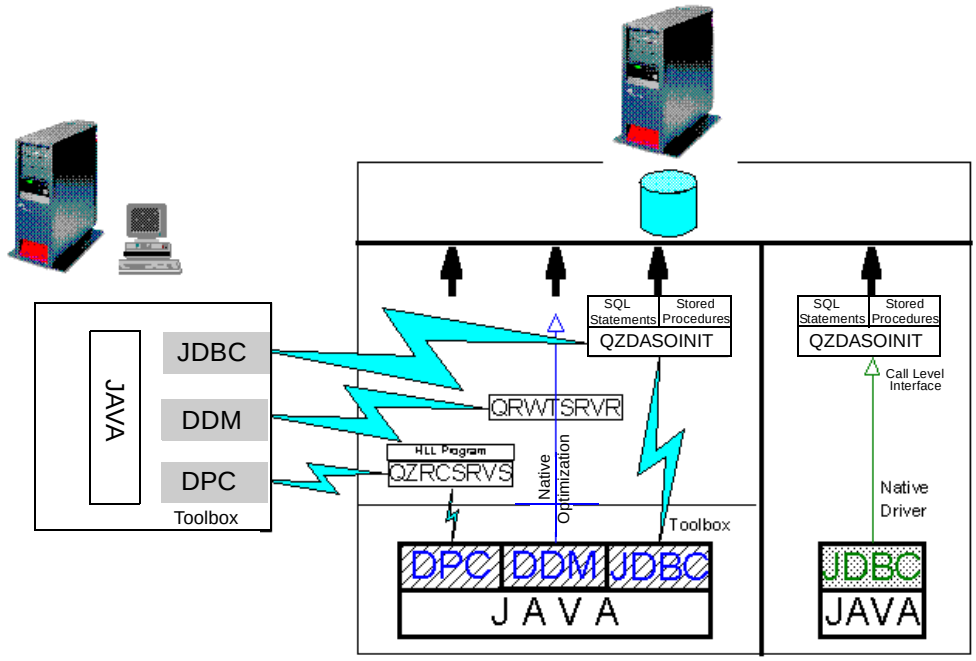


Figure 276. Common Access Methods to Existing Programs and Data

In Figure 276, the left portion represents a Java application running on a client system and using the AS/400 Toolbox for Java classes to access the AS/400 system. Since the AS/400 Toolbox for Java classes are 100% pure Java, this application is portable to any compliant JVM. However, the server that it accesses has to be an AS/400 server.

The middle portion represents a Java application running on the AS/400 system using various ways of accessing system services. These include:

- The DPC model is designed for client-server work so the Java application communicates through the QZRCSRVS server job to access and AS/400 program.
- The DDM or record-level access model has the AS/400 Toolbox for Java classes "optimized for native." This means that if the AS/400 Toolbox for Java classes determine that access is being done locally, there is no need to go through a server job or through sockets. The Java application accesses the database directly.
- This JDBC approach uses the AS/400 Toolbox for Java JDBC driver.

The right portion in Figure 276 shows a server Java application using the "native" JDBC driver.

Regardless of which driver is used, JDBC can access DB2 through SQL statements or through stored procedures. The AS/400 system is unique in the industry because stored procedures do not have to contain embedded SQL statements. Even traditional 3GL programs using native I/O operations can be used as stored procedures.

14.4.1 AS/400 Toolbox for Java Driver versus AS/400 Developer Kit for Java Driver

As described previously, there are two JDBC drivers available when running a Java application on the AS/400 system. One is the native driver or the AS/400 Developer Kit for Java driver, com.ibm.db2.jdbc.app.DB2Driver. The other is the AS/400 Toolbox for Java driver, com.ibm.as400.access.AS400JDBCDriver. Some of the properties may vary, but the SQL functionality is mostly similar. The AS/400 Toolbox for Java driver is a type 4 driver and provides network access for clients that are physically separated from the database. The AS/400 Developer Kit for Java driver is a type 2 driver that provides direct access.

Other differences are described in *The AS/400 Developer Kit for Java*. From a performance standpoint, there are more significant differences.

14.4.1.1 AS/400 Toolbox for Java Driver Performance

The AS/400 Toolbox for Java driver was originally designed to let a client access DB2/400 in a client/server environment through a network. It uses a sockets connection between the client and the server. The server job to which the JDBC requests are submitted is named QZDASOINIT. This is the same server job name used to process 32-bit client requests in the ODBC environment.

The AS/400 Toolbox for Java driver allows SQL Extended Dynamic support, which allows SQL statements and their access plans to be stored in objects of type SQL Package (*SQLPKG) on the AS/400 system. This provides a significant performance benefit for repeatable SQL statements because of the reduced parsing and syntax checking overhead. Note that one performance difference between the AS/400 Toolbox for Java JDBC environment and ODBC is that in the latter, Extended Dynamic mode is false, by default.

However, when used in a server Java application within the same AS/400 system that contains the data, the current implementation does not "short circuit" the sockets implementation. This means that requests from a server Java program go through the sockets interface before accessing DB2/400. The result is additional latency as well as additional processing overhead.

14.4.1.2 AS/400 Developer Kit for Java Driver Performance

The AS/400 Developer Kit for Java driver, on the other hand, was designed to process SQL requests from a server Java program to access DB2/400. It uses the Call Level Interface (CLI) standard to submit SQL requests. CLI is like ODBC running natively on the AS/400 system, with essentially the same APIs. It produces dynamic SQL statements, as do the ODBC or Toolbox JDBC environments. The difference is in the way that SQL statements are cached.

Past experience with dynamic SQL in various computing platforms brings back memories of poor performance. Most of the problem was due to statement parsing and syntax checking. However, starting with OS/400 V4R2, an internal system-wide SQL statement cache was implemented, which significantly improves the execution of dynamic SQL. This cache stores dynamic SQL statements for much improved subsequent execution.

Currently, there is a maximum of 500 concurrent statement handles per connection. In the unlikely situation that more than 500 are required, another connection can be established.

14.4.1.3 Which One to Use

When running Java on the AS/400 system, we recommend that you use the AS/400 Developer Kit for Java driver or "native" driver, both for server efficiency and total response time.

Use the AS/400 Toolbox for Java driver when issuing requests from a client to the server. However, switching from one driver to another is a simple change, usually affecting only one class. Using one over the other in the server application does not mean that the application needs to be redesigned if the driver needs to be switched.

14.4.2 JDBC through Native Driver

When a Java program on the AS/400 system performs SQL requests to DB2/400, the AS/400 Developer Kit for Java JDBC driver issues dynamic SQL requests through the CLI standard. Under the CLI implementation, the SQL requests are not performed by the Batch Immediate (BCI) job. They are sent to a separate pre-started job named QSQSRVR. When Java establishes a connection to the DB, a message is sent to the BCI job's log:

```
123456/QUSER/QSQSRVR used for SQL server mode processing.
```

When analyzing the database, monitor the QSQSRVR job for optimizer messages. This is done by placing the job in service mode and debug mode (using the STRSRVJOB command followed by STRDBG). Its job log then records the optimizer messages.

The Database Monitor can also be used to monitor this job (using the STRDBMON and ENDDBMON or the STRPFRMON commands). This monitor provides much more information than when observing the job log in debug mode. Actual SQL statements are recorded.

Another option for analyzing SQL requests is the SQL Monitor available in the V4R4 Operations Navigator.

14.4.2.1 Statement Cache for Dynamic SQL

There are three kinds of SQL statements that can be issued on the AS/400 system. They are:

- Static SQL
- Dynamic SQL
- Extended dynamic SQL

Static SQL is the most efficient, but is currently not available to a server Java application unless it is issued from a stored procedure. Future Java releases may provide this option in the form of SQLJ. However, current implementations of SQLJ provide "ease of coding" advantages rather than performance advantages.

Static SQL, as the term indicates, is assumed to be constant. The statements and the access plans are stored in the program object that contains the embedded SQL statements. These are still the best performing SQL statements because there is no need to parse, check the syntax, and possibly create the access plan for every execution of the statement.

Extended Dynamic SQL (strictly speaking, these can be considered a form of dynamic SQL statements), when issued from a client, has a fairly good

performance profile, close to that of static SQL. The reason is that SQL statements are stored in an SQL package (*SQLPKG) object. However, as mentioned earlier, the AS/400 Toolbox for Java JDBC driver is not optimized for requests from Java applications within the server even though it generates Extended Dynamic.

Starting with V4R2, an SQL statement cache is implemented internally. This statement cache stores dynamic SQL statements so that subsequent executions of the same statement run much faster. Since this is a system-wide cache, a statement that is executed in one job also benefits other jobs. Using the statement cache is automatic and is available to all applications that generate dynamic SQL, not just server JDBC requests. Nothing needs to be done to activate it. The SQL statement cache is cleared at IPL time. Unlike other platforms where the SQL statement has to be an exact match (including spaces), the AS/400e cache is more tolerant of differences and can also handle parameter marker replacement. Not only are the statement strings stored, but other internal structures such as access plans are stored as well, reducing the need to re-create them each time the statement executes.

14.4.2.2 Repeatable Statements

Even with the statement cache, it is still necessary for performance reasons to prepare (prepareStatement) SQL statements prior to executing (executeQuery or executeUpdate methods) them repeatedly. If the repeatable statement is not prepared beforehand, the executeQuery causes an implicit prepareStatement to occur every time the statement is run. Therefore, the overhead will be higher without an explicit prepare.

14.4.2.3 DB Tuning

In an online transaction processing environment (OLTP), the query runtime should be very short. The database should then be tuned properly. Tables should have the proper indexes and the SQL statements should be written properly to use these indexes. The proper level of normalization and the use of the SMP feature also helps.

In analyzing JDBC environments, the debug feature or the DB Monitor should be used to determine the amount of time spent in running the queries. These facilities also provide recommendations on which indexes need to be created.

Other ways to monitor and analyze optimizer messages are to put the SQL server job in debug mode, and the SQL Monitor within Operations Navigator.

Proper DB tuning also improves the probability of the application generating reusable ODPs. Generally, using temporary objects, such as indexes or tables, results in non-reusable ODPs.

14.4.2.4 Stored Procedures CLOSQLCSR

For stored procedures that are called repeatedly within the same connection, the ODPs should be kept open by specifying the proper CLOSQLCSR parameter such as *ENDJOB or *ENDACTGRP.

14.4.3 DDM (Record Level Access)

There is a significant difference between SQL access and record level access. In the former, the programmer specifies what to retrieve. Then, the optimizer decides how to retrieve it. Where in the latter, the programmer decides how and what to retrieve. Therefore, efficient data retrieval logic is important in record level access. This problem is not new in Java. It has persisted since the first RPG, COBOL, or C programs were written.

A common scenario is reading multiple records and checking a status flag until the required one is found. It is much more efficient to include such logic in the database, for example, as an index with the proper selection criteria. That way, only the needed record is read.

14.4.3.1 Minimizing Open and Close

Repeated execution of the open() and close() methods over the same file is expensive and unnecessary. The program logic should be implemented so that a file is opened only once and kept open within the Java application.

14.4.3.2 Blocking on READ_ONLY and WRITE_ONLY

A file that is opened for input-only can take advantage of blocked input when it is appropriate. Note the following example:

```
myKeyedFile.open(AS400File.READ_ONLY, 100,
            AS400File.COMMIT_LOCK_LEVEL_NONE);
```

The attribute of 100 is the blocking factor, which means that the first read() operation retrieves as many records in a single operation. Subsequent invocations of read() retrieves data from the buffer instead of going back to the database. A similar benefit can be derived from a file that is opened with a WRITE_ONLY attribute.

14.4.3.3 Downloading a Small File Using the ReadAll Method

When retrieving the data from a small file, the readAll() method is more efficient. All of the records in the file are retrieved in one method call. The client should have enough resources to contain the entire file.

14.4.4 Distributed Program Call (DPC)

DPC may be used to call to existing RPG, COBOL, or C programs. A typical performance pitfall with such languages is that the programs and their files are usually closed down when they are exited. For a repeatedly called RPG program, the LR indicator should not be set on.

For OPM COBOL, it may not be possible to keep the programs and files open. Running an EXIT PROGRAM or GOBACK from a COBOL main program ends the run unit, shutting everything down. However, the ILE COBOL/400 environment will perform much better. In a recent release, an IBM extension was added to the EXIT PROGRAM operation. The EXIT PROGRAM AND CONTINUE RUN UNIT clause allows the COBOL run unit to remain active. It keeps the files and programs open for subsequent calls.

14.5 Baseline Measurement

The following description explains the base script and the response times that were observed. The version of the application that was measured is the new object-oriented version that is made up of three logical tiers:

- Presentation layer
- Business logic layer
- Persistence layer

Among the different persistence methods, the one that was measured is the JDBC method because it is portable and, therefore, is expected to be the most widely used. The test was done on a uni-processor Model 620-2181, which has a CPW rating of 210.

As indicated in Chapter 13, “Java Performance and Work Management Overview” on page 323, measuring and recording end-user response times in a client/server environment is always a challenge. The reason is because the server tools do not have any indication of what actually happens at the client end. To obtain the end-to-end response time, the three GUI classes on the client were modified to calculate the elapsed time for transactions that involved communicating with the server. For example, each of the window classes were modified to have additional properties of elapsedTime, as shown here:

```
public class OrderEntryWdw . . .
            .
            .
            .
      static float startingTime;
      static float elapsedTime;
```

To calculate the elapsed time, code was inserted within the class' method that performs the work. For example, within the OrderEntryWdw class' submitOrder() method:

```
public void submitOrder( ) {
 updateStatus("Processing order ...");
 startingTime = System.currentTimeMillis();  // Mark beginning of
                                            //  transaction.
 orderPortfolio.submitOrder(order);
 elapsedTime = (((System.currentTimeMillis() -
               OrderEntryWdw.startingTime)) /
              1000);   // Calculate response at end of transaction.
 System.out.println("submit order took = " + elapsedTime + " seconds.");
 updateStatus("Order successfully processed!");
```

Note: Some of the initial transactions (for example, signing on) are only performed at the beginning of the day and may not be critical.

Several observations were noted regarding the baseline measurements. Generally, those system transactions that involved only the client were much faster than those that required collaboration with the server. This is not surprising in a client/server application. Any access to the server means additional network time, as well as additional latency at the server end. This is explained in detail in Chapter 13, “Java Performance and Work Management Overview” on page 323.

There are two major performance objectives for the application:

- Improve the end-user response time.
- Improve the server components to increase its ability to scale.

Figure 277 shows the data access methods that are used within the application.

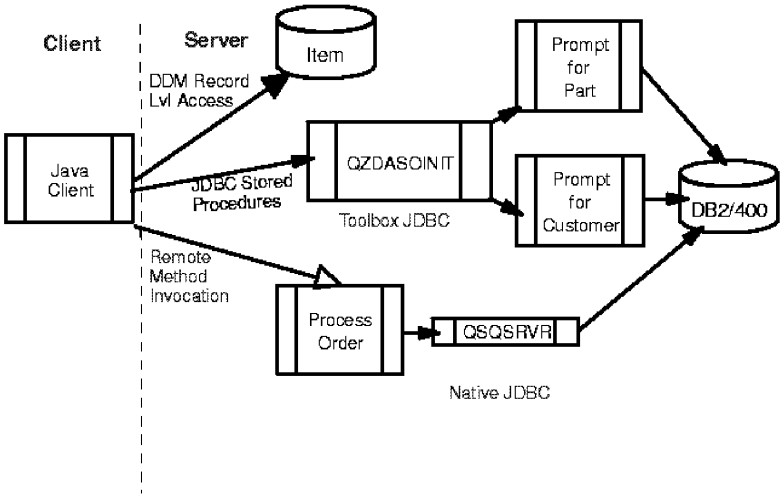


Figure 277. Application Data Access Methods

14.5.1 Transaction Script — First Order

In this section, we cover the transaction script that was used to measure response time. We ran the following steps in this scenario:

1. Start the RMI registry on the AS/400 system.
2. Start the RMI application on the AS/400 system.
3. Start the client Order Entry application.
4. Sign on to the AS/400 system.
5. Retrieve a list of customers.
6. Select a customer.
7. Retrieve a list of available items.
8. Select an item.
9. Enter an order quantity for the item.
10. Add the item and quantity to the order list box.
11. Submit the order (first order).
12. Retrieve a list of customers.
13. Select a customer.
14. Retrieve a list of available items.
15. Select an item.
16. Enter an order quantity for the item.
17. Add the item and quantity to the order list box.
18. Submit the order (second order).
19. Sign off.

In the following list, we show and explain the measurements recorded for the first time we ran the scenario. Then, we review what we did to improve the response time. In Section 14.10, “Summary and Recommendations” on page 393, we show the key response times recorded for several different versions of the application running the same application scenario.

1. 12:37:00 Start RMI Registry interactively. Response = 13 seconds:

   The registry was started from a 5250 session using the following command:

   ```
   JAVA CLASS(sun.rmi.registry.RegistryImpl) PARM(1234) OPTION(*VERBOSE)
   ```

   This only needs to be run once per port. Once it is set, multiple users can go through the same registry.

   This is a never-ending process and should normally be done in background mode. There are several reasons for this, for example, not having to waste a display session and minimize the 5250 workload to make the application a better fit for the e-server models. Most of the overhead incurred by the interactive session involves creating the temporary data queue, starting the Dynamic Screen Manager, and then displaying the output. Submitting this to batch also makes it easier to control the work management environment.

   The PARM(1234) represents the port number that is used by Remote Method Invocation.

   Note: The *VERBOSE option was used. In a production environment, this is normally not used because of the high volume of class loading messages that it generates between the spawned BCI job and the job that spawned it, in this case, the interactive session. The *VERBOSE mode certainly adds overhead. However, the emphasis in this measurement is to help identify long running (class loading) portions of the application and improvements that can be made.

   Note: The registry startup time is long. However, this does not have to affect the interactive users directly since this step can be done once at the start of day, for example, when the subsystem is started.

   In this example, the following two jobs involve the RMI registry:

   - Job: QPADEV000A User: LLAMES Number: 001241
   - Job: QJVACMDSRV User: LLAMES Number: 001252

   A 5250 session, such as QPADEV000A, is the interactive session from which the registry was started. It is used to display stdout. It runs at the default interactive priority of 20.

   QJVACMDSRV is a BCI job that runs the RMI Registry itself. Because it was spawned by a priority 20 interactive session, this BCI job's Java threads run at priority 26 (refer to Chapter 13, “Java Performance and Work Management Overview” on page 323, for run priority assignments).

2. 12:39:00 Start OrderEntryJdbcRMIPersistencyManager. Response = 47 seconds:

   This program was run under an interactive QSHELL session. After setting the CLASSPATH appropriately, the following command was issued:

   ```
   java -verbose PersistencyManagers.OrderEntryJdbcRMIPersistencyManager
    systemName 1234
   ```

   After many classes were searched, verified, and loaded, the application issued the message: `Main: Successfully registered with the security manager`. This never-ending process needs to be run only once and can be shared by many users. For every user that needs to use it, a separate thread is started so there is no queuing similar to that experienced in the Multiple Requestor Terminal (MRT) programming technique.

This step started the server portion of the application. Similar to the RMI Registry, this should be submitted to batch because it is a never-ending process. However, it was run interactively and under verbose mode so that the class loading steps are visible in realtime.

Similar to the RMI Registry, the startup time for this server is very long. However, this does not have to affect the interactive users directly since this step can be done once at the start of day, for example, when the subsystem is started.

The following jobs are involved:

- Job: QPADEV0003 User: LLAMES Number: 001240
- Job: QZSHSH User: LLAMES Number: 001253
- Job: QSQSRVR User: QUSER Number: 000681

QPADEV0003 is the 5250 session from which the QSHELL interpreter was started. It interactively displays the class loading messages (due to -verbose) as well as other stdout messages. From the messages, it is apparent which of the groups of classes are taking a lot of time to load.

QZSHSH performs the work submitted through the QSHELL interpreter. In this transaction, the work involves a never-ending process Java application OrderEntryJdbcRMIPersistencyManager. Because this class performs server JDBC requests to DB2/400, it links up with one of the pre-started Call Level Interface (CLI) database servers. The specific server is identified in this job's log through the entry shown in Figure 278.

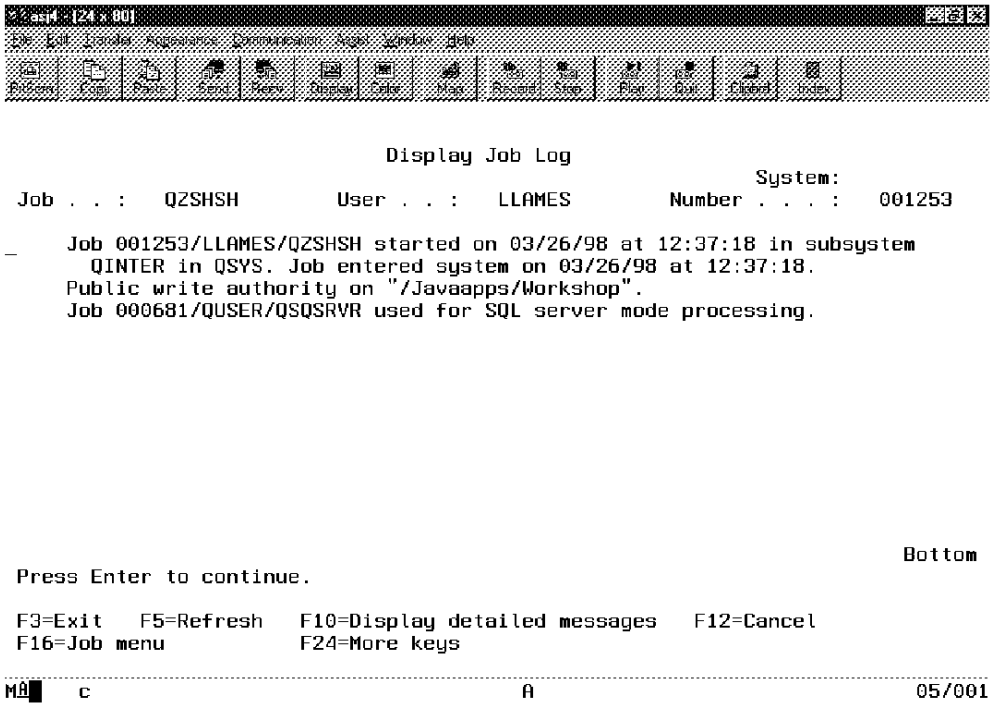


Figure 278. SQL Server Job Log Message

This QSQSRVR job performs all of the DB requests from the server Java application. When analyzing the state of the DB through debug mode or

through the DB monitor, this job needs to be monitored. Note all of the application files that are open within this job.

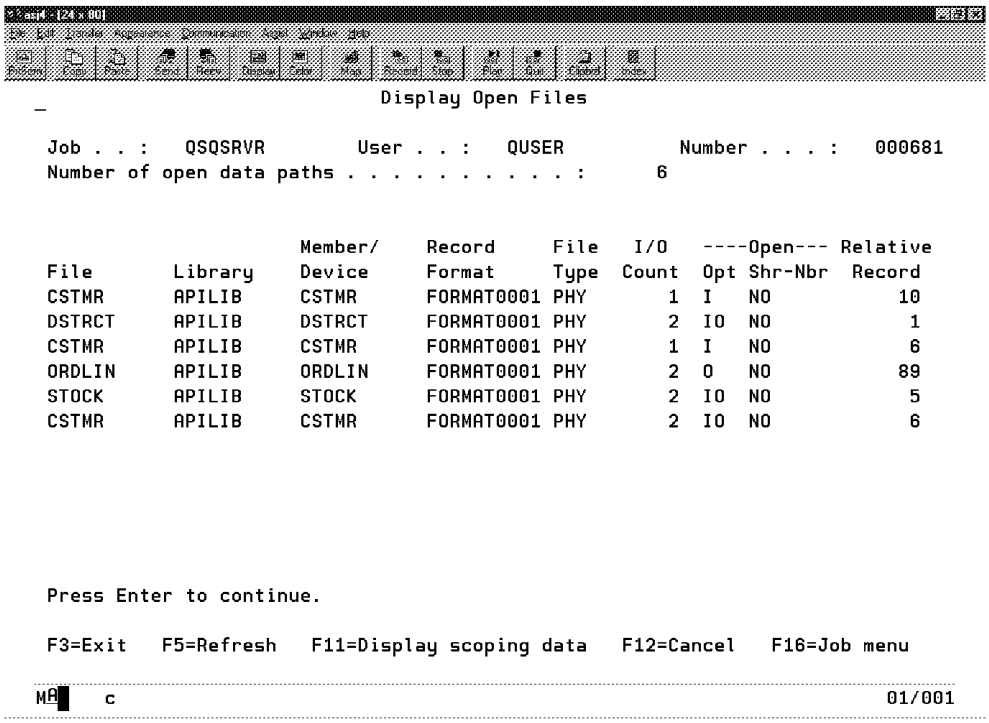


Figure 279. Order Entry Open Files

3. 12:41:00 Start OrderEntryWdw Response = 7 seconds:

   This entire transaction occurs within the client. As in the AS/400 system, the client JVM is started, the required classes are loaded, and new objects are instantiated.

4. 12:43:00 Sign On, click on OK. Response = 45 seconds:

   ". . . Connected to asj4"

   This transaction connects to the AS/400 system through the AS/400 Toolbox for Java.

5. 12:45:00 List Customers button. Response = 7.145 seconds:

   "Customer list retrieved"

   This request goes through the AS/400 Toolbox for Java JDBC driver. A stored procedure, SLTCUSTR, is used. Therefore, the statement and the access plan are stored as part of the program object.

6. 12:46:00 Select a customer, OK:

   This transaction affects the client only. There is no network or server time involved.

7. 12:47:00 List Items button. Response = 15.189 seconds:

   This submits a JDBC stored procedure request to the AS/400 system. The stored procedure is SLTPARTR. Currently, all of the items are retrieved. In a production database that can contain many more items, this results in poor response times.

To improve its performance, the result set should be limited to perhaps two pages worth of data. This is analogous to the recommendation for 5250 sub-files. The GUI should include a prompt to position to a specific item.

8. 12:48:00 Select an item, OK:

   This affects the client only. There is no network or server time involved.

9. 12:49:00 Type Quantity = "2", click on Add Item button:

   This affects the client only. There is no network or server time involved.

10. 12:50:00 Submit Order button. Response = 94 seconds:

    "Processing order . . ."

    "Order successfully processed"

    The majority of the response time in this transaction appeared to be caused by class loading within the server portion of the application. Because it was run under -verbose mode, class loading messages were generated. The classes within the package com/ibm/as400/access/ had long class load times. This package contains the classes for the AS/400 Toolbox for Java.

14.5.2 Transaction Script — Second Order

Continuing the sequence from the first order, this list shows the measurements obtained in the second order:

11. 12:53:00 List Customers button. Response = 6.99 seconds:

    "Customer list retrieved"

    This is similar to the preceding transaction #5.

12. 12:54:00 Select a customer, OK:

    This is similar to the preceding transaction #6.

13. 12:55:00 List Items button. Response = 14.416 seconds:

    "Item list retrieved"

    This is similar to the preceding transaction #7.

14. 12:56:00 Select an item, OK:

    This is similar to the preceding transaction #8.

15. 12:57:00 Type Quantity = "2", click on Add Item button:

    This is similar to the preceding transaction #9.

16. 13:01:00 Submit Order button. Response = 5.327 seconds:

    "Order successfully processed"

    Compare this to the functionally equivalent transaction #10. Note the large difference in response times. This is explained later.

17. 13:03:00 Disconnect menu option. Response = 10 seconds.

18. 13:04:00 Exit button.

14.6 Analysis and Improvements

Several improvements were made to the application. This was done in an iterative fashion, starting with the changes that were expected to provide the largest improvement with the lowest effort. As in any complex system, it is possible to make continuous improvements. However, due to the limited time available, a finite number of improvements were made.

14.6.1 Transaction #10, First Submit Order

This section analyzes the submission of the first order. It is a very long running transaction and we need to improve the end user response time for it.

14.6.1.1 Creating a Persistent Java Program for jar and zip

Based on the recommendations in Chapter 13, “Java Performance and Work Management Overview” on page 323, it was apparent that the AS/400 Toolbox for Java classes did not have Java programs built. Remember that for class files, the associated Java program, if missing, is transformed at first load and kept persistent. However, if the classes are packaged in jar or zip files, the transformed Java program is not persistent.

The AS/400 Toolbox for Java classes are packaged in both a jar and a zip file. Either copy can be used in the CLASSPATH. To check if they have Java programs pre-built, the following commands were run:

```
DSPJVAPGM CLSF('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.zip')
DSPJVAPGM CLSF('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar')
```

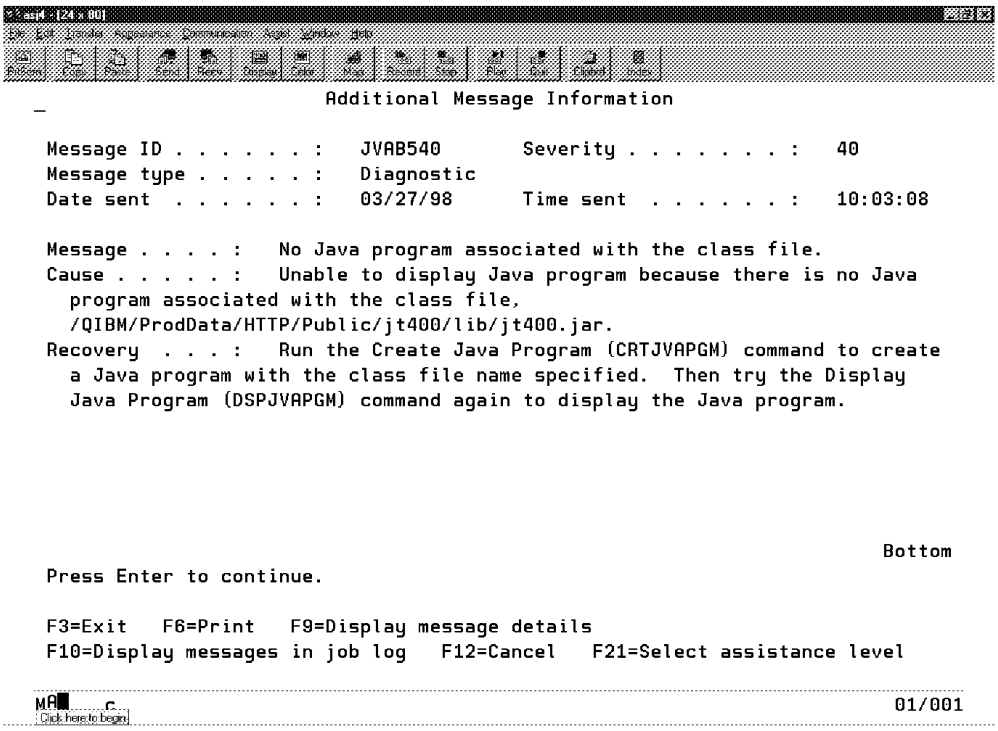


Figure 280. Message — No Java Program Found

A separate Performance Explorer trace was run on the same transaction. The report shows that the majority of the processing time was spent on the following operations with the first entry having a higher contribution:

```
*CRTJVPG        = CREATE JAVA PROGRAM
CLLOD   =  Java Class Load
```

This explains why class loading was slow for those classes within the com/ibm/as400/access/ package.

To improve on this, Java programs were created for both the jar and the zip files. It may not be necessary to do this for both if it can be ensured that only one (for example, the jar file) is used in the CLASSPATH of all server Java applications. The following commands were used to create the Java programs separately:

```
CRTJVAPGM CLSF('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.zip')
OPTIMIZE(40)
CRTJVAPGM CLSF('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar')
OPTIMIZE(40)
```

In V4R3 and V4R4, the AS/400 Toolbox for Java is shipped with the Java programs already created.

It is also possible to use a generic class file name, as shown here:

```
CRTJVAPGM CLSF('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.*')
OPTIMIZE(40)
```

For the jt400.jar and jt400.zip files, it took approximately 45 minutes to create the Java programs, which makes this type of work more suitable for the batch environment. It is imperative to run this in non-interactive mode when using a server model, a Model 170, or a system with a low interactive feature since these systems are subject to penalties with significant 5250 work loads. If the jar or zip files are modified, the Java program has to be created again. Note that it may be necessary to repeat the creation of the Java program if the jar or the zip files are modified (for example, upgraded through a corrective process).

For the zip files containing the Java Runtime Environment (JRE), that is, java.zip and sun.zip, their Java programs are retained during the corrective PTF process. This is achieved by installing replacement classes into a different zip file that appears earlier in the classpath.

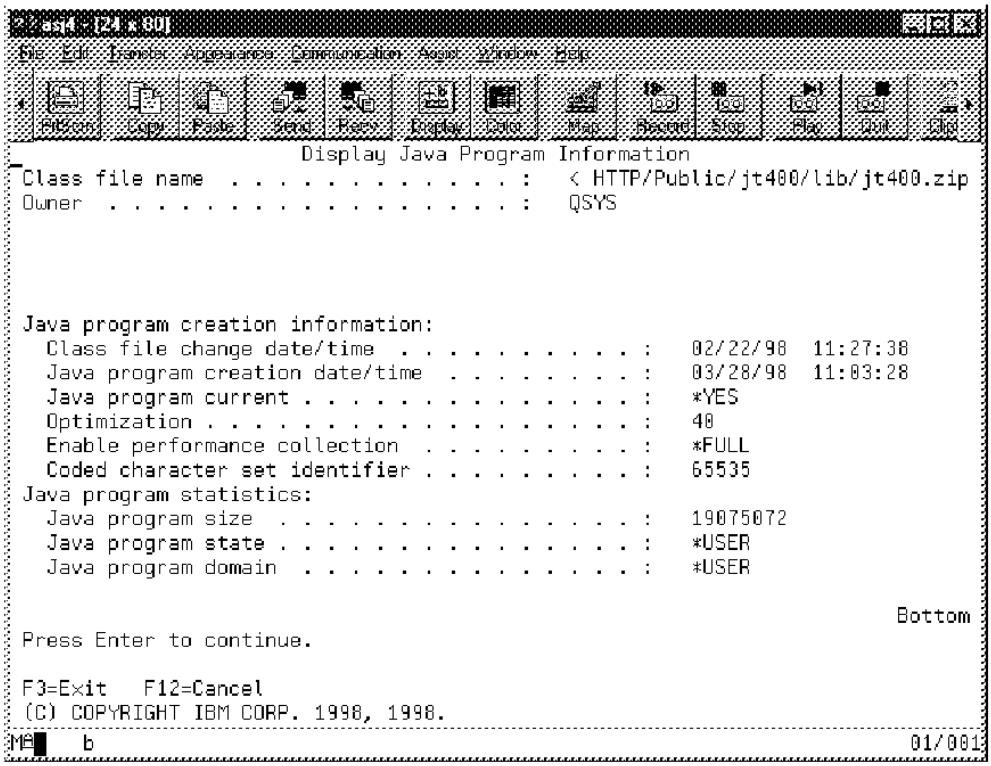


Figure 281. Java Program Information

Note the size of the Java program that was created. This is a small price to pay for much improved runtime performance.

After the Java programs were created, performance of the first order submission improved significantly. However, the first order submission still took several seconds, indicating the need for further improvements.

14.6.1.2 Pre-Creating Data Queue Objects

A significant portion of the response time in the order submission was spent in creating the data queue objects within the PersistencyManagers.OrderEntryJdbcRMIPersistencyManager writeDataQueue method. Here is the original implementation:

```
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), "customerID");
 PackedDecimalFieldDescription as4DistrictID = new
 PackedDecimalFieldDescription(new AS400PackedDecimal(3,0),
      "districtID");
 CharacterFieldDescription as4WarehouseID = new
 CharacterFieldDescription(new AS400Text(4), "warehouseID");
 PackedDecimalFieldDescription as4OrderID = new
 PackedDecimalFieldDescription(new AS400PackedDecimal(9,0), "orderID");

 DataQueue dqOutput = new DataQueue(as400, "/"+SYSTEM_LIBRARY+"/"+
      DATA_QUEUE_LIBRARY+"/"+DATA_QUEUE_NAME);
```

```
 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());
```

To minimize this response time, some portions of this method were moved to a separate method initDQFormat() that is invoked when the initialize() method is run. Here is the new writeDataQueue method. It checks to see if the record format object rfOutput has been instantiated. This is normally done in the initialize method at startup time. It calls the initDQFormat if the object does not exist. Now the writeDataQueue method is only concerned with formatting and creating records on the data queue. It does not create objects over and over again.

```
private void writeDataQueue (String aCustomerID,
          BigDecimal anOrderID ) throws Exception
{


 // Set up the data queue entry field values
 //private RecordFormat rfOutput = null;
 if (rfOutput == null) initDQFormat();

 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;
 }
```

14.6.1.3 JDBC PreparedStatement and SQL String Manipulation

With the exception of the psAddOrderLine PreparedStatement, all of the other SQL requests had been defined as Statement objects. Because these SQL statements are all executed repeatedly, changing them to PreparedStatement objects provides a significant performance boost. Here is the original coding example for inserting Order rows to the Order database in the addOrderHeader method:

```
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(currentDateTime)+
    ", "+getRawTime(currentDateTime)+")";

addOrderHeader.executeUpdate(sql);
```

The modification, when applied, provided a significant performance improvement. In the previous implementation, every execution of the executeQuery(sql) or executeUpdate(sql) methods caused the statement to be implicitly prepared before it was run. With the PreparedStatement, the prepareStatement() method for the statement is only done once per application. Setting the values for the parameter markers is all that was needed prior to the executeQuery() or the executeUpdate(). When the executeUpdate() method runs, there is no need for an implicit prepare and the SQL statement is processed from the SQL statement cache.

In addition, the formation of the SQL statement strings was done each time, prior to the executeQuery() or the executeUpdate(). This resulted in unnecessary string manipulation through assignments and concatenation.

By converting the SQL statements to PreparedStatement objects, the formation of the SQL statement string only has to be done once during the life of the application, that is, within the initialize() method. First, we prepare the statement in the **initialize** method:

```
psAddOrderHeader = dbConnection.prepareStatement("INSERT INTO ORDERS " +
 "(OWID, ODID, OCID, OID, OLINES, OCARID, OLOCAL, OENTDT, OENTTM) "+
 "VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?)");
```

In the new **addOrderHeader** method, we set the parameters and execute the statement:

```
// Add a new order header record
 psAddOrderHeader.setString(1,WAREHOUSE);
 psAddOrderHeader.setBigDecimal(2,new BigDecimal(DISTRICT));
 psAddOrderHeader.setString(3,aCustomerNumber);
 psAddOrderHeader.setBigDecimal(4, anOrderNumber);
 psAddOrderHeader.setBigDecimal(5, anOrderLineCount);
 psAddOrderHeader.setString(6,"ZZ");
 psAddOrderHeader.setBigDecimal(7, new BigDecimal(1));
 psAddOrderHeader.setBigDecimal(8,
        new BigDecimal(getRawDate(currentDateTime)));
 psAddOrderHeader.setBigDecimal(9,
        new BigDecimal(getRawTime(currentDateTime)));

 psAddOrderHeader.executeUpdate();
```

14.6.1.4 Remote Method Invocation Support

Three changes were made to the RMI support logic to help improve response time:

- RMI initialization was moved to system sign-on time.

  Previously, the lookup for the RMISecurityManager was done at the time of the first order submission. This caused several seconds of delay.

- A dummy order is processed at signon time to cause all AS/400 Java class loading to be done before a real order is processed.
- The RMI initial processing is done in a unique thread to allow it to be done concurrently with system sign on.

These modifications do not improve the overall efficiency of the application. Its effect is to shift a significant response time component from a more critical transaction to a less noticeable one.

14.6.1.5 Method Inlining through javac -O

The OrderEntryJdbcRMIPersistencyManager class most likely does not need to be subclassed with modified behavior and attributes. It can then be changed to a final class. Method inlining can then be taken advantage of by re-compiling the program with the -O option.

14.6.2 SQL Stored Procedures to Replace RPG Stored Procedures

Using stored procedures in a client/server environment provides a large performance improvement by packaging multiple database operations into a single request. However, one common objection is that a high-level language stored procedure implementation is tied to a specific platform. The SQL stored procedure support available in V4R2 allows stored procedures to be written using ANSI standard syntax.

This modification was already included in the base measurement of the List Customers function. It is an improvement over the previous version of the application. Creation of the SQL Stored Procedure, available in V4R2, requires the following program products:

- 5769ST1 - DB2 Query Mgr and SQL DevKit for AS/400
- 5769CX2 - ILE C for AS/400

The SQL stored procedure is implemented as SQL statements embedded in an ILE C program. The program size is smaller than the equivalent RPG stored procedure.

A sample SQL statement used to create the stored procedure is shown in the following example:

```
CREATE PROCEDURE apilib/sltcustr (IN p1 CHAR(1), IN p2 CHAR(1))
  RESULT SET 1
  LANGUAGE SQL
  proc1:
     BEGIN
  DECLARE customer CURSOR FOR
  SELECT cid, cfirst, cinit, clast, caddr1, caddr2, ccity, cstate, czip
  FROM cstmr
  WHERE cdid = 1 AND cwid = '0001'
  ORDER BY cid
```

```
FOR FETCH ONLY;

OPEN customer;

SET RESULT SETS CURSOR customer;

END
```

There are several ways to run this statement, including:

- Within an interactive SQL session
- Embedded in a high-level language program
- Run from a source file member with the RUNSQLSTM command

14.6.2.1 Verifying Item and Customer Information

Instead of retrieving data from the server each time, customer and item details can be taken from their respective managers. For example, before an item is added to an order, the item number must be validated. Previously, the application used a DDM record level read to read the Item file on the AS/400 system to check if the item was valid. Now, the catalog object that resides on the client is checked first. If the item is found there, no AS/400 access is required.

14.7 Performance Reports

The performance reports that are available with the Performance Tools licensed program product provide new information regarding threads. They are sufficient for identifying system tuning opportunities, hardware bottlenecks, and providing accurate statistics on resource consumption.

However, they are of limited use in analyzing the performance of Java applications. The recommended procedures for analyzing Java program instruction execution and database access are described in this section.

14.7.1 System Report

Within the system report, the resources consumed by the BCI jobs are reported under the general category of "Batch." Also note that the priority that is reported is of the main Java thread. In the following example (Figure 282 on page 378), the Java application was run from an interactive session, causing the BCI job and the initial thread to run at a default priority of 20. However, the actual Java application workload is running at a thread priority of 26 and is reported as such. Refer to Chapter 13, “Java Performance and Work Management Overview” on page 323, for a review on run priority assignments.

```
                                        System Report
                                  Resource Utilization Expansion                                    Page
                                 Check job Priority if GCPTY changed
Member  . . . : Q981031739 Model/Serial  . : 620-2181/10-0CD2D   Main storage . . :  512.0 M   Started  . . . . : 04/13/98 17:40:2
  Library . . : LLAMESPFR  System name . . :          SYSNAME    Version/Release  :   4/ 2.0   Stopped  . . . . : 04/13/98 17:45:3
              ------------------------------------------- Average Per Transaction ------------------------------------------
              ---------- Physical Disk I/O ----------  -------- Logical ---------  -- Communications --
     Job      --------- Synchronous ---------  -------- Asynchronous ---------  ----- Data Base I/O ------          I/O
     Type       DBR     DBW    NDBR    NDBW      DBR     DBW    NDBR    NDBW     Read     Write     Other     Get          Put
-------------  ------- ------- ------- -------  ------- ------- ------- -------  -------  -------   -------   -----        -----
Client Access       .00     .00   22.00     .00      .00     .00     .00     .00      .00      .00       .00      .0           .0
PassThru            .25     .00   39.75     .93      .00     .06     .06    5.18      .00      .00       .00      .0           .0
Average             .23     .00   38.70     .88      .00     .05     .05    4.88      .00      .00       .00      .0           .0
-------------------------------------------------------------------------------------------------------------------------------

                  Job          CPU       Cum      ------- Disk I/O -------    ----- CPU Per I/O ------     -- DIO /Sec --
Priority          Type         Util      Util         Sync        Async          Sync         Async         Sync    Async
--------  --------------       ----      -----    ----------  ----------    ----------   ----------      -----    -----
  000     Batch                 .2        .2            477         260         .0015        .0028        1.5       .8
          System               1.2       1.4            790          91         .0047        .0412        2.6       .3
  009     System                .0       1.4              0           0         .0000        .0000         .0       .0
  010     Batch                 .0       1.4              2           0         .0005        .0000         .0       .0
  016     System                .0       1.4              0           0         .0000        .0000         .0       .0
  020     Client Access         .0       1.4             22           0         .0002        .0000         .0       .0
          PassThru             1.7       3.2            655          85         .0083        .0640        2.1       .2
          Batch                3.0       6.3            805          63         .0113        .1455        2.6       .2
          AutoStart             .0       6.3              1           0         .0010        .0000         .0       .0
          System                .0       6.3              1           0         .0010        .0000         .0       .0
  021     Batch                 .3       6.6              0           0         .0000        .0000         .0       .0
  025     Batch                 .2       6.9            440          99         .0014        .0064        1.4       .3
  026     Batch                4.0      10.9            883          10         .0137        1.24         2.9       .0
  030     Batch                 .0      10.9             10           0         .0000        .0000         .0       .0
  035     Batch                 .0      10.9             29           0         .0004        .0000         .0       .0
  036     System                .0      10.9              0           0         .0000        .0000         .0       .0
  040     System                .0      10.9              8           4         .0016        .0032         .0       .0
  050     Batch               82.3      93.2          2,011           0         .1240        .0000        6.6       .0
```

Figure 282. System Report for Java Workload

14.7.2 Component Report

As in the past, the component report shows one entry for every job, including BCI jobs, which are also reported as job type "B", as in batch. The priority is shown to be that of the Java application threads (priority 26 in this example). A sample report is shown in Figure 283.

```
                                        Component Report
                                     Job Workload Activity                                          Page
                                 Check job Priority if GCPTY changed
Member  . . . : Q981031739 Model/Serial  . : 620-2181/10-0CD2D   Main storage . . :  512.0  M  Started . . . . :  04/13/98 17:40:2
  Library . . : LLAMESPFR  System name . . :       SYSNAME       Version/Release  :    4/ 2.0  Stopped . . . . :  04/13/98 17:45:3
                           T     P
   Job       User    Job    y     t    CPU           Tns          ---------- Disk I/O ---------    Cmn     PAG  Arith   Perm
   Name      Name    Number p  Pl y    Util    Tns   /Hour   Rsp      Sync     Async    Logical     I/O    Fault Ovrflw  Write
--------- ---------- ------ -- -- -- ------ ------- ------- ------- --------- --------- --------- -------- ----- ------- -----
QACSOTP     QUSER    003395 B  02 20   .000       0       0    .000         2         0         0        0     1       0
QALERT      QSYS     003125 S  02 20   .000       0       0    .000         0         0         0        0     0       0
QBATCH      QSYS     003387 M  02 00   .000       0       0    .000         0         0         0        0     0       0
QCMN        QSYS     003391 M  02 00   .000       0       0    .000         3         0         0        0     3       0
QCMNARB01   QSYS     003134 S  02 00   .000       0       0    .000         0         0         0        0     0       0
QCMNARB02   QSYS     003135 S  02 00   .000       0       0    .000         0         0         0        0     0       0
QCTL        QSYS     003137 M  02 00   .000       0       0    .000         0         0         0        0     0       0
QDBSRVXR    QSYS     003128 S  02 00   .002       0       0    .000        36         5         2        0    13       0
QDBSRVXR2   QSYS     003132 S  02 00   .000       0       0    .000         0         0         0        0     0       0
QDBSRV01    QSYS     003115 S  02 09   .000       0       0    .000         0         0         0        0     0       0
QDBSRV02    QSYS     003116 S  02 16   .000       0       0    .000         0         0         0        0     0       0
QDBSRV03    QSYS     003117 S  02 16   .000       0       0    .000         0         0         0        0     0       0
QDBSRV04    QSYS     003118 S  02 52   .000       0       0    .000         0         0         0        0     0       0
QDBSRV05    QSYS     003119 S  02 52   .000       0       0    .000         0         0         0        0     0       0
QDCPOBJ1    QSYS     003120 S  02 60   .000       0       0    .000         0         0         0        0     0       0
QDCPOBJ2    QSYS     003121 S  02 60   .000       0       0    .000         0         0         0        0     0       0
QESRRMI     GAPASSE  004079 B  02 56   .005       0       0    .000         8         0         0        0     2       0
QESRSVR     GAPASSE  004080 B  02 56   .042       0       0    .000        12         0         0        0     2       0
QFILESYS1   QSYS     003127 S  02 00   .005       0       0    .000        13         0         0        0     2       0
QINTER      QSYS     003384 M  02 00   .018       0       0    .000       119         8         0        0    17       0     1
QIWVPPJT    QUSER    003461 B  02 20   .000       0       0    .000         1         0         0        0     1       0
QJOBSCD     QSYS     003124 S  02 00   .000       0       0    .000         0         0         0        0     0       0
QJVACMDSRV  LLAMES   004551 B  04 26   .563       0       0    .000      1128        35         1        0     1       0     3
QLUR        QSYS     003126 S  02 00   .000       0       0    .000         0         0         0        0     0       0
QLUS        QSYS     003113 S  02 00   .000       0       0    .000         0         0         0        0     0       0
```

Figure 283. Component Report for Java Workload

14.7.3 Transaction Report

The transaction report shows a separate entry for every thread as seen in Figure 284.

```
                                                  Job Summary Report
                                                      Job Summary
                                                Check job Priority if GCPTY changed
Member  . . . : Q981031739 Model/Serial  . : 620-2181/10-0CD2D   Main storage . . :  512.0 M   Started  . . . . :04/13/98 17:40:37
  Library . . : LLAMESPFR  System name . . :          ASJ4       Version/Release  :   4/ 2.0   Stopped  . . . . :04/13/98 17:45:35
                    *On/Off*    T  P  P  Tot  Response Sec        CPU Sec        ---- Average DIO/Transaction -----   Number  K/T
   Job     User Name/  Job      y  t  r  Nbr  -------------  ------------------  ------ Synchronous ----- --Async--    Cft    /Tn
   Name    Thread     Number  Pl p  y  g  Tns    Avg    Max   Util   Avg    Max  DBR  NDBR  Wrt  Sum  Max  Sum  Max  Lck Sze  Sec
---------- ---------- ------  -- -- -- -  ----  ------ ------  ---- ------ ------  ---- ----  ---- ---- ---- ---- ----  --- ---- ----
ENDPFRSUN  LLAMES     004542  02 B  50                        82.0        244.62                      1967
QTTFT00259 QTFTP      004544  02 B  25
QTTFT00195 QTFTP      004545  02 B  25
QPADEV0004 LLAMES     004547  04 I  20
QPFRMON    QPGMR      004550  02 B  00                         .2          .83                       294       118
QJVACMDSRV LLAMES     004551  04 BD 20                        .3          .87                  261        25
QJVACMDSRV  00000011  004551  04 BD 20                       23.0        10.85                  837        10
QJVACMDSRV  00000012  004551  04 BD 20                        .4         1.16
QJVACMDSRV  00000013  004551  04 BD 20                                   .01
QJVACMDSRV  00000014  004551  04 BD 20                                   .11
QJVACMDSRV  00000015  004551  04 BD 20                                   .01
QJVACMDSRV  00000016  004551  04 BD 20                                   .25                   24
QJVACMDSRV  00000017  004551  04 BD 20                                   .01
QJVACMDSRV  00000018  004551  04 BD 20                       12.2         .02                    6
QP0ZSPWT    00000010  003639  02 BD 50                                              .01                  24
QP0ZSPWT    00000011  003639  02 BD 50                                                                    8
QJVACMDSRV   00000019    004551   04  BD 20
QJVACMDSRV   0000001A    004551   04  BD 20
```

Figure 284. Transaction Report for Java Workload

The entry, QJVACMDSRV LLAMES 004551, represents the initial thread described in Chapter 13, “Java Performance and Work Management Overview” on page 323. The other threads in the process are listed with the same job names and job numbers but have their thread numbers listed under the "User Name/Thread" column.

Note the job types of "BD" and the run priorities. In V4R2, there appears to be a defect in the report since the reported thread priorities are not as expected. In this run, all of the Java application threads and garbage collection threads were running at priority 26.

One way to verify thread information about a job is to use the Display Job (DSPJOB) command, as in the following example. In this case, the output is sent to a spooled file.

```
DSPJOB JOB(004551 / LLAMES / QJVACMDSRV) OUTPUT(*PRINT) OPTION(*THREAD)
```

14.7.4 Convert Performance Thread Data (CVTPFRTHD)

The Convert Performance Thread Data (CVTPFRTHD) command is part of OS/400 and converts performance data records collected by the Performance Monitor. The specified member of the file QAPMJOBS contains records with thread-level performance data. This command can be used to convert this data and write the resulting records to a member in file QAPMTJOB. The output file member contains records with job-level (consolidated) performance data, that is, the total of the performance information for all threads running within the job.

14.8 Application Analysis — System Services (DB)

Chapter 13, “Java Performance and Work Management Overview” on page 323, describes the typical commercial application profile. Since it consists of both Java instruction execution and access to system services, it is important to account for each to determine where most of the processing resources are being consumed. Knowing this provides a direction on how to maximize performance.

14.8.1 Performance Explorer *STATS Mode

The first step in analyzing the server application is to obtain a breakdown of the database portion versus the Java instruction execution portion. This helps to determine whether the most significant improvements can be derived from database tuning or from Java application modification.

One accurate way to do this is to use the Performance Explorer statistics (*STATS) mode. For more information on the use of the Performance Explorer, refer to the manual *Performance Tools V4R2*, SC41-5340.

The first step is to add a Performance Explorer Definition as shown in Figure 285.

```
                      Add PEX Definition (ADDPEXDFN)
Type choices, press Enter.
Definition . . . . . > STATSDEF      Name
Type . . . . . .   *STATS        *STATS, *TRACE, *PROFILE
Job name . . . . . > *ALL          Name, generic*, *ALL, *
  User . . . . . .                 Name, generic*, *ALL
  Number . . . . . .               000001-999999, *ALL
             + for more values
Task name  . . . . . > *ALL
             + for more values
Data organization  . . . . . .   *FLAT         *FLAT, *HIER
```

Figure 285. Add PEX Definition (ADDPEXDFN) Display

In Figure 285, the job identifier can be made more specific to include only the jobs involved in running Java applications. To start the measurement, the STRPEX command is used. The application or function can then be run. The PEX measurement is ended with the ENDPEX command.

The PRTPEXRPT command within the Performance Tools product is used to print the *STATS report. In this report is a Process/Task section. Within this section, the comparison of overhead between the Java BCI job, for example, QJVACMDSRV, can be compared with its corresponding database server job, for example, QSQSRVR.

This information can also be derived from the Performance Monitor and a standard Performance Tools report such as the Component Report. If trace data was collected, the Performance Tools Transaction Report also provides job-level information.

The PEX *STATS report provides more granularity within its module section. If the application uses stored procedures or DPC, the associated program and its statistics appear. Individual SQL requests and optimizer processing would be grouped into the cumulative statistics of the QSQROUTE module. The low level implementation of logical I/O requests and file open and close operations would

appear under the QDM. . . (data management) modules and their subprograms, the QDB (database) modules.

14.8.2 Database Monitors

Because JDBC requests are done through the SQL interface, the recommended methods for analyzing the SQL statements are through either the Database Monitor, the Debug facility, or the V4R4 SQL Monitor available from Operations Navigator. Remember that the database requests are processed through a separate job described in Section 14.4.2, “JDBC through Native Driver” on page 362.

For more information about some of these facilities, see *OS/400 DB2 for AS/400 Database Programming V4R2*, SC41-5701.

14.9 Application Analysis — Java Instructions

The recommended way for detailed analysis of Java applications on the AS/400 system is through the Performance Explorer. Other means are also available, such as combining stop-watch measurements with program output (System.out.println) within various portions of the application. However, these are neither accurate nor precise and are subject to external factors.

Within the Performance Explorer, two facilities have been tested with respect to this application. There are other variations to the methodology using different combinations of PEX definition parameters and reporting parameters.

The first is the profiling technique, which makes use of program instruction statistical sampling to identify hot spots within the Java programs. The results of the sampling are reported through the PRTPEXRPT . . . TYPE(*PROFILE) command.

The *PROFILE sampling technique is meant for measuring applications with "significant" amounts of work, possibly from multiple users. It is described in detail in a later section.

Another facility is the PEX trace technique, which records and reports on all of the events that occur within the application. This has been used in analyzing individual transactions, although it can measure and report on system-wide data. It generates large amounts of data. The type of analysis that is described is analogous to the trace job methodology that is normally used in other languages.

14.9.1 The PEX Definition

For simplicity, a single Performance Explorer definition was used for both techniques. In a single measurement, both profile and trace facilities can be used to report on and to analyze the application. It is also possible to use more specialized PEX definitions if a specific performance problem is being analyzed. This reduces the amount of data collected.

The following PEX definition was used to measure the application. This PEX definition, when used in a measurement, collects sufficient data for both *TRACE and *PROFILE reports.

```
ADDPEXDFN DFN(JAVAPERF) TYPE(*TRACE) JOB(*ALL) TASK(*ALL)
MAXSTG(1000000) INTERVAL(1) TRCTYPE(*SLTEVT) SLTEVT(*YES) BASEVT(*PMCO)
PGMEVT(*ALL) JVAEVT(*ALL)  TEXT('Java application analysis')
```

The INTERVAL(1) parameter, in millisecond units, specifies the sampling rate for the *PROFILE report. In most other languages, a separate PEX definition of TYPE(*PROFILE) can be created to monitor the hot spots within a program. However, in a typical Java application, the specific program name is not known.

The other parameter needed for the profile report is BASEVT(*PMCO).

The parameter PGMEVT(*ALL) includes Java entry and exit events, as well as Java pre-call and post-call events. The Java exit events are used for detailed application analysis as shown later.

If too much data is collected and performance data processing time is deemed too long, a single purpose measurement can be used. For example, if only a *PROFILE report is needed, the following PEX definition can be used.

```
ADDPEXDFN DFN(JAVAPERF) TYPE(*TRACE) JOB(*ALL) MAXSTG(1000000) INTERVAL(1)
SLTEVT(*YES) BASEVT(*PMCO) TEXT('Java application analysis')
```

If sampling for profile data is not desired, for example, trace data analysis is needed, remove the INTERVAL(1) parameter from the original PEX definition.

Another way to reduce data is to include only those jobs or groups of jobs that are directly related to the Java application.

14.9.2 ENBPFRCOL(*ENTRYEXIT)

To gather complete information about the application's performance, the Java programs should be created with the ENBPFRCOL(*ENTRYEXIT) parameter. The default value is ENBPFRCOL(*NONE). This change affects the PEX trace data.

Note that the standard Java Runtime Environment (JRE) classes are not enabled for performance collection. For V4R2, these classes are stored in the following files:

- /QIBM/ProdData/Java400/lib/java.zip
- /QIBM/ProdData/Java400/lib/sun.zip

In V4R4, the JDK 1.1.7 support is in a single file: /QIBM/ProdData/Java400/jdk117/lib/classes.zip.

Modifying these files' Java programs for any reason, such as enabling performance collection, can cause problems and may require the Java program product to be re-installed.

Enabling the Java runtime support for performance collection is usually not necessary. Since these classes are not enabled within a PEX measurement, the runtime statistics that are accumulated within the runtime classes are accumulated upwards in the call stack until the first class, which is enabled for performance collection. The first classes encountered should be application classes that you, as a developer, created. These are classes that you have control over and can modify.

On the other hand, AS/400 Toolbox for Java programs and the native JDBC driver classes can be enabled for performance collection.

14.9.3 PEX Measurement

In this example, the first order submission was analyzed, since this was the longest running transaction in the initial version. We followed this process:

1. Start the PEX measurement as shown in Figure 286.

```
                    Start Performance Explorer (STRPEX)

Type choices, press Enter.

Session ID . . . . . . . . . . . > ORDER01       Name
Option . . . . . . . . . . . . .   *NEW          *NEW, *INZONLY, *RESUME
Definition . . . . . . . . . . . > JAVAPERF      Name
```

Figure 286. Start Performance Explorer (STRPEX) Display

   The STRPEX command starts the Performance Explorer measurement under the PEX definition of JAVAPERF that was created earlier through the ADDPEXDFN command. The "Session ID" parameter is used to identify the measurement.

2. Run the transaction.

   The Submit Order button was clicked. The Java program on the AS/400 system starts processing the order as PEX measurements are taken. For a complete analysis, the application must be running in Transformed mode (not interpreted).

3. End the measurement.

   After the confirmation message "Order successfully processed" is displayed, the PEX measurement ends.

```
                     End Performance Explorer (ENDPEX)

Type choices, press Enter.

Session ID . . . . . . . . . . . > ORDER01       Name, *SELECT
Option . . . . . . . . . . . . .   *END          *END, *SUSPEND
Data option  . . . . . . . . . .   *LIB          *FILE, *LIB, *DLT
Data library . . . . . . . . . .   llamespex     Name
Data member  . . . . . . . . . .   *SSNID        Name, *SSNID
Replace data . . . . . . . . . .   *NO           *YES, *NO
Text 'description' . . . . . . .   First submit order
```

Figure 287. End Performance Explorer (ENDPEX) Display

   The "Session ID" value matches the measurement that was started.

   The approach that is presented here requires the use of the default "Data option" of *LIB. This generates several file members in the specified "Data library."

4. Process the PEX Data.

14.9.4 Performance Explorer *PROFILE Reports

Use this facility to identify the most heavily used Java programs, methods, and statements within the system. Since this uses a sampling mechanism, it depends on having sufficient samples to provide a statistically significant measurement that truly represents the hot spots within the Java application.

This approach is valid if there is a significant amount of repetitive work. The recommended definition of "significant work" is at least two to three minutes of CPU time. It is possible to accumulate this much if the application is being used by many users or if the application is looping continuously so that enough hits are generated.

In the profile report for our application, the testing was done with a single user and few samples were generated. However, the example may indicate the kind of information that may be available from this report.

14.9.4.1 PEX Definition and Print PEX Report (PRTPEXRPT) Parameters

The PEX definition previously given is the same one used for this measurement.

The Print PEX report parameter is shown here:

```
PRTPEXRPT MBR(order01) LIB(LLAMESPEX) TYPE(*PROFILE)
PROFILEOPT(*ADDRESS *PROCEDURE)
```

The previous PROFILEOPT values specify that the report details are printed in address order instead of by the number of hits. In addition, the statistics are summarized by *PROCEDURE. A procedure within a Java application translates into Java methods.

It is possible to summarize the statistics by *STATEMENT. The actual Java statement can be determined by printing a disassembly listing of the program through System Service Tools and scanning for the "Start Addr" value from the PEX profile report.

Figure 288 shows some of the output available from the profile report summarized by *PROCEDURE.

```
                                          Performance Explorer Report
                                              Profile Information
Library . . : LLAMESPEX
Member. . . : ORDER01
Description : Submit 1st Order  PEXDFN JAVAPERF    ENBPFR sun zip java zip
                          Histogram     Hit   Hit   Cum   Start            Map    Stmt   Name
                                        Cnt    %     %    Addr             Flag   Numb
-------------------------------------------------------------------------------------------------------------------
                                         10   0.1  99.7 04FA57C405004900   ==        0  QP0WPTHR/pthread_getthreadid_np
                                          2   0.0  99.7 04C6B36E8A0063D0   ==        0  QWSSFLCT
                                          5   0.1  99.7 03E2C4694D002BAC   ==        0  QSFPUT
                                ====>     1   0.0  99.8 03A3FA9CCA011AB4   ==        0  RENTRYJDBC DEMOD/PersistencyManagers-Ord
                                                                                          erEntryJdbcRMIPersistencyManager-addOrder
                                                                                          Line(Ljava-math-BigDecimal;LDomainObject
                                                                                          s-Order;)Ljava-math-BigDecimal;
                                          1   0.0  99.8 035941E788001904   ==        0  QUIMGFLW
                                          1   0.0  99.8 032FA3E19A00150C   ==        0  QSPBPPRT
                                          1   0.0  99.8 03237BD5030039F0   ==        0  QJVAJNM    QJVALIBIO/Java_java_io_Random
                                                                                          AccessFile_close
```

*Figure 288. Profile Report by *PROCEDURE for Java Workload*

This example obviously did not collect enough samples because there is only one hit count for the application's addOrderLine() method. The other methods do not even show up because their hit counts are zero. With such a small number of

samples, this measurement is not statistically significant. It is not representative of the real application profile. With many users running this function repeatedly, it may be possible to collect many more samples.

When the report is summarized by *STATEMENT, the entry for the addOrderLine() method is further subdivided. Each roughly represents a Java program statement, each with its own hit count which, when added up, matches the total in the *PROCEDURE report.

14.9.5 Performance Explorer *TRACE Data

To get the finest level of granularity in this analysis, the applications have to be running in transformed mode, that is, not interpreted. In addition, all of the Java application programs need to have performance collection fully enabled.

Section 14.9.2, “ENBPFRCOL(*ENTRYEXIT)” on page 382, contains a discussion regarding the enabling of standard Java support for performance collection.

This approach collects data based on events, unlike the profile approach that relies on statistical sampling. Therefore, using trace is more appropriate for analyzing individual transactions in detail.

14.9.5.1 PEX Definition and PEX Print Parameters

The PEX definition previously given is the same one used for this measurement.

The current version of the PEX reports is not adequate for analyzing Java application performance. The problem can be attributed to the reports being designed to handle short program names. For example, RPG program names are only expected to have 10 characters. Java programs, on the other hand, generally have much longer names. Include the package directories and the method names, and the PEX report no longer indicates anything beyond the initial directory structure.

A recent test modification was made to the PRTPEXRPT . . . OUTPUT(*OUTFILE) support to get around this current limitation. This modification will become a PTF starting with V4R2. It involves generating a link between the output file's QTRDID column and the PEX file QAYPENMI. The latter file is the PEX MI Name Data file, which has a variable length field containing all MI program names, as well as Java class and method names. For Java trace events, the field QTRDID is used to store the link to the PEX MI Name data file. In the subsequent steps, these two files are joined in the queries.

PTF Information

The PTF number for V4R2 and V4R3 is SF52818. By the time this redbook is published, it may already be superseded. In V4R4, this feature will be integrated.

The parameters shown in Figure 289 on page 386 were used to generate the *OUTFILE.

```
                        Print PEX Report (PRTPEXRPT)

Type choices, press Enter.

Member . . . . . > ORDER01       Name
Library  . . . . . > LLAMESPEX     Name
Type . . . . . > *TRACE        *STATS, *TRACE, *PROFILE...
Output . . . . . > *OUTFILE      *PRINT, *OUTFILE
File to receive output . . . . . > ORDER01       Name
  Library  . . . . . >   LLAMESPEX   Name, *LIBL, *CURLIB
Output member options:
  Member to receive output . . .   *FIRST        Name, *FIRST
  Replace or add records . . . .   *REPLACE      *REPLACE, *ADD
Trace options:
  Sort by  . . . . . > *TASK         *TIMESTAMP, *TASK
  Omit completion records  . . .   *NO           *NO, *YES
  Omit Category  . . . . . .   *NONE         *NONE, *PGM, *LICPGM, *ASM...
              + for more values
Select trace type  . . . . . .   *ALL          *ALL, *CALLRTN, *BASIC...
              + for more values
```

Figure 289. Print PEX Report (PRTPEXRPT) Display

The member name is the same one that was generated during the PEX measurement for the first submit order transaction. Data is taken from the raw QAYP... data file members and combined into the DB file that is specified in the "File to receive output" parameter, LLAMESPEX/ORDER01. If this file does not exist, it is created when the command is run. The report type is *TRACE and is sorted by *TASK.

14.9.5.2 Query Definition — Application Report

In the Application report (Figure 290 on page 387), the purpose of the first query is to identify which application methods result in high CPU consumption and in high response time contributions.

```
5769QU1  V4R2M0  980228           IBM Query/400         SYSNAME      4/01/98   12:43:24
  Query . . . . . . . . . . . . . . . . JVAEVT6
    Library . . . . . . . . . . . . . . LLAMES
  Query text  . . . . . . . . . . . . . Detail JVXIT, Cumul.,Sort-task,recid, Persistency
  Query CCSID . . . . . . . . . . . . . 65535
  Query language id . . . . . . . . . . ENU
  Query country id  . . . . . . . . . . US
    *** . is the decimal separator character for this query ***
  Collating sequence  . . . . . . . . . Hexadecimal
  Processing options
    Use rounding  . . . . . . . . . . . Yes (default)
    Ignore decimal data errors  . . . . . No (default)
    Ignore substitution warnings  . . . . Yes
    Use collating for all compares  . . . Yes
Selected files
  ID     File          Library      Member       Record Format
  T01    ORDER01       LLAMESPEX    *FIRST       QVPETRC
  T02    QAYPENMI      LLAMESPEX    ORDER01      QYPENMI
Join tests
  Type of join  . . . . . . . . . . . . Matched records
  Field             Test             Field
  T01.QTRDID        EQ               T02.QNMOFF
Select record tests
  AND/OR   Field            Test     Value (Field, Numbers, or 'Characters')
           T01.QTREID       EQ       'JVXIT '
   AND     T01.QTRTNM       LIKE     'QP0Z%'
   AND     T02.QNMLNM       LIKE     'Persis%'
Ordering of selected fields
  Field          Sort      Ascending/  Break  Field
  Name           Priority  Descending  Level  Text
  T01.QTRTCT     10        A                  Event task id
  Field          Sort      Ascending/  Break  Field
  Name           Priority  Descending  Level  Text
  T01.QTRRCN     20        A                  Record number (UNIQUE)
  T01.QTRSPC                                  Record is suspicious
  T01.QTRCLV                                  Program relative call level
  T01.QTRCUS                                  Program cumulative cpu microseconds
  T01.QTRCES                                  Program cumulative elapsed microseconds
  T02.QNMLNM                                  Mi long name
Report column formatting and summary functions
  Summary functions:  1-Total, 2-Average, 3-Minimum, 4-Maximum, 5-Count          Overrides
  Field          Summary    Column                              Dec  Null       Dec  Numeric
  Name           Functions  Spacing  Column Headings         Len  Pos  Cap   Len  Pos  Editing
  T01.QTRTCT                0        Task ID                  10    0
  T01.QTRRCN                2        Record                    9    0
                                     Number
  T01.QTRSPC                2        Is                        2    0
                                     Suspicious
  T01.QTRCLV                2        Call                      4    0
                                     Level
  T01.QTRCUS                2        Program CPU              20    0
                                     Microseconds
                                     Cumulative
  T01.QTRCES                2        Program elapsed          20    0
                                     Microseconds
                                     Cumulative
  T02.QNMLNM                2        Mi long name            256   V
Report breaks
  Break  New   Suppress   Break
  Level  Page  Summaries  Text
  0      No    Yes        FINAL TOTALS
Selected output attributes
  Output type . . . . . . . . . . . . . Printer
  Form of output  . . . . . . . . . . . Detail
  Line wrapping . . . . . . . . . . . . Yes
    Wrapping width  . . . . . . . . . .
    Record on one page  . . . . . . . . No
              * * * * *   E N D   O F   Q U E R Y   P R I N T   * * * * *
```

*Figure 290. Query Definition for PEX Trace *OUTFILE — Application Methods*

Only the records with an event ID of JVXIT are used. A JVXIT record is generated when an exit is made from a Java method. These records that contain the accumulated CPU measurements.

The other record selection parameter "QTRTNM LIKE QP0Z% includes only the events associated with the BCI job that is running the Java application.

The following possible values are for Java BCI job names:

- **QJVACMDSRV**

  The JAVA or the RUNJVA commands were used to run the Java application.

- **QZSHSH**

  The application was run within an interactive QSH or QSHELL session.

- **QP0ZSPWT**

  The application was run within a QSH or QSHELL script.

The third selection criterion, T02.QNMLNM LIKE 'Persis%', selects only the application methods that belong to the Java package PersistencyManagers. This excludes the standard Java classes.

The cumulative statistics are used. They represent the accumulation of a specific method's direct CPU usage, as well as usage by the other methods that it called. That is why the Program relative call level field is included as well. It makes it easy to see which method called another.

14.9.5.3 Application Query Results and Interpretation

The report shown in Figure 291 is the result of the query over the application methods.

```
04/18/98  12:19:46                                    Original SBMORDER                                                PAGE   1
      Task ID        Record   Is          Call          Program CPU          Program elapsed
                     Number   Suspicious  Level         Microseconds         Microseconds
                                                        Cumulative           Cumulative
Mi long name
       47,859       378,273        0         69                153,436                4,201,472
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getOrderNumber()Ljava-math-BigDecimal;
       47,859       381,248        0         70                  7,219                  268,208
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getCustomerDiscount(Ljava-lang-String;)Ljava-math-BigDecimal;
       47,859       391,143        0         70                 13,311                  391,344
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-updateStock(Ljava-lang-String;Ljava-math-BigDecimal;)V
       47,859       391,149        0         69                 36,836                1,288,712
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-addOrderLine(Ljava-math-BigDecimal;
                                                                                        LDomainObjects-Order;)Ljava-math-BigDecimal;
       47,859       927,744        0         70              1,193,201               25,512,280
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getRawDate(Ljava-util-Date;)Ljava-lang-String;
       47,859       941,101        0         70                 19,100                  576,456
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getRawTime(Ljava-util-Date;)Ljava-lang-String;
       47,859       946,376        0         69              1,219,601               26,782,288
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-addOrderHeader(Ljava-lang-String;Ljava-math-BigDecimal;Ljava-math-BigDecimal;
       47,859       964,775        0         70                 19,462                  166,536
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getRawDate(Ljava-util-Date;)Ljava-lang-String;
       47,859       973,752        0         70                 18,398                  527,248
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getRawTime(Ljava-util-Date;)Ljava-lang-String;
       47,859       975,886        0         69                 66,838                1,239,512
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-updateCustomer(Ljava-lang-String;Ljava-math-BigDecimal;)V
       47,859     1,111,931        0         69              1,140,497               16,527,720
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-writeDataQueue(Ljava-lang-String;Ljava-math-BigDecimal;)V
       47,859     1,111,933        0         68              2,620,705               50,094,504
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-submitOrder(LDomainObjects-Order;)V
       47,859     1,112,122        0         67              3,097,963               63,460,424
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager_Skel-dispatch(Ljava-rmi-Remote;Ljava-rmi-server-RemoteCall;IJ)V
* * *  E N D  O F  R E P O R T  * * *
```

Figure 291. Query Results — Application Methods

The "Task ID" column shows that all of these application methods were running under the same thread.

The "Record Number" column shows the sequence in which these records were generated.

Note: Because these are the Java Exit event records, they are generated when the methods end.

The gaps in record numbers indicate that many other methods were executed between the application's methods. These other methods are part of the Java support and not of the PersistencyManagers application package.

The "Is Suspicious" flag is normally zero. If it is not, the statistics associated with that trace record may not be valid. For example, the flag will be non-zero if the record does not fit the call/return flow, as in an exit record that does not have a corresponding entry record.

The combination of the "Call Level" and the "Record Number" makes the call structure obvious:

- OrderEntryJdbcRMIPersistencyManager_Skel-dispatch
  - submitOrder
    - getOrderNumber
    - addOrderLine
    - getCustomerDiscount
    - updateStock
  - addOrderHeader
    - getRawDate
    - getRawTime
  - updateCustomer
    - getRawDate
    - getRawTime
  - writeDataQueue

The "Program CPU Microseconds Cumulative" is the cumulative total of the CPU consumed by a method, plus all other methods called underneath.

Within this measurement, the execution of the addOrderHeader method is one of the most expensive steps with over 1.2 seconds of CPU time. A major contributor to this is its invocation of the getRawDate method. What happens in the latter method is obviously part of the standard Java support. This will be analyzed later using the detailed query report.

Another expensive method is writeDataQueue, which is charged with 1.14 CPU seconds. It does not show any calls to other methods within the application so the overhead is probably due to the standard Java methods. This will also be analyzed with the detailed query report.

Note: How do we know that the actual overhead is occurring in the standard Java classes, instead of the methods that show up in the query, for example, getRawDate? Actually, we do not know. One way to verify this is to modify the query to use the QTRDUS (Program direct CPU microseconds). This gives the CPU usage that can be directly attributed to a method instead of an accumulation of its CPU and those of its underlying calls. However, in an object-oriented implementation such as Java, program statements within a method tend to result in calls to other methods in other classes.

If the overhead is indeed in other classes, such as the standard Java classes, then what can be done to improve them? For the same explanation given earlier,

statements within the application can be modified to improve on or to make different calls to the standard Java classes.

14.9.5.4 Query Definition — Detail Report

This query definition is similar to the previous one. However, instead of selecting only the methods that are part of the application package, all methods are selected, including those of the standard Java support. It produces detailed output so the final report can be several hundred or a few thousand pages long, depending on how complex the Java transaction is. However, it provides comprehensive information on the most CPU intensive and longest operations, and it is not difficult to find the problem areas within the application. Figure 292 shows the query definition.

```
5769QU1  V4R2M0  980228           IBM Query/400         SYSNAME      4/01/98   17:29:59
  Query . . . . . . . . . . . . . . . . JVAQRYDTL
    Library . . . . . . . . . . . . . . LLAMES
  Query text  . . . . . . . . . . . . . Detail JVXIT, Cumulative Stats
  Query CCSID . . . . . . . . . . . . . 65535
  Query language id . . . . . . . . . . ENU
  Query country id  . . . . . . . . . . US
    *** . is the decimal separator character for this query ***
  Collating sequence  . . . . . . . . . Hexadecimal
  Processing options
    Use rounding  . . . . . . . . . . . Yes (default)
    Ignore decimal data errors  . . . . . No (default)
    Ignore substitution warnings  . . . . Yes
    Use collating for all compares  . . . Yes
Selected files
  ID     File          Library      Member       Record Format
  T01    ORDER01       LLAMESPEX    *FIRST       QVPETRC
  T02    QAYPENMI      LLAMESPEX    ORDER01      QYPENMI
Join tests
  Type of join  . . . . . . . . . . . . Matched records
  Field            Test            Field
  T01.QTRDID       EQ              T02.QNMOFF
Select record tests
  AND/OR   Field           Test    Value (Field, Numbers, or 'Characters')
           T01.QTREID      EQ      'JVXIT '
   AND     T01.QTRTNM      LIKE    'QP0Z%'
Ordering of selected fields
  Field          Sort     Ascending/  Break  Field
  Name           Priority Descending  Level  Text
  T01.QTRTCT                                 Event task id
  T01.QTRTNM                                 Job/Task name
  T01.QTRSPC                                 Record is suspicious
  T01.QTRCLV                                 Program relative call level
  T01.QTRCUS                                 Program cumulative cpu microseconds
  T01.QTRCES                                 Program cumulative elapsed microseconds
  T02.QNMLNM                                 Mi long name
Report column formatting and summary functions
  Summary functions:  1-Total, 2-Average, 3-Minimum, 4-Maximum, 5-Count          Overrides
  Field          Summary   Column                                 Dec  Null      Dec  Numeric
  Name           Functions Spacing  Column Headings        Len  Pos  Cap   Len  Pos  Editing
  T01.QTRTCT                0        Task ID                 10    0
  T01.QTRTNM                2        Job                     30
                                     Task
                                     Name
  T01.QTRSPC                2        Is                       2    0
                                     Suspicious
  T01.QTRCLV                2        Call                     4    0
                                     Level
  T01.QTRCUS                2        Program CPU             20    0
                                     Microseconds
                                     Cumulative
  T01.QTRCES                2        Program elapsed         20    0
                                     Microseconds
                                     Cumulative
  T02.QNMLNM                2        Mi long name           256   V
Report breaks
  Break  New   Suppress   Break
  Level  Page  Summaries  Text
  0      No    Yes        FINAL TOTALS
Selected output attributes
  Output type . . . . . . . . . . . . . Printer
  Form of output  . . . . . . . . . . . Detail
  Line wrapping . . . . . . . . . . . . Yes
    Wrapping width  . . . . . . . . . .
    Record on one page  . . . . . . . . No
              * * * * *   E N D   O F   Q U E R Y   P R I N T   * * * * *
```

*Figure 292. Query Definition for PEX Trace *OUTFILE Details*

14.9.5.5 Detailed Query Results and Interpretation

The detailed report for the query over the first Submit Order transaction was 9264 pages long. The PRTPEXRPT command specified that the output be sorted in *TASK order, so that is the way that the query output is presented. Within each task or thread, Java events are presented in the order by which they occurred.

In the previous query, we determine that a substantial portion of the CPU was spent in the addOrderHeader method calling the getRawDate method. Another hot spot was the invocation of the writeDataQueue method. From this detailed query, it is possible to determine which standard Java methods are being used and which ones are resulting in high CPU consumption.

Within the query, the getRawData method was searched for. Figure 293 show the the first invocation of this method.

```
      Task ID   Job                          Is          Call              Program CPU              Program elapsed
                Task                         Suspicious  Level             Microseconds             Microseconds
                Name                                                       Cumulative               Cumulative
Mi long name
       47,859   QP0ZSPWT   LLAMES     004654         0         74                       1                        8
java-lang-StringBuffer-length()I
       47,859   QP0ZSPWT   LLAMES     004654         0         73                      30                      144
java-lang-String-<init>(Ljava-lang-StringBuffer;)V
       47,859   QP0ZSPWT   LLAMES     004654         0         72                      45                      192
java-lang-StringBuffer-toString()Ljava-lang-String;
       47,859   QP0ZSPWT   LLAMES     004654         0         71                   3,871                   12,944
java-text-DateFormat-format(Ljava-util-Date;)Ljava-lang-String;
       47,859   QP0ZSPWT   LLAMES     004654         0         70               1,193,201               25,512,280
PersistencyManagers-OrderEntryJdbcRMIPersistencyManager-getRawDate(Ljava-util-Date;)Ljava-lang-String;
       47,859   QP0ZSPWT   LLAMES     004654         0         72                       2                        8
java-lang-String-length()I
       47,859   QP0ZSPWT   LLAMES     004654         0         72                       2                        8
java-lang-String-charAt(I)C
```

Figure 293. Detailed Query Output — getRawDate()

Searching for methods that caused such high overhead requires paging up through the report. The methods that were found are shown in Figure 294 on page 392.

```
      Task ID   Job                          Is          Call                Program CPU             Program elapsed
                Task                         Suspicious  Level               Microseconds            Microseconds
                Name                                                         Cumulative              Cumulative
Mi long name
       47,859   QP0ZSPWT   LLAMES     004654         0         82                        1                       8
java-util-zip-ZipFile-get16([BI)I
       47,859   QP0ZSPWT   LLAMES     004654         0         82                        1                       8
java-util-zip-ZipFile-get16([BI)I
       47,859   QP0ZSPWT   LLAMES     004654         0         83                        4                       8
java-lang-String-hashCode()I
       47,859   QP0ZSPWT   LLAMES     004654         0         83                        2                       8
java-util-HashtableEntry-<init>()V
       47,859   QP0ZSPWT   LLAMES     004654         0         82                       39                     144
java-util-Hashtable-put(Ljava-lang-Object;Ljava-lang-Object;)Ljava-lang-Object;
          .
          .
          .
Repeats_for_thousands_of_pages
          .
          .
          .
          .
          .
          .
Other_entries_in_between
          .
          .
          .
       47,859   QP0ZSPWT   LLAMES     004654         0         81                  116,115               2,562,256
java-util-zip-ZipFile-readCEN()V
       47,859   QP0ZSPWT   LLAMES     004654         0         80                  116,860               2,576,384
java-util-zip-ZipFile-<init>(Ljava-lang-String;)V
       47,859   QP0ZSPWT   LLAMES     004654         0         82                        4                       8
java-lang-String-hashCode()I
       47,859   QP0ZSPWT   LLAMES     004654         0         81                       23                     104
java-util-Hashtable-get(Ljava-lang-Object;)Ljava-lang-Object;
       47,859   QP0ZSPWT   LLAMES     004654         0         80                       35                     136
java-util-zip-ZipFile-getEntry(Ljava-lang-String;)Ljava-util-zip-ZipEntry;
       47,859   QP0ZSPWT   LLAMES     004654         0         80                      328                     400
java-util-zip-ZipFile-close()V
       47,859   QP0ZSPWT   LLAMES     004654         0         79                  516,328              11,597,944
java-lang-ClassLoader-getSystemResourceAsStream(Ljava-lang-String;)Ljava-io-InputStream;
       47,859   QP0ZSPWT   LLAMES     004654         0         78                  516,341              11,597,984
java-util-SystemClassLoader-getResourceAsStream(Ljava-lang-String;)Ljava-io-InputStream;
          .
          .
          .
Other_entries_in_between
          .
          .
          .
       47,859   QP0ZSPWT   LLAMES     004654         0         77                        2                       8
java-util-Calendar-complete()V
       47,859   QP0ZSPWT   LLAMES     004654         0         76                        9                      40
java-util-Calendar-get(I)I
       47,859   QP0ZSPWT   LLAMES     004654         0         75                      341                   2,528
java-text-SimpleDateFormat-parseAmbiguousDatesAsAfter(Ljava-util-Date;)V
       47,859   QP0ZSPWT   LLAMES     004654         0         74                    1,334                   5,288
java-text-SimpleDateFormat-initializeDefaultCentury()V
       47,859   QP0ZSPWT   LLAMES     004654         0         73                   18,014                 170,720
java-text-SimpleDateFormat-initialize(Ljava-util-Locale;)V
       47,859   QP0ZSPWT   LLAMES     004654         0         72                1,137,834              25,083,040
java-text-SimpleDateFormat-<init>(Ljava-lang-String;Ljava-util-Locale;)V
       47,859   QP0ZSPWT   LLAMES     004654         0         71                1,137,861              25,083,144
java-text-SimpleDateFormat-<init>(Ljava-lang-String;)V
```

Figure 294. Detailed Query Output — Beneath getRawDate()

The problem appears to be in the repetitive invocation of the java-util-zip-ZipFile-get16([BI) and the java-util-zip-ZipFile-get32([BI) methods, which fill thousands of pages of the query.

The getRawDate method was moved to the initialize() method as part of the improvement to make the addOrderHeader a PreparedStatement. The getRawDate method was originally being invoked when creating the SQL statement string.

A similar analysis was performed on the writeDataQueue method. Most of the overhead and response time appears to be in setting up the sockets connection

through the AS/400 Toolbox for Java classes and in initializing or creating data queue objects.

14.10 Summary and Recommendations

Table 12 summarizes key performance response time measurements for several versions of the application. We list four different application versions running the same scenario:

- **Original**: The original OrderEntryJDBC application (inside VisualAge for Java)
- **TB created**: The original application after creating program objects for all the AS/400 Toolbox for Java classes (inside VisualAge for Java)
- **Modified(VAJ)**: The new application named OrderEntry, which includes performance enhancements (inside VisualAge for Java)
- **Modified**: The new application named OrderEntry, which includes performance enhancements running outside the VisualAge for Java IDE

Table 12. Order Entry Application Response Times

Version	Start Server	Sign on	Submit first order	Submit second order
Original	47 seconds	45 seconds	94 seconds	5.3 seconds
TB created	9 seconds	9 seconds	11 seconds	< 1 second
Modified(VAJ)	9 seconds	5 seconds	.7 seconds	< 1 second
Modified	9 seconds	2.5 seconds	.5 seconds	< 1 second

The following items were key to improving the performance of the OrderEntry application:

- AS/400 Toolbox for Java classes should have a Java program created. This is not by default in V4R2, as these classes are shipped without a Java program. You must run the CRTJVAPGM command to do this. This should be done with an OPTIMIZE parameter of 40. See Section 14.6.1.1, “Creating a Persistent Java Program for jar and zip” on page 371, for details. You do not have to do this for V4R3 or V4R4.
- String handling should be avoided. In this application, changing the SQL statements from Statement objects that used strings to PreparedStatement objects, which used parameters provided a major performance gain. See Section 14.6.1.3, “JDBC PreparedStatement and SQL String Manipulation” on page 374, for details.
- When using RMI, separate the naming lookup from the actual RMI calls. See Section 14.6.1.4, “Remote Method Invocation Support” on page 376, for details.
- Minimize object creation and re-use them. See Section 14.6.1.2, “Pre-Creating Data Queue Objects” on page 373, for an example.
- If you are using an RMI interface, try to prime the processing to ensure that all of the required classes are loaded and all the objects required are instantiated on the server.
- Run the CRTJVAPGM command against your program using optimization level 40.

- Use threads to allow for concurrent processing where applicable to overlap processing. In the OrderEntry application, we use a thread to establish and prime the RMI connection.
- Make your performance measurements outside of the Java Integrated Development Environment. We saw a much better response time after exporting the classes and running outside of VisualAge for Java.
- Do not run in -verbose mode because it adds overhead to the application.

When running some of the AS/400 Toolbox for Java classes on the AS/400 system, a sockets interface is used just as if they were running from a client because certain functions have not been "optimized for native." One example is the use of the AS/400 Toolbox for Java JDBC driver instead of the native JDBC driver. This situation can cause diminished performance due to additional network latency. Be aware of this anomaly.

Chapter 15. Application Performance Analysis with GUI

Since the initial publication of this redbook, other methods of analyzing server Java performance on the AS/400 system have been developed. These methods provide graphical user interfaces (GUIs) that make analysis easier.

Both methods rely on raw Performance Explorer (PEX) data. When running a Java application on the AS/400 system, the **-prof** option is not supported. PEX provides much more data and the tools provide ways of extracting meaningful information out of the raw data files. PEX measurements can be done on all AS/400 RISC systems. The procedures that are presented here are for both V4R4 and V4R3.

One of the tools is the IBM Performance Trace Data Visualizer for AS/400 (PTDV). This is a pure Java application that accesses the PEX data through JDBC. Download information is provided below.

Another facility is the Java Performance Data Converter which is packaged with the AS/400 Developer Kit for Java 5769-JV1 in V4R4. This takes a different approach, wherein it converts Java-related PEX data into the Java -prof format that is recognizable by several visual analysis tools. Another format that can be generated is Jinsight format which provides more detail than the standard -prof format. Such tools can also be downloaded from the World Wide Web and instructions are provided later.

This chapter is not an introduction or a primer on performance methodology. Therefore, you should already be familiar with performance analysis methodology, such as identifying the problem area, determining the proper environment to measure, identifying peak periods or setting up stress tests or unit tests, and formulating a data collection plan.

In a live production environment, several factors, such as system tuning and proper sizing, may also have a significant effect on performance. This chapter only covers the analysis of application issues.

15.1 Preparing the Application

There are two runtime requirements for both tools:

- The Java application should be running in direct execution mode, non-interpreted mode.
- The Java application must be enabled for performance collection, at least at the *ENTRYEXIT level.

The first requirement can be satisfied by using the Transformer on the application, either explicitly or automatically. This is explained in Chapter 13, “Java Performance and Work Management Overview” on page 323.

The second requirement can be satisfied by using the Create Java Program (CRTJVAPGM) command or the V4R4 CHGJVAPGM command. The parameter ENBPFRCOL(*ENTRYEXIT) should be specified to set hooks into the Java application, as in shown in Figure 295 on page 396.

```
                Change Java Program (CHGJVAPGM)

 Type choices, press Enter.

 Class file or JAR file . . . . . > 'Javaapps/Workshop/OrderEntryJDBCPkg/OrderEn
tryJDBC.class'




                                                                        ...
 Optimization . . . . . . . . . . .   *SAME         10, *INTERPRET, 20, 30, 40...
 Enable performance collection  . > *ENTRYEXIT    *NONE, *ENTRYEXIT, *FULL...
 Merge  . . . . . . . . . . . . . .   *RPL          *YES, *RPL
```

Figure 295. Change Java Program (CHGJVAPGM)

Since the standard Java runtime classes in the V4R4 classes.zip (sun.zip and java.zip in previous releases) are not enabled for performance collection, performance data is not accumulated for them. Instead, the resources that they consume are reported to the lowest program in the program stack that is enabled for performance collection, your application program.

Java programs associated with the V4R4 classes.zip (or sun.zip and java.zip for previous releases) should not be modified in any way, including enabling them for performance collection. Doing so may require the re-installation of the Java licensed program product.

15.2 Measuring the Transaction

Regardless of which tool is used, the measurement procedure will be similar since it is all based on PEX trace collection. However, the PEX definition may vary slightly.

15.2.1 PEX Definition — Specifying the Measurement Environment

The PEX definition is used to specify the data that would be collected by PEX. The definition is added by using the Add PEX Definition (ADDPEXDFN) command. Successful execution of this command places a physical file member with the definition name in file QAPEXDFN in library QUSRSYS. Once this definition is added, it can be used repeatedly for various measurements.

15.2.1.1 PEX Definition for PTDV

Figure 296 shows the recommended parameters when analyzing a server Java application using the IBM Performance Trace Data Visualizer for AS/400.

```
 Type command, press Enter.
 ===> ADDPEXDFN DFN(PTDV) TYPE(*TRACE) JOB(*ALL) TASK(*ALL) MAXSTG(1000000)
TRCTYPE(*SLTEVT) SLTEVT(*YES) BASEVT(*PMCO) PGMEVT(*MIENTRY *MIEXIT *MISTR *MIEND
*JVAENTRY *JVAEXIT) JVAEVT(*OBJCRT *LCKSTR) TEXT('Llames-for Performance Trace
Data Visualizer')
```

Figure 296. PEX Definition for PTDV

In Figure 296 on page 396, a member named PTDV is added to the QAPEXDFN file. The physical file member may later be browsed to determine what the various parameters were.

To reduce the data in a busy production system, specify job identifiers or generic job names rather than JOB(*ALL). The measurements for PGMEVT(*MISTR *MIEND) for MI complex instructions may not be needed because an application developer can only influence their usage but cannot modify them. PGMEVT(*MIENTRY and MIEXIT) are for measuring procedure calls to non-Java programs and may not be required either.

All of the various events such as base events, program events, and Java events are described in the Version 4 manual *Performance Tools for AS/400,* SC41-5340.

15.2.1.2 PEX Definition for JPDC

Figure 297 shows the recommended PEX definition parameters when using the Java Performance Data Converter to convert data into the -prof format.

```
 Type command, press Enter.
 ===> ADDPEXDFN DFN(JPDC) TYPE(*TRACE) JOB(*ALL) MAXSTG(1000000) SLTEVT(*YES)
PGMEVT(*JVAENTRY *JVAEXIT) TEXT('Llames-for Java Performance Data Converter')
```

Figure 297. PEX Definition for JPDC

Note that trace events are generated when a Java method is called and when a Java method is returned.

15.2.2 Starting the PEX Measurement and Running the Transaction

When the transaction is ready to run, the PEX measurement can be started. This is done through the Start PEX command, as shown in Figure 298.

```
                  Start Performance Explorer (STRPEX)

Type choices, press Enter.

Session ID . . . . . . . . . . . > SBMORD01      Name
Option . . . . . . . . . . . . .   *NEW          *NEW, *INZONLY, *RESUME
Definition . . . . . . . . . . .   ptdv          Name
```

Figure 298. Start Performance Explorer (STRPEX)

The definition name is the same one that was just added. The session ID should be a unique and meaningful identifier for this measurement. In the example, the identifier describes that this is the first Submit Order transaction. Next, run the transaction.

15.2.3 Ending the PEX Measurement

Use the End PEX (ENDPEX) command to stop the measurement and to place the raw data into several file members in a library. In Figure 299, the data is placed in default library QPEXDATA into several file members.

```
                End Performance Explorer (ENDPEX)

Type choices, press Enter.

Session ID . . . . . . . . . . . > SBMORD01      Name, *SELECT
Option . . . . . . . . . . . . .   *END          *END, *SUSPEND
Data option  . . . . . . . . . .   *LIB          *FILE, *LIB, *DLT
Data library . . . . . . . . . .   QPEXDATA      Name
Data member  . . . . . . . . . .   *SSNID        Name, *SSNID
Replace data . . . . . . . . . .   *NO           *YES, *NO
Text 'description' . . . . . . . > 'order 1 submission'
```

Figure 299. End Performance Explorer (ENDPEX)

Depending on the amount of data collected and the system model, this step may take several minutes. When ending PEX on a server model, a Model 170, or a system with a low interactive feature, this command should be submitted to batch. Otherwise, the server algorithm will result in very high processor utilization.

15.3 Analysis — IBM Performance Trace Data Visualizer for AS/400 (PTDV)

The IBM Performance Trace Data Visualizer for AS/400, or Trace Visualizer, is a Java application that can be used for performance analysis of applications running on the AS/400 system. The Trace Visualizer can be used to view program flows and to get details such as the number of instructions, number of cycles, wall clock time, and time on stack of individual procedure/method calls, threads, and jobs.

When visualizing Java application traces, additional details, such as the number and type of objects created and locked, can be displayed. The Trace Visualizer allows sorting of columns, exporting of data, and many levels of data summarization.

Note

The name does not contain the word Java. The Trace Visualizer can be used to analyze both Java and, up to a certain extent, non-Java applications.

This tool is currently available as a free download from the Web at:
http://www.alphaWorks.ibm.com/tech/ptdv

It is written entirely in Java and can run on any Java compatible platform although the data must be measured on the AS/400 system. Obviously, a system with GUI capabilities and an html browser is the best platform on which to run this tool.

The download will result in a Java class file called install.class that is saved on your workstation. This class file must be run by entering `java install` to start the Java installation program.

A README.html file and an FAQ.html file are included by the installation process. These files provide README.html detailed information about configuring and using the tool. This tool supports a lot of functions, and not all of them can be covered in this chapter.

15.3.1 Starting PTDV and Retrieving the Data

An MS-DOS batch file is provided to simplify the startup of the tool in Windows-based systems. If you accept the disclaimer, click on the appropriate button. The window shown in Figure 300 is displayed.

Note

Since PTDV is still undergoing changes, your windows may not match exactly.

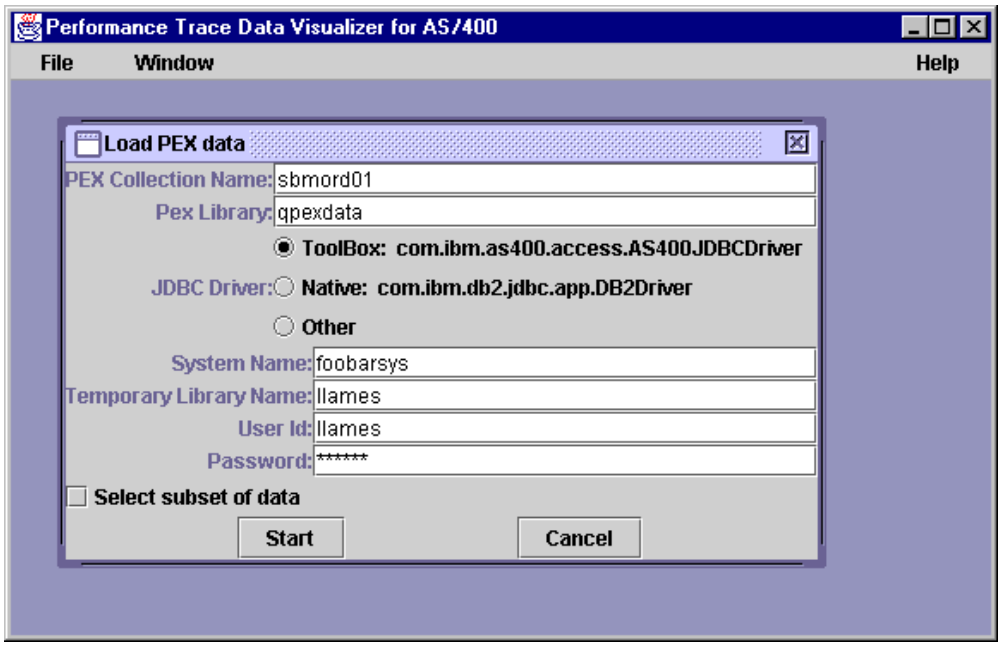


Figure 300. Selection of PEX Collection

Within this window, specify the collection name and the library where it resides. To access the AS/400 system where the data resides, you need a valid user ID and password. Also, a temporary library needs to be pre-created as a work area for the tool.

At the time this redbook was published, the checkbox to "Select subset of data" was not yet available. This may become available in the future to provide filters to reduce the amount of data processed.

Click on the **Start** button to start loading the PEX events. You will eventually arrive at a summary display as shown in Figure 301 on page 400, which describes the measurement environment.

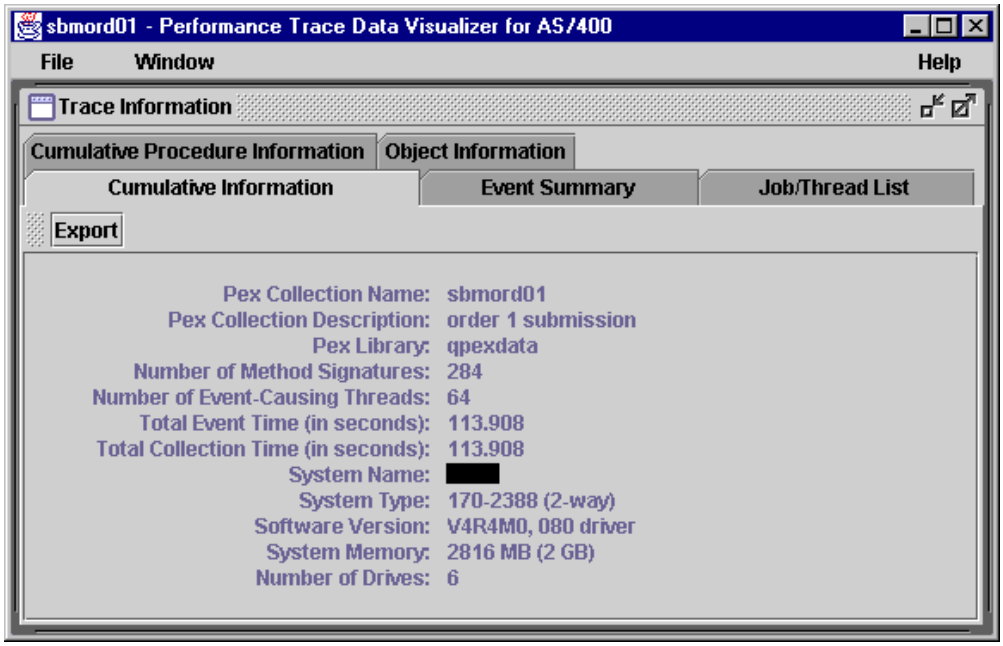


Figure 301. Summary Display Describing Measurement

15.3.2 Jobs and Threads

Click on the **Job/Thread List** tab to view a list of all of the jobs in this measurement. The jobs are grouped according to whether they caused events. The first page of the list is shown in Figure 302 on page 401. Event-causing jobs generated any of the events that were specified in the PEX definition.

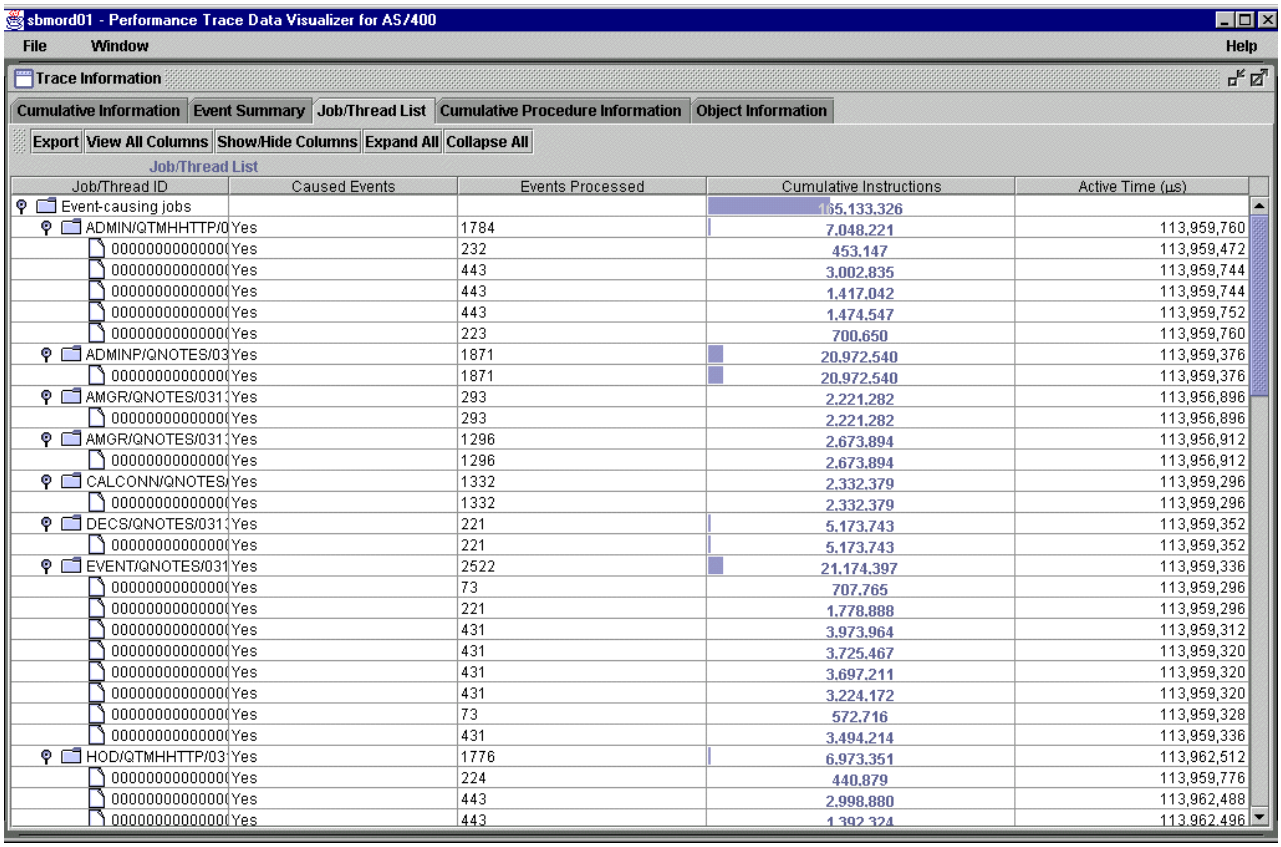


Figure 302. Job/Thread List with Event-Causing Jobs

By paging down, the job running this Java application is displayed, as shown in Figure 303 on page 402.

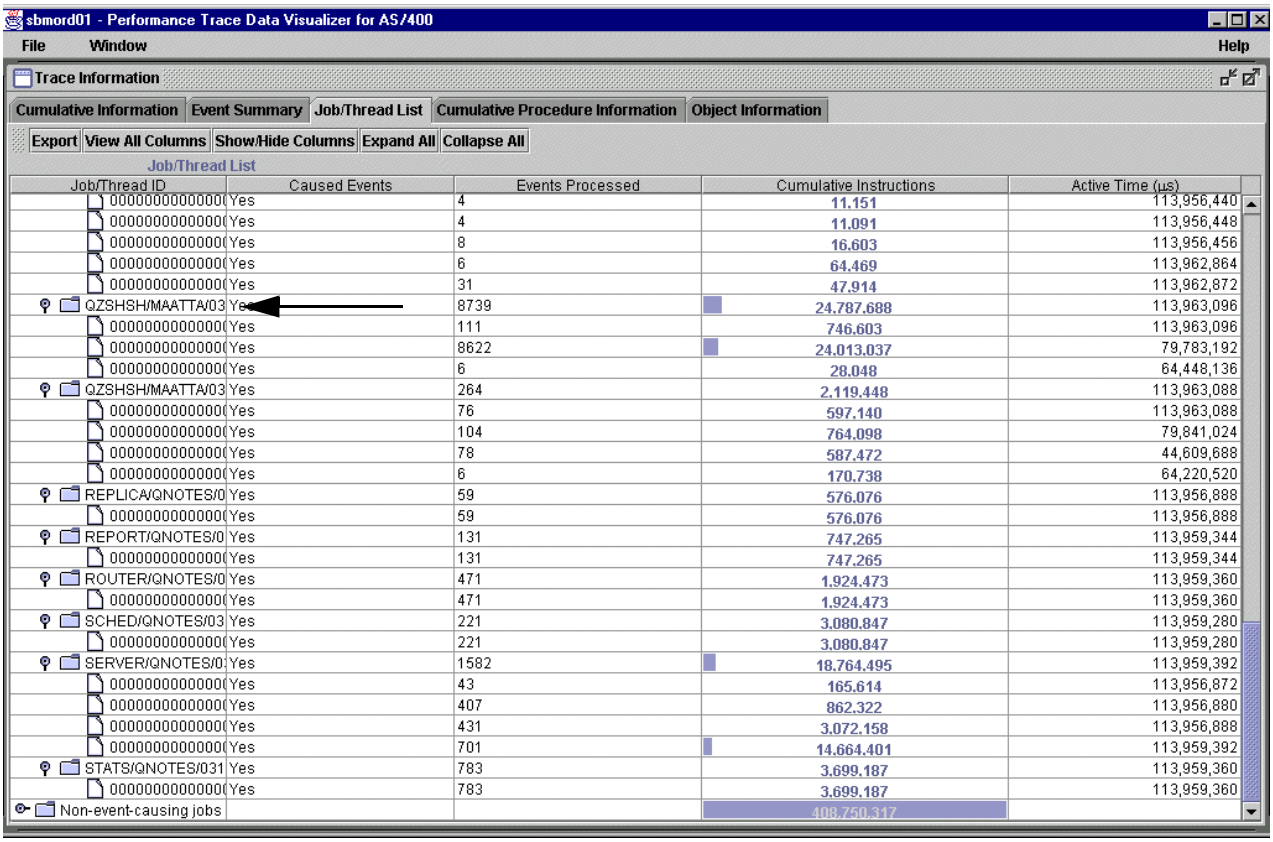


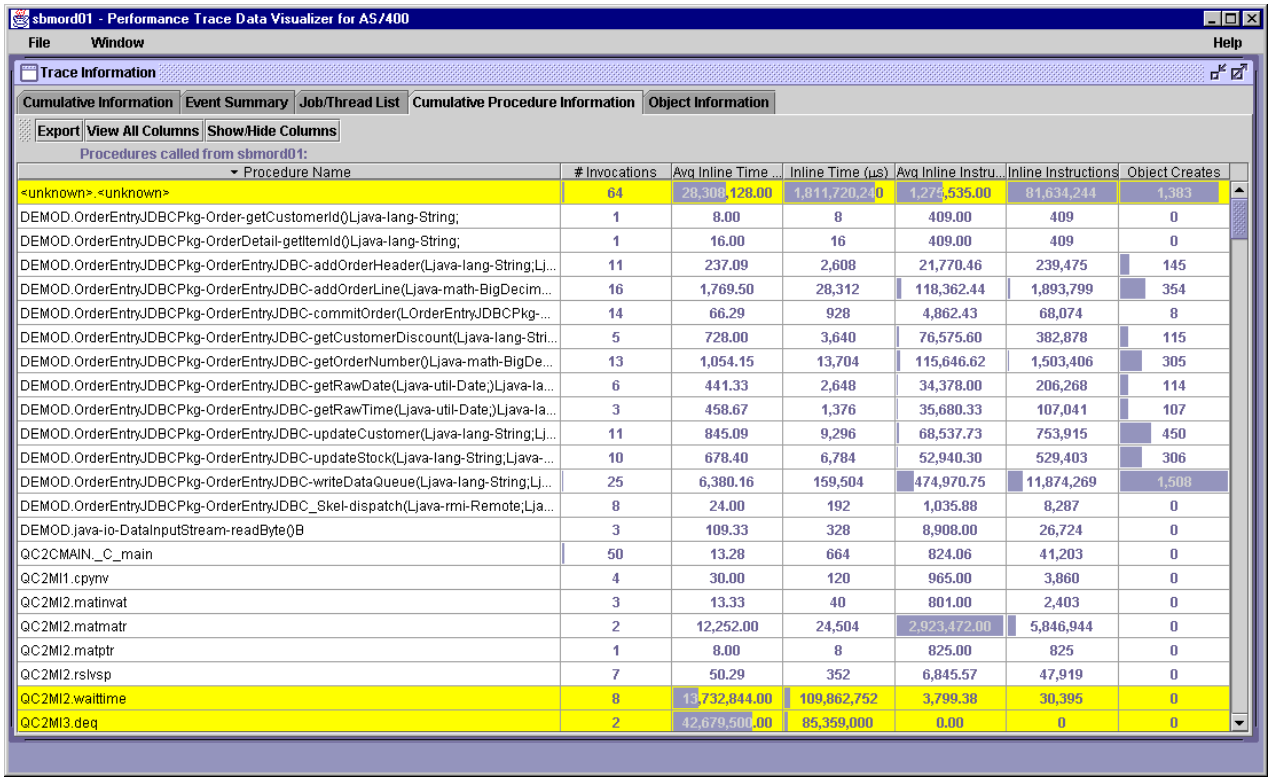
Figure 303. Job/Thread List Showing Java BCI Job and Threads

The job in question is listed as QZSHSH/MAATTA/032827 (as indicated by the arrow in Figure 303). Note the number of events that it processed, as well as the cumulative instructions. The bar within the "Cumulative Instructions" column represents the relative amount of processing done by this job.

This job had three threads running and most of the processing was done by the second thread on the list. The last row in this table shows an entry for "Non-event-causing Jobs." This list can be expanded by clicking on the horizontal key on the left. To view more details on the second thread of job QZSHSH/MAATTA/032827, double-click on the thread. This presents the details for this thread.

15.3.3 Identifying the Worst Methods

If this particular measurement only includes the jobs that are being analyzed (either because the PEX definition included only specific jobs or because the measurement was done on a dedicated system), click directly on the **Cumulative Procedure Information** tab without going through the list of jobs. A list of the various Java methods appear as shown in Figure 304 on page 403.



Procedure Name	# Invocations	Avg Inline Time ...	Inline Time (µs)	Avg Inline Instru...	Inline Instructions	Object Creates
<unknown>.<unknown>	64	28,308,128.00	1,811,720,20 0	1,275,535.00	81,634,244	1,383
DEMOD.OrderEntryJDBCPkg-Order-getCustomerId()Ljava-lang-String;	1	8.00	8	409.00	409	0
DEMOD.OrderEntryJDBCPkg-OrderDetail-getItemId()Ljava-lang-String;	1	16.00	16	409.00	409	0
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-addOrderHeader(Ljava-lang-String;Lj...	11	237.09	2,608	21,770.46	239,475	145
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-addOrderLine(Ljava-math-BigDecim...	16	1,769.50	28,312	118,362.44	1,893,799	354
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-commitOrder(LOrderEntryJDBCPkg-...	14	66.29	928	4,862.43	68,074	8
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-getCustomerDiscount(Ljava-lang-Stri...	5	728.00	3,640	76,575.60	382,878	115
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-getOrderNumber()Ljava-math-BigDe...	13	1,054.15	13,704	115,646.62	1,503,406	305
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-getRawDate(Ljava-util-Date;)Ljava-la...	6	441.33	2,648	34,378.00	206,268	114
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-getRawTime(Ljava-util-Date;)Ljava-la...	3	458.67	1,376	35,680.33	107,041	107
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-updateCustomer(Ljava-lang-String;Lj...	11	845.09	9,296	68,537.73	753,915	450
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-updateStock(Ljava-lang-String;Ljava-...	10	678.40	6,784	52,940.30	529,403	306
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC-writeDataQueue(Ljava-lang-String;Lj...	25	6,380.16	159,504	474,970.75	11,874,269	1,508
DEMOD.OrderEntryJDBCPkg-OrderEntryJDBC_Skel-dispatch(Ljava-rmi-Remote;Lja...	8	24.00	192	1,035.88	8,287	0
DEMOD.java-io-DataInputStream-readByte()B	3	109.33	328	8,908.00	26,724	0
QC2CMAIN._C_main	50	13.28	664	824.06	41,203	0
QC2MI1.cpynv	4	30.00	120	965.00	3,860	0
QC2MI2.matinvat	3	13.33	40	801.00	2,403	0
QC2MI2.matmatr	2	12,252.00	24,504	2,923,472.00	5,846,944	0
QC2MI2.matptr	1	8.00	8	825.00	825	0
QC2MI2.rslvsp	7	50.29	352	6,845.57	47,919	0
QC2MI2.waittime	8	13,732,844.00	109,862,752	3,799.38	30,395	0
QC2MI3.deq	2	42,679,500.00	85,359,000	0.00	0	0

Figure 304. Cumulative Procedure Information — Java Methods

This display also shows other programs and procedures in addition to the Java methods. The "DEMOD" string just before the Java package name of OrderEntryJDBCPkg indicates that these methods are being executed in Direct Execution mode from the native Java programs.

The gray bar in the different columns indicates the relative weight of the number in relation to the total. Among these Java methods, OrderEntryJDBCPkg.OrderEntryJDBC.writeDataQueue(String, BigDecimal) shows the widest gray bars under the "Avg Inline Instructions," "Inline Instructions," and "Object Creates" columns. This method is obviously the culprit in this transaction.

Click on the **View All Columns** tab to see many more columns, including some cumulative statistics, processor cycle counts, cycles per instruction, and so on. Click on the **Show/Hide Columns** tab for a screen to selectively add or remove columns from the display.

15.3.4 Specific Method Invocations

Since the "Inline Instructions" count for the "writeDataQueue" method is high, we need to look for more information. By double-clicking on this row and selecting the Calls tab, we are taken to a screen showing all 25 invocations of this method. As shown in Figure 305 on page 404, we can see that by far, the worst call to this method occurred on the last one.

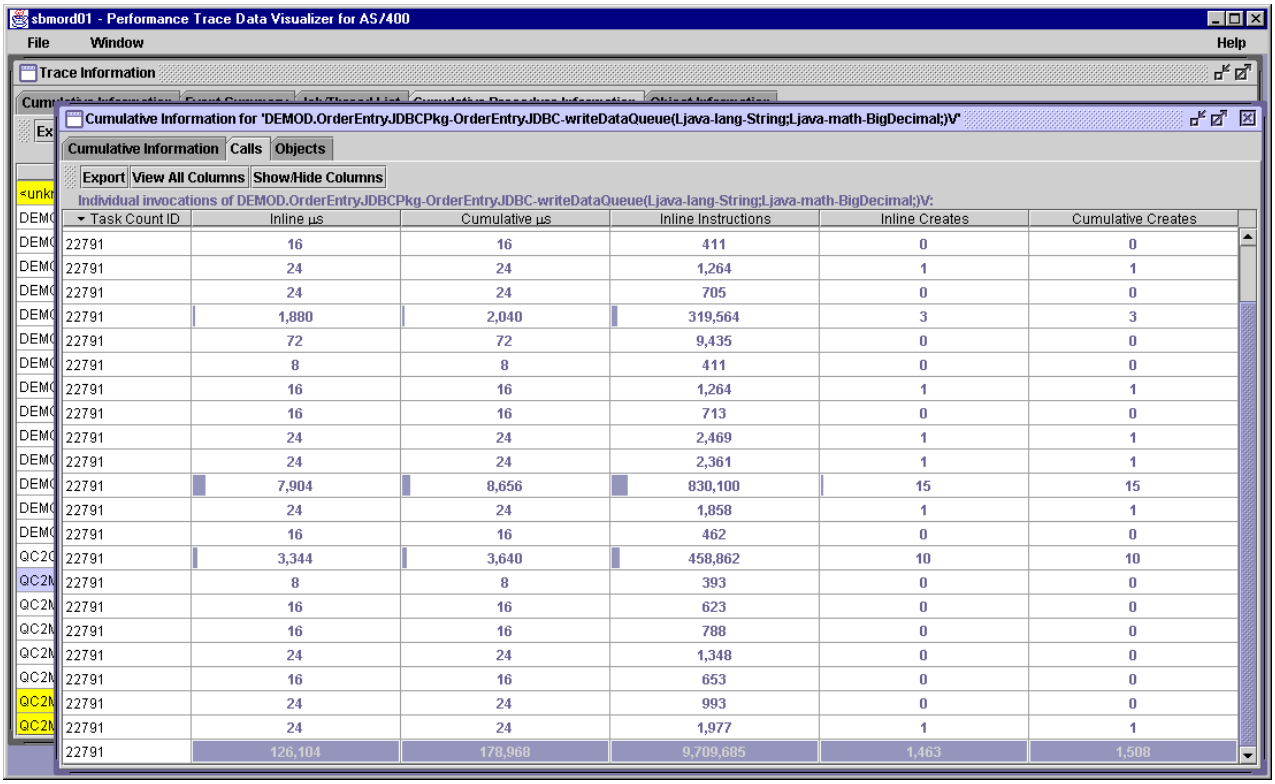


Figure 305. Calls to the writeDataQueue Method (the Last One Was the Worst)

15.3.5 Objects Created within Method Invocation

The last row shows that in the last invocation of this method, many objects were created. Double-click on the **Inline Creates** element for this row. We are taken to another table where the objects that were created are listed for this particular invocation. This is shown in Figure 306 on page 405.

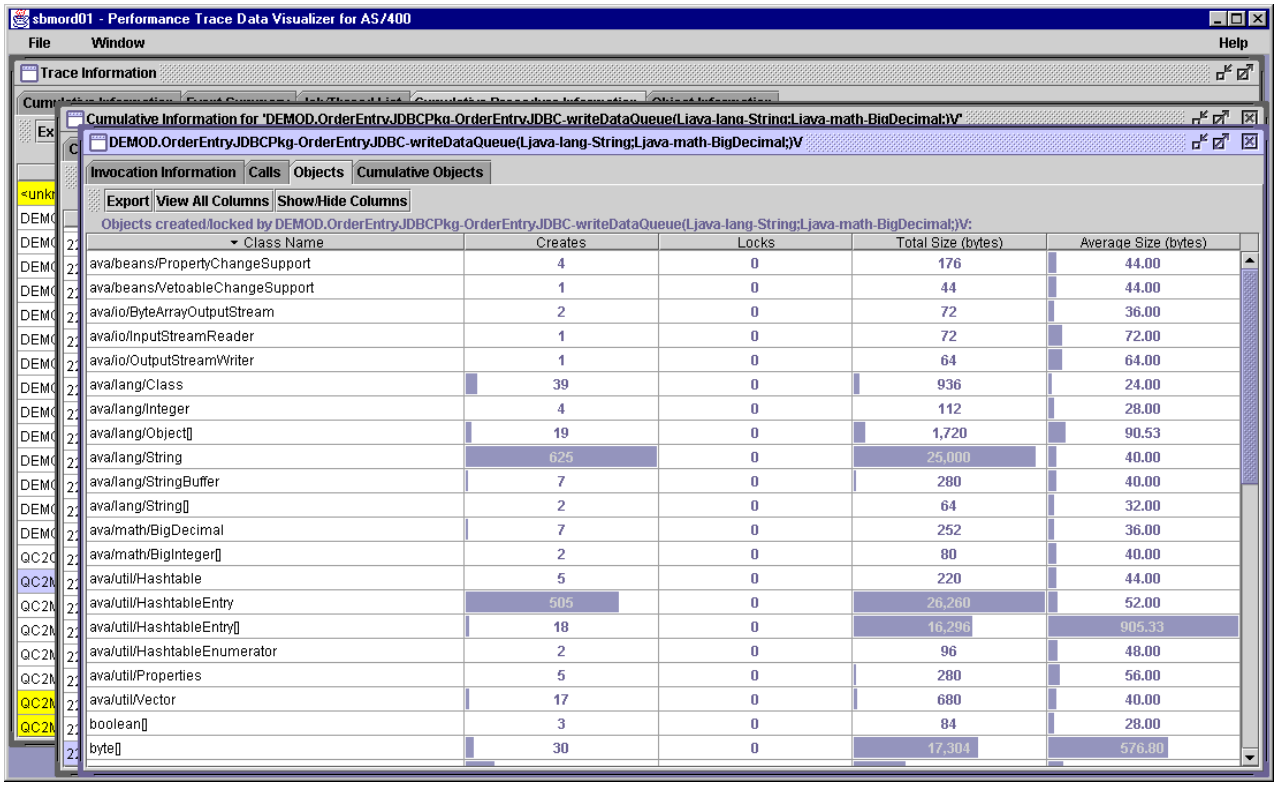


Figure 306. Objects Created in the Worst writeDataQueue() Invocation

This display shows that the worst object instantiations were on String and HashtableEntry objects. For details about how the writeDataQueue method was improved, see Section 14.6.1.2, “Pre-Creating Data Queue Objects” on page 373.

PTDV provides a much better way of analyzing AS/400 Java applications. While some technical professionals still prefer a report-based analysis, the GUI approach provides significant productivity benefits. Not only is it more intuitive, but it is also more iterative.

15.4 Analysis — Java Performance Data Converter (JPDC) and IBM Jinsight

Because the Java runtime -prof option is not supported on the AS/400 system, a way to use generic GUI tools to analyze AS/400 Java applications was needed. Adding another monitor to handle -prof was not practical since the existing PEX facility already measures much more than any profiling option provides.

The practical approach was to take the raw PEX data and run it through a program to convert the relevant data into profile format. Therefore, JPDC was developed.

Actually, there are two profile output formats supported by JPDC. These formats are:

- The *general format* is the same as running the -prof option on other JVMs. A trace file in this format can be read by generic tools such as Hyperprof.

  Note: The format is not consistent from Java 1.1 to Java 2.

- The *Jinsight format* is more granular. Unlike the general format, where statistics are aggregated by method within thread, the chronological sequence of events is maintained in the Jinsight format.

JPDC is available in both V4R4 and V4R3. Note the following points:

- In V4R4, it is part of the no-charge licensed program product 5769-JV1 AS/400 Developer Kit for Java. It is then installed as a jar file in the directory structure /QIBM/ProdData/Java400/ext/JPDC.jar.
- In V4R3 systems, it can be installed as PTF SF55565. It will then reside as a jar file in the directory structure /QIBM/ProdData/Java400/JPDC.jar.

The IBM Jinsight Visualizer tool to analyze the Jinsight format is currently available as a free download from the IBM alphaworks Web site at: http://www.alphaworks.ibm.com/tech/jinsight

Follow the instructions to download and install it on your graphical workstation.

IBM Jinsight has several views and functions. Only a subset is presented in this section. A important advantage that the Jinsight trace format and tool have over other approaches is that Jinsight does not aggregate the data and maintains the chronological sequence of method calls. Detailed documentation is provided with the tool in HTML format.

15.4.1 Converting PEX Data through JPDC

After going through the PEX measurement procedures, the PEX data can be run through JPDC. JPDC is simply a Java application and can be executed using any of the techniques described in Chapter 13, “Java Performance and Work Management Overview” on page 323. It uses the native JDBC driver to access the PEX data on the *LOCAL relational database.

The parameters for running JPDC are:

- The class file is com.ibm.as400.jpdc.JPDC.
- The CLASSPATH is /QIBM/ProdData/Java400/ext/JPDC.jar for V4R4 and /QIBM/ProdData/Java400/JPDC.jar for V4R3.
- The first parameter specifies the format of the trace file that is generated:
  - **Jinsight**: IBM Jinsight data format is generated.
  - **General**: The -prof format is produced.
- The second parameter specifies the name of the PEX collection member. Continuing with the example, the value should be SBMORD01. The PEX data should be in the default library QPEXDATA. As of this writing, a library-qualified member name is not valid.
- The third parameter is the directory/name of the trace output file. This defaults to the current directory in the IFS. Using the Jinsight output, as specified in the first parameter, has an extension of .trc.

- The fourth parameter is the name of the relational database that is located at *LOCAL. Use the Work with Relational Database Directory Entries (WRKRDBDIRE) command to find the name of this relational database. Although this parameter may indicate that PEX data located in other systems can be retrieved, this was not the case at the time this was written.

Figure 307 shows using JPDC from an AS/400 command line.

```
 ===> JAVA CLASS(com.ibm.as400.jpdc.JPDC) PARM(Jinsight sbmord01 '/llamesRedBk
/sbmord01' myrdbname) CLASSPATH('/QIBM/ProdData/Java400/ext/JPDC.jar')
OPTION(*VERBOSE)
```

Figure 307. Using JPDC from an AS/400 Command Line

This conversion generated a trace file sbmord01.trc in directory llamesRedBk. This file serves as the input to Jinsight analysis.

15.4.2 Starting the Jinsight Visualizer to Analyze JPDC Trace

To access the trace file sbmord01.trc from your graphical workstation, either map a network drive to your AS/400 IFS or copy the file to your workstation hard drive.

Start the Jinsight Visualizer tool. This presents the Jinsight control panel shown in Figure 308.

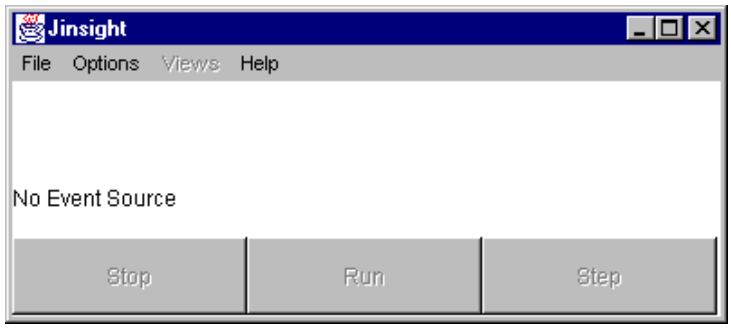


Figure 308. Jinsight Visualizer Startup Control Panel

To access the trace file, click on **File**. Then, click on **Open**, and navigate towards the trace file. Select and open the file.

15.4.3 The Histogram View

To display another window that shows the chronological sequence of events within the application, click on **Views—>Histogram**. This window has two modes:

- Object mode shows the sequence in which objects are created and garbage is collected within the application. This is the default mode as indicated by the radio button at the bottom of this view. Figure 309 on page 408 shows this window before the trace of the application has executed.

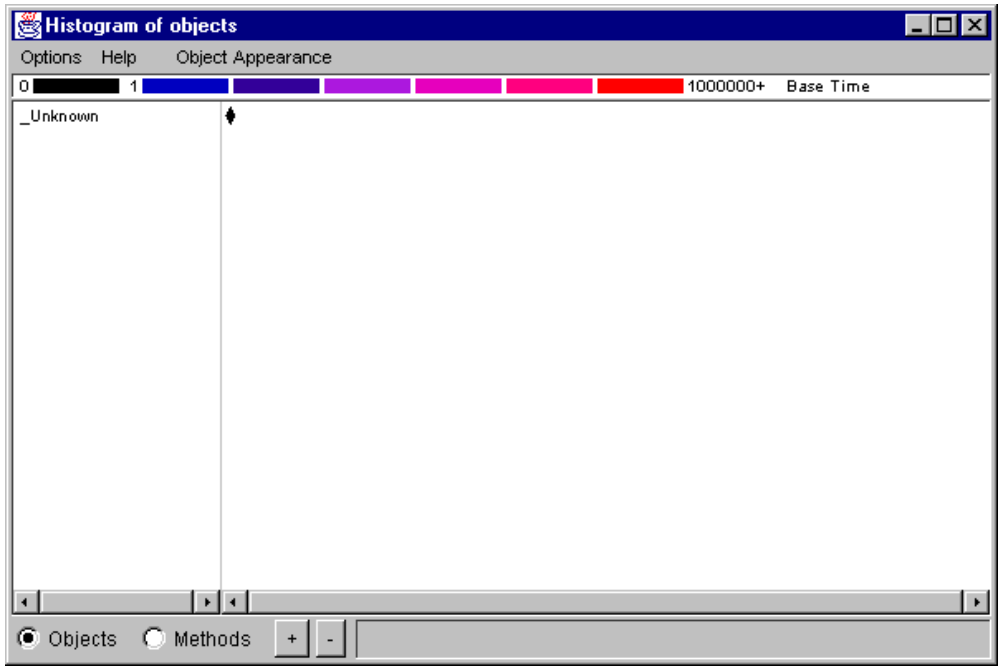


Figure 309. Histogram View of Objects before the Trace Runs

Click on the **Run** button in the Control Panel to run the trace. The histogram shows how the application ran. Unfortunately (or fortunately), this was a relatively fast running transaction so the simulated execution of the Java application finished quickly. A much longer measurement is more interesting to watch. The final window is shown in Figure 310.

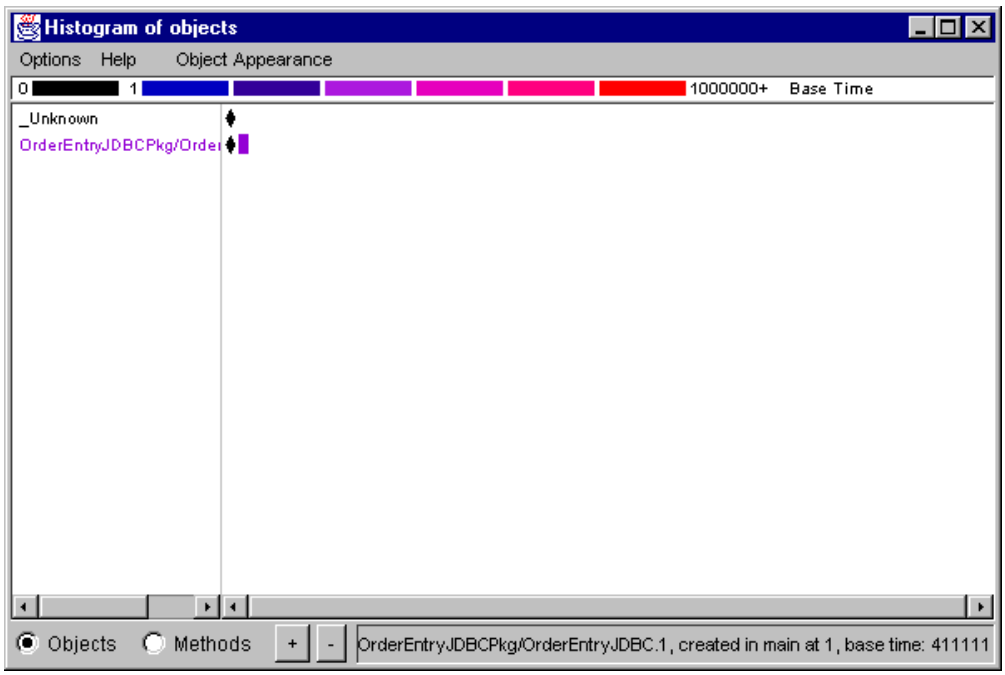


Figure 310. Histogram View of Objects after the Trace Runs

The left column gives the name of the class, OrderEntryJDBC. The diamond right next to it represents the class object. Any rectangles to the right of it represent every instance of the class. In this case, there was only one instance of this class that was measured (recall that the Sun classes were not enabled for performance collection).

The value of this window is that it shows the amount of object instantiation and garbage collection for various classes that were enabled for performance collection.

The rectangle representing each object instance changes to a white background when garbage is collected. By the time this trace ended, this OrderEntryJDBC object was still alive, meaning it had not yet been garbage collected.

This window has cursor-sensitive features, referred to by the tool designers as a "fly-over" technique. When this screen was captured, the cursor was over the first and only rectangle for the OrderEntryJDBCPkg.OrderEntryJDBC object. Note the object details that are provided at the bottom of the window.

By right-clicking on any of the instances of an object, a menu is displayed that allows more details to be provided, such as Who Calls Object, Who Creates Object, Object Calls Whom, Object Creates Whom, and so on.

- Method mode is displayed when the radio button for "Methods" at the bottom left corner of the window is clicked.

  In the window shown in Figure 311, the different application methods are represented by the rectangles to the right of the class name. In a graphical workstation, the color indicates the relative duration of the method call.

  This display was captured while the cursor was over the second method or rectangle. Details about this method call to getOrderNumber are shown at the bottom of the window.

  Also note the arrows between the rectangles. These appear when a right-click is done on a method, and the "Show Caller" or "Show Callee" menu option is selected.

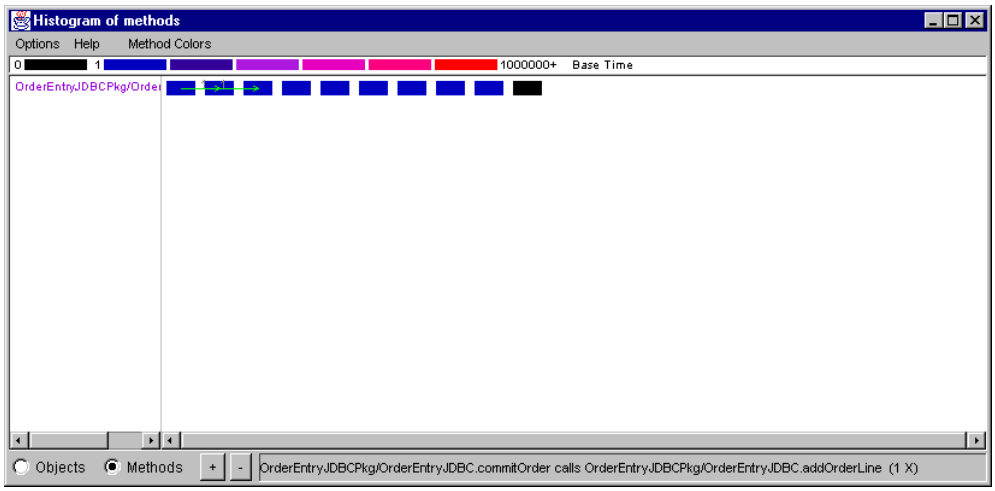


Figure 311. Histogram View of Methods after the Trace Runs

The value of this window is that it shows the relative duration of a method's execution, based on a color coding scheme. More precise durations are shown by bringing the cursor over a rectangle.

15.4.4 The Execution View

This view is very useful in determining the relative durations of the time spent in every method call within the application. It also indicates the call hierarchy.

The window is divided vertically by thread. Figure 312 on page 411 shows an example. The left half of the window shows the garbage collection thread, marked by "GC" on the upper left corner of the display. The right half of the window represents the application thread, labelled as "main" at the top of the column.

The vertical bars represent the time that a method is active within that thread. The horizontal lines represent the start of a method call. The name of the called method is visible right below the horizontal line.

The execution view also uses the fly-over capability where detailed information is provided on the method over which the cursor is located. In Figure 312 on page 411, the cursor is over the left vertical bar. According to the information at the bottom of the window, this vertical bar represents the commitOrder method within the OrderEntryJDBC class.

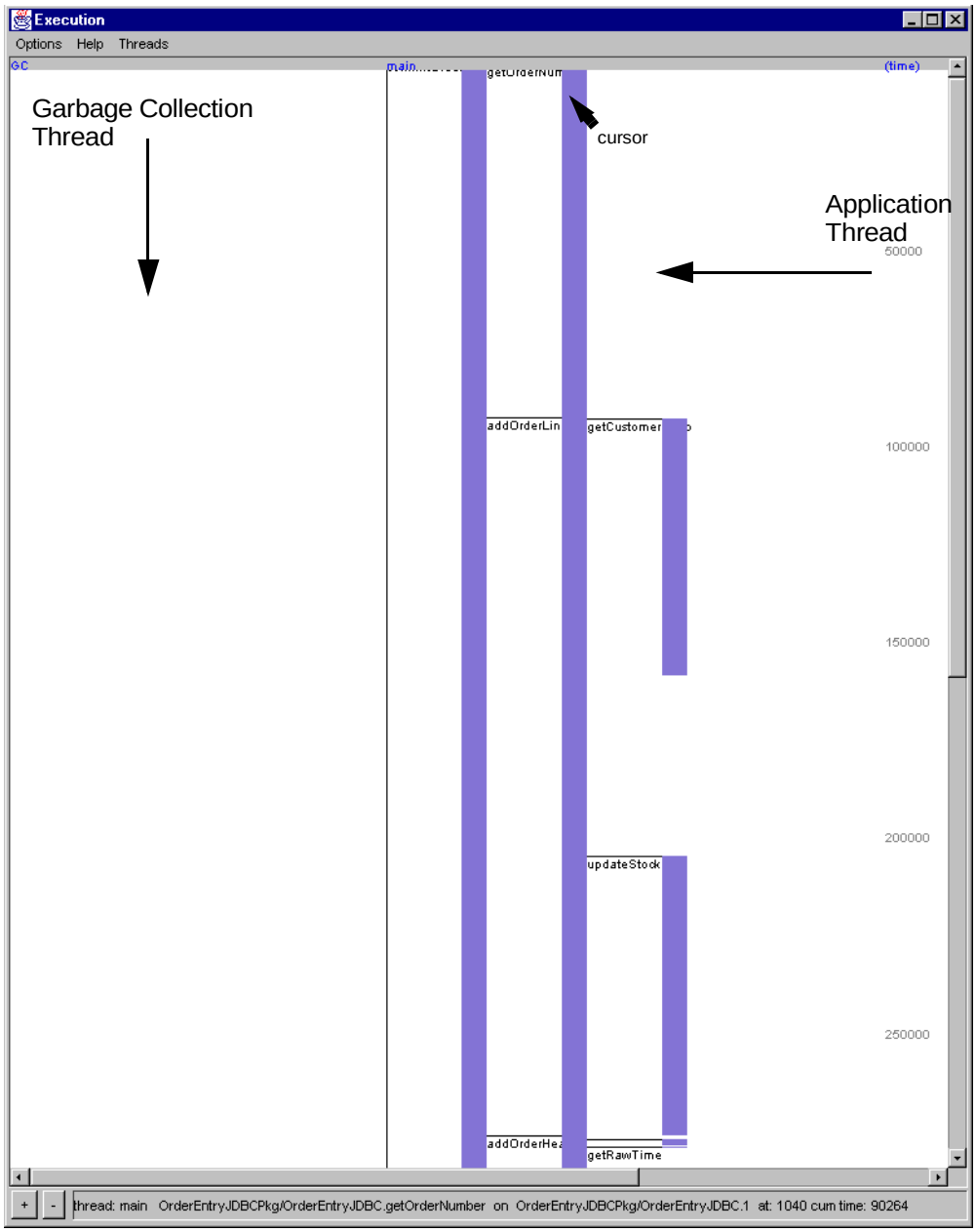


Figure 312. Execution View of Methods after the Trace Runs — commitOrder

The notation at the bottom, ". . . getOrderNumber on OrderEntryJDBCPkg/OrderEntryJDBC.1 at: 1040 cum time: 90264" provides significant statistics regarding this method call to the getOrderNumber method. These notes include:

- ". . . getOrderNumber on OrderEntryJDBCPkg/OrderEntryJDBC.1" signifies that this method call was done within the first instance of the OrderEntryJDBC class.
- ". . . at 1040" signifies that this method was started at relative time 1. The relative time scale is shown on the vertical axis on the right side of the window
- ". . . cum time: 90264" signifies that this method was active for that number of time units.

Consider this example. In Figure 313, the cursor is positioned over the second vertical bar, just below the horizontal line marked as "addOrderLine."

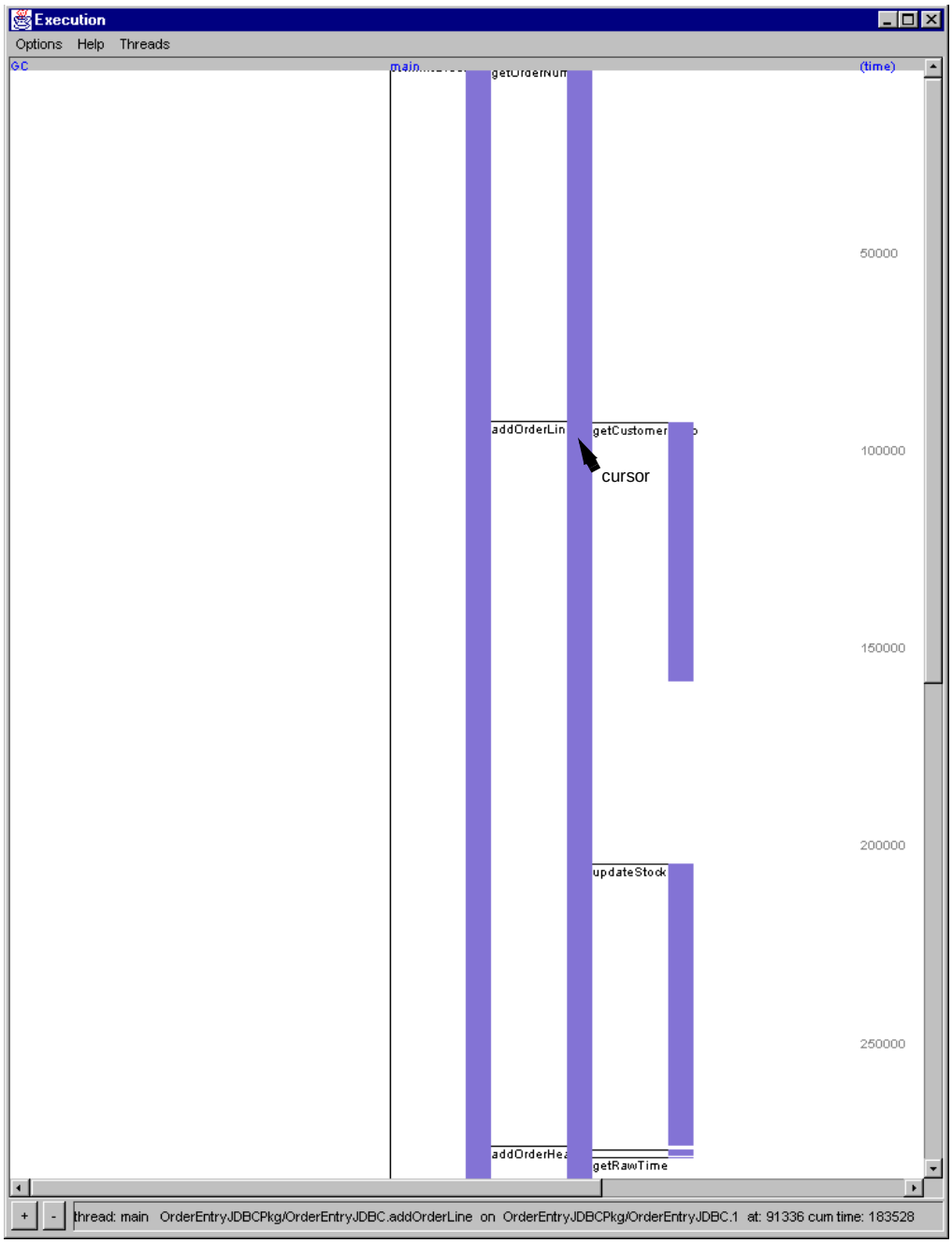


Figure 313. Execution View of Methods after the Trace Runs — addOrderLine

The notation at the bottom, ". . . addOrderLine on OrderEntryJDBCPkg/OrderEntryJDBC.1 at: 91336 cum time: 183528" provides the statistics for this method call:

- ". . . addOrderLine on OrderEntryJDBCPkg/OrderEntryJDBC.1" indicates that this method call to addOrderLine was done by the first instance of the OrderEntryJDBC class.
- ". . . at 91336" shows that this method was started at relative time stamp of 91336. Compare the position of the horizontal line "addOrderLine" with the scale on the right. It is just before the 100000 mark.
- ". . . cum time: 183528" shows that this method was active or in the stack for 183528 relative time units. However, as the window shows, a fairly large portion of the time was spent processing methods getCustomerDiscount() and updateStock().

With the Execution View, it is possible to identify the offending methods and address them accordingly using the various recommendations that were presented in the previous chapters.

The Jinsight Tool provides several other views and functions that were not presented here. The documentation that is included in the software provides explanations on using these other facilities.

15.5 Deciding Which Tool to Use

Based on the examples that were presented here, either the IBM Performance Trace Data Visualizer for AS/400 (PTDV) or the Java Performance Data Converter (JPDC)-Jinsight approaches provide the performance analyst with the offending methods. There are other criteria which can be used to determine which one is appropriate for an application environment:

- PDTV
  - Can be used to analyze applications that do not have Java events.
  - Does not require an intermediate step to convert the data into a different format. It is used to access the PEX data directly.
- JPDC-Jinsight
  - Provides more granularity and retains the chronological sequence of events. It does not aggregate or accumulate statistics by method.
  - Provides a more graphical environment; much more than tables and horizontal bars.

  If JPDC is used to generate "general" trace data format rather than Jinsight format, the resultant trace can be accessed with other generic GUI tools.

In the end, it is simply a matter of preference. We recommend that you learn the capabilities of both tools. Each provides significant productivity benefits over printed text reports.

Appendix A. Example Programs

You can download the Java client programs and the AS/400 programs and libraries that are used in this redbook from the Internet. These examples were developed using VisualAge for Java Enterprise edition. OS/400 V4R2 or later is required. OS/400 V4R4 is required for the SQLJ and JNI examples. These components are available:

- AS/400 RPG code
- AS/400 databases
- AS/400 Java code
- Client Java code

Important

These example programs have not been subjected to any formal testing. They are provided "as is". They should be used for reference only. Please refer to the Special Notices section at the back of this document for more information.

A.1 Downloading the Files from the Internet

To use these files, download them to your personal computer from the Internet Web site. See the file named **README.TXT**. It contains instructions for restoring the AS/400 libraries, the VisualAge for Java examples, and runtime notes.

The Web address to access is: www.redbooks.ibm.com

In the left panel, click on **Additional Materials**. Follow the instructions on the page that appears so you can access the list of available downloads. When the downloads page appears, scroll down until you find **SG242163** and select it. Then, download the files to your system.

Appendix B. Special Notices

This publication is intended to help anyone that needs to use Java for building AS/400 applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by the AS/400 Developer Kit for Java (program Product 5769-JV1) or the AS/400 Toolbox for Java (Program Product 5769-JC1). See the PUBLICATIONS section of the IBM Programming Announcement for the AS/400 Developer Kit for Java (5769-JV1) and the AS/400 Toolbox for Java (5763-JC1) for more information about what publications are considered to be product documentation.

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, 500 Columbus Avenue, Thornwood, NY 10594 USA.

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 information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. 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.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating

environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

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

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

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

AFP
AIX
AS/400
AT
COBOL/400
Common User Access
CT
DB2
FFST
GDDM
IBM ®
Integrated Language Environment
Language Environment
Network Station
OfficeVision
Operating System/400
OS/2
OS/400
RS/6000
SP
System/36
System/38
System/390
VisualAge
WebSphere
XT
400

The following terms are trademarks of other companies:
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. (For a complete list of Intel trademarks see www.intel.com/tradmarx.htm)

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

SET 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.

Appendix C. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

C.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 421.

- *Building AS/400 Client/Server Applications with Java*, SG24-2152

C.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at `http://www.redbooks.ibm.com/` for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	Collection Kit Number
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037

C.3 Other Publications

These publications are also relevant as further information sources:

- *Java in a Nutshell*, ISBN 1-56592-183-6
- *Java Developer's Reference*, ISBN 1-57521-129-7
- *Object Oriented Technology: A Manager's Guide*, ISBN 0-201- 56358-4
- *ILE RPG for AS/400 Programming Guide*, SC09-2507
- *OS/400 Work Management*, SC41-5306
- *Performance Tools/400*, SC41-5340
- *AS/400 Database Programming*, SC41-5701
- *Distributed Database Programming*, SC41-5702
- *Qshell Interpreter* (See http://www.as400bks.com/ on the AS/400 Internet book server. **Note:** This book is only available on the Internet.)
- *AS/400 Java Developer Kit for Java* (See http://www.as400bks.com/ on the AS/400 Internet book server. **Note:** This book is only available on the Internet.)
- *AS/400 Toolbox for Java* (See http://www.as400bks.com/ on the AS/400 Internet book server. **Note:** This book is only available on the Internet.)

- Lemay, Laura. *JAVA 1.1 Interactive Course*. The Waite Group, 1997 (ISBN 1-57169-083-2).
- Couthard, Phillip and Farr, George. *Java for RPG Programmers*. IBM Press, 1998 (GK2T-9890; available on diskette).
- Cooper, Alan. *About Face*. Foster City, CA: IDG Books Worldwide.
- Englander, Robert. *Developing JavaBeans*. Sebastopol, CA: O'Reilly Associates, Inc., 1997.
- Galitz, Wilbert O. *User-Interface Screen Design*. New York, New York: Wiley-QED, 1993.
- Mayhew, Deborah J. *Principles and Guidelines in Software User Interface Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.
- *Object-Oriented Interface Design* (IBM Common User Access™ Guidelines), Carmel, IN: Que, 1992.
- Preece, Jenny. *Human Computer Interaction*. Reading, Massachusetts: Addison-Wesley, 1994.
- Vanderburg, Glenn. *Tricks of the Java Programming Gurus*. Indianapolis, IN: Sams.net Publishing, 1996.

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** `http://www.redbooks.ibm.com/`

  Search for, view, download, or order hardcopy/CD-ROM redbooks from the redbooks Web site. Also read redpieces and 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.

- **E-mail Orders**

  Send orders by e-mail including information from the redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at `http://w3.itso.ibm.com/` and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access `MyNews` at `http://w3.ibm.com/` for redbook, residency, and workshop announcements.

IBM Redbook Fax Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

AFP	advanced function printing
APA	all points addressable
AWT	Abstract Windowing Toolkit
CORBA	Common Object Request Broker Architecture
COM	Component Object Model
CPW	Commercial Processing Workload
EAB	Enterprise Access Builder
EJB	Enterprise JavaBean
DAX	Data Access Builder
DDM	Distributed Data Management
DPC	Distributed Program Call
FFST	First Failure Support Technology
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IDL	Interface Definition Language
IIOP	Internet inter-ORB protocol
ITSO	International Technical Support Organization
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Development Toolkit
JFC	Java Foundation Classes
JIT	Just in Time Compiler
JVM	Java Virtual Machine
MI	Machine Interface
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OOP	Object Oriented Programming
PTF	Program Temporary Fix
RAD	Rapid Application Development
RMI	Remote Method Invocation
SCS	SNA Character Set
SLIC	System Licensed Internal Code
SSL	secure sockets layer
TIMI	Technology Independent Machine Interface
UML	Unified Methodology Language
URL	Universal Resource Locator
VCE	Visual Composition Editor
WWW	World Wide Web

Index

Symbols

Numerics

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

ITSO Redbook Evaluation

Building AS/400 Applications with Java
SG24-2163-02

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to `redbook@us.ibm.com`

Which of the following best describes you?
_ **Customer** _ **Business Partner** _ **Solution Developer** _ **IBM employee**
_ **None of the above**

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

_____

_____

_____

_____

_____

SG24-2163-02

Printed in the U.S.A.

Building AS/400 Applications with Java

SG24-2163-02