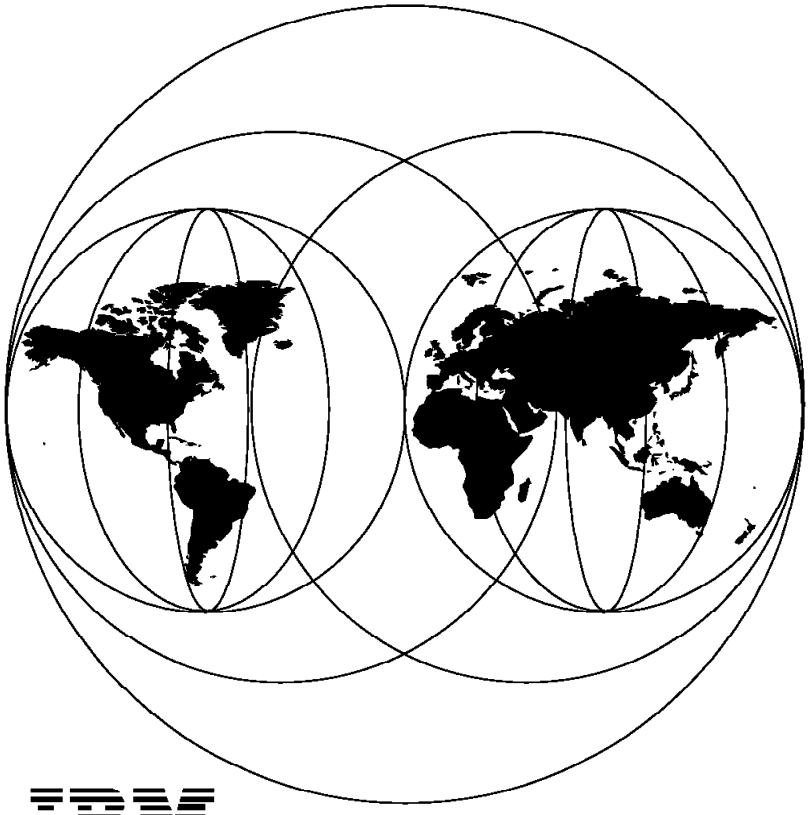
SG24-2163-00

Building AS/400 Applications with Java

February 1998



IBM

International Technical Support Organization
Rochester Center

International Technical Support Organization

SG24-2163-00

Building AS/400 Applications with Java

February 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special Notices” on page 213.

First Edition (February 1998)

This edition applies to Version 4 Release 2 of AS/400 Developer Kit for Java (Product Number 5769-JV1) and Version 3 Release 2 of AS/400 Toolbox for Java (Program Number 5763-JC1) for use with Version 4 Release 2 of the OS/400 Operating System (5769-SS1).

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

Contents

Preface

In the past year, Java has become the hot new programming language. The reasons for Java's popularity are its portability, robustness, and its ability to produce Internet enabled applications. This redbook is intended for customers and service providers who have a 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.

We cover how you can use Java and the AS/400 system to build server applications and client/server applications for the new network computing paradigm.

This redbook provides many practical programming examples with detailed explanations on how they work. We also describe how to take legacy RPG applications and modernize them in a practical and evolutionary way using client and server Java examples. These examples are available to download from our Internet site. Tips are given on how to improve Java performance.

This redbook gives you a fast start on your way to using Java with the AS/400 system.

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.

Neil Willis is a Senior Systems Engineer with IBM Australia, currently on assignment at the ITSO Rochester. Neil has been with IBM for 12 years, working with IBM customers on the S/38 and AS/400 platforms. His areas of specialty include Performance Tuning, Capacity Planning, Application Development, Communications, and the Internet. His professional qualifications include a degree in Computer Science, a degree with honors in Electrical Engineering, and a Masters Degree in Information Systems.

Bob Maatta is a Senior Software Engineer from the United States at the International Technical Support Organization, Rochester Center. He writes extensively and teaches IBM classes worldwide on all areas of AS/400 client/server. Before joining the ITSO 3 years ago, 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 for the last 10 years.

Simon Coulter is a principal of FlyByNight Software, an Australian AS/400 consulting company specializing in data replication, data warehousing, AS/400 education, and programming services. Previously, he was a Software Engineer with IBM Australia for 8 years and has 13 years experience in the computer industry. He may be contacted through e-mail at shc@flybynight.com.au.

Jim Fair is a Staff Software Analyst at the IBM AS/400 Support Line Organization in Rochester, MN. He has been working in the area of application support the past 9 years with specific focus on object-oriented technology for the last 4 years. He

provides expert technical advice and support to IBM's Customers and Business Partners in the area of application support and enablement.

Pierre Goudet is a Senior I/T Specialist at the AS/400 Technical Support Center in Paris, France. He has worked for IBM for over 25 years and has held several positions as a field technical support representative and as a Consultant in Application Development. He provides expert technical advice to IBM's Customers and Business Partners and teaches IBM classes on AS/400 Application Development, Client/Server, Network Computing, and Java. Pierre is currently the AS/400 Java Technical Support Specialist for EMEA West Region.

Leonardo Llames is a Senior Software Analyst at the IBM Advanced Technical Support Organization in Rochester, MN. He has been working in the areas of performance analysis, application, and database design for IBM customers for the past 9 years and currently works in object-oriented technology and Java. Prior to his move to Rochester, he was involved in application development with IBM Business Partners.

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

Jennifer Bigus
IBM Rochester Laboratory

Phil Coulthard
IBM Toronto Laboratory

John Gojnich
Computer Results Team
Sydney, Australia

Greg Hoffa
IBM Rochester Laboratory

Mike Jacobs
IBM Rochester Laboratory

Mike McRoberts
IBM Rochester Laboratory

Mark Megerian
IBM Rochester Laboratory

Barbara Morris
IBM Toronto Laboratory

Gary Mullen-Schultz
IBM Rochester-Partners in Development

Conny Nordstrom
IBM Sweden

John Orbeck
IBM Rochester Laboratory

Pramod Patel
IBM Toronto Laboratory

Jon Peterson
IBM Rochester Laboratory

Pamela Tse
IBM Toronto Laboratory

Jerry Wille
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 225 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM Intranet users `http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@vnet.ibm.com`

Chapter 1. Introduction

In this redbook, we cover Java on the AS/400 system by focussing on three main areas:

- An overview of Java on the AS/400 system
- Installation of Java on the AS/400 system
- Building AS/400 applications with Java

We first provide an overview of Java and its key components. We then cover how Java is implemented on the AS/400 system and provide details about AS/400 specific considerations.

We next show details about how to install Java on the AS/400 system and how to set up the Java environment.

Finally, 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 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 feature to allow the client Java program to interface with the AS/400 server Java program. The migration scenario is shown in the following diagrams.

We start with an existing RPG order entry application. This application is described in more detail in Chapter 5, “Overview of the Order Entry Application” on page 93.

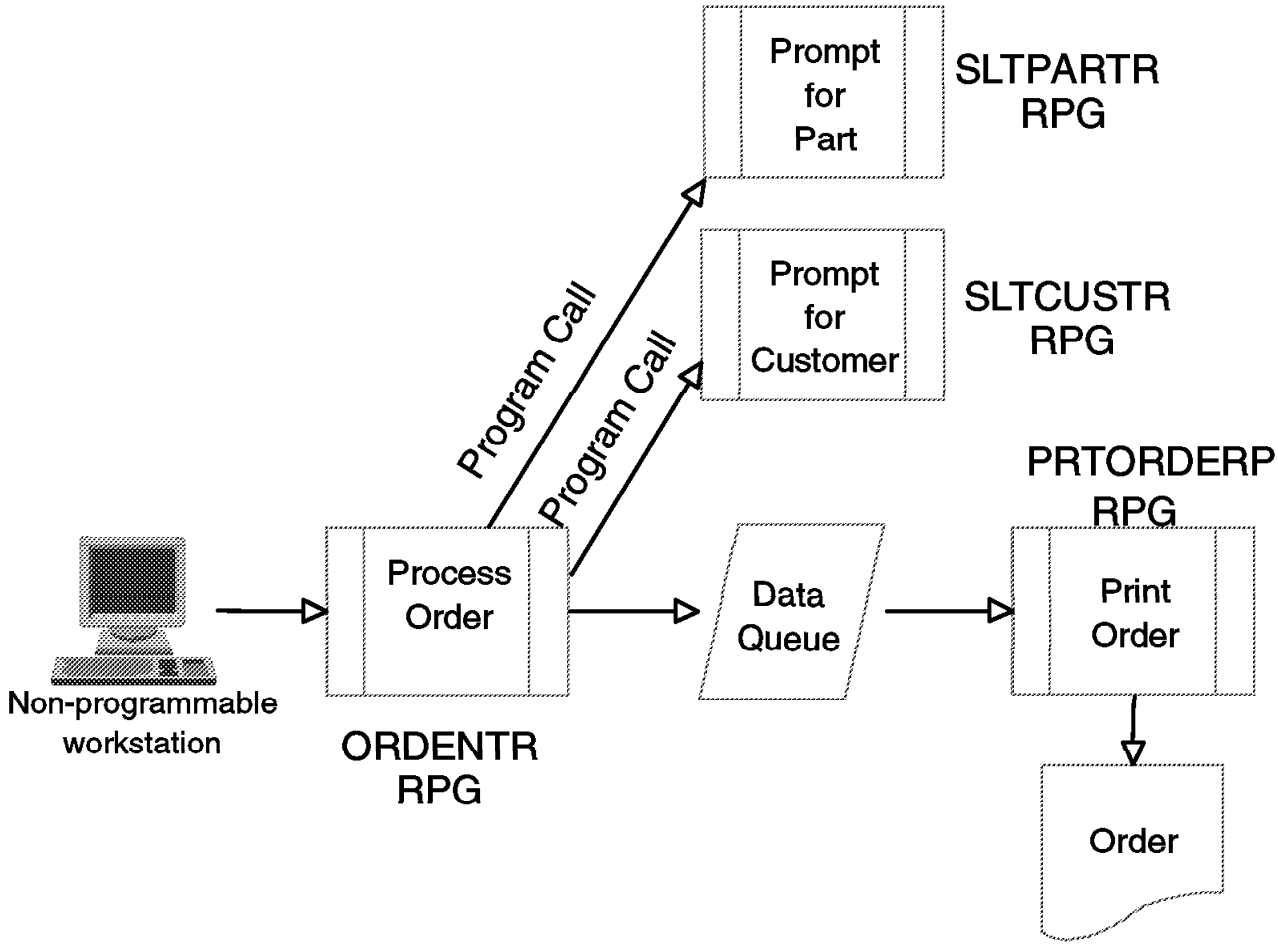


Figure 1. RPG Order Entry Application

In the second version of the application, we develop a Java client graphical user interface. 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. This scenario is described in detail in Chapter 6, “Migrating the User Interface to Java Client” on page 109.

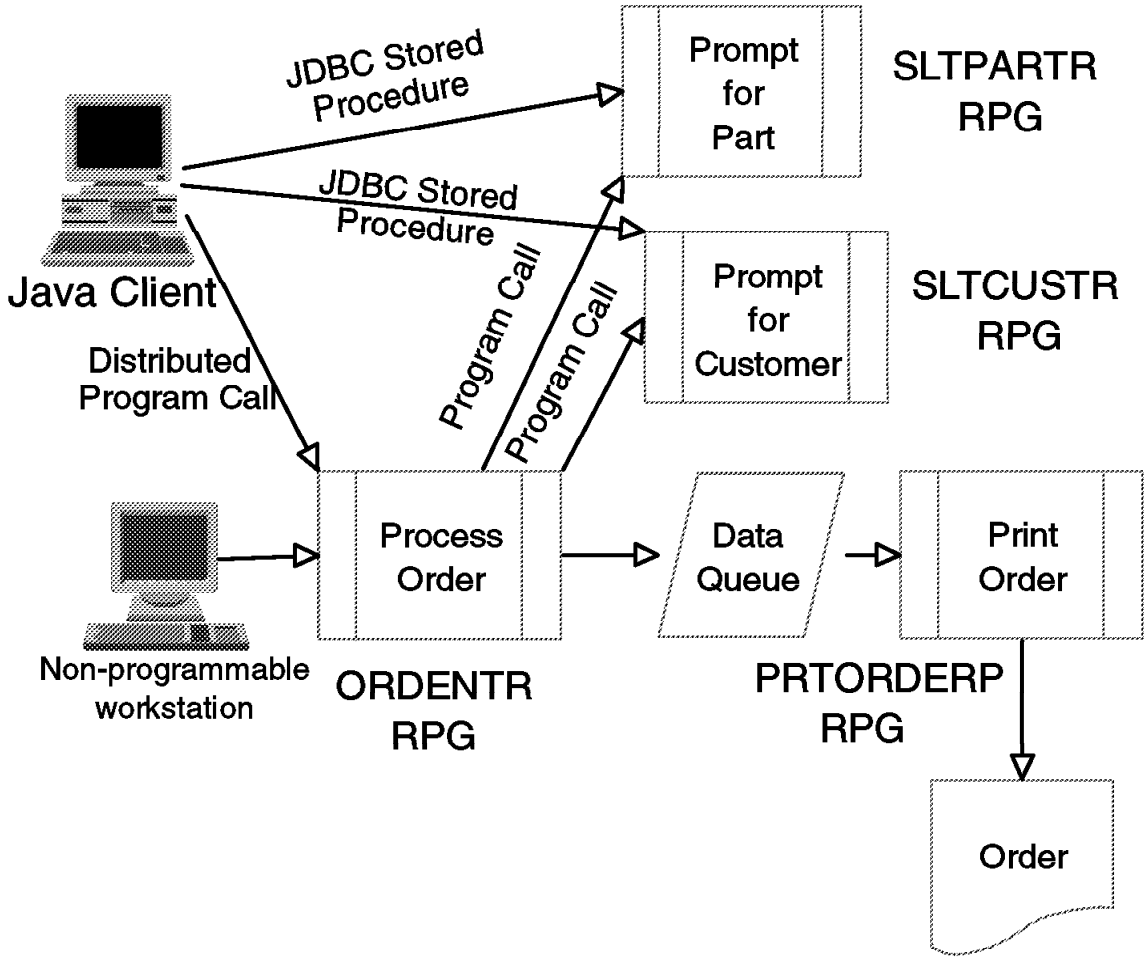


Figure 2. Java Client Order Entry Application

Finally, we change the RPG order entry program to an AS/400 Java program. We modify the client Java program to allow it to interface with the new Java server program through the remote method invocation interface. This scenario is described in detail in Chapter 7, “Moving the Server Application to Java” on page 141.

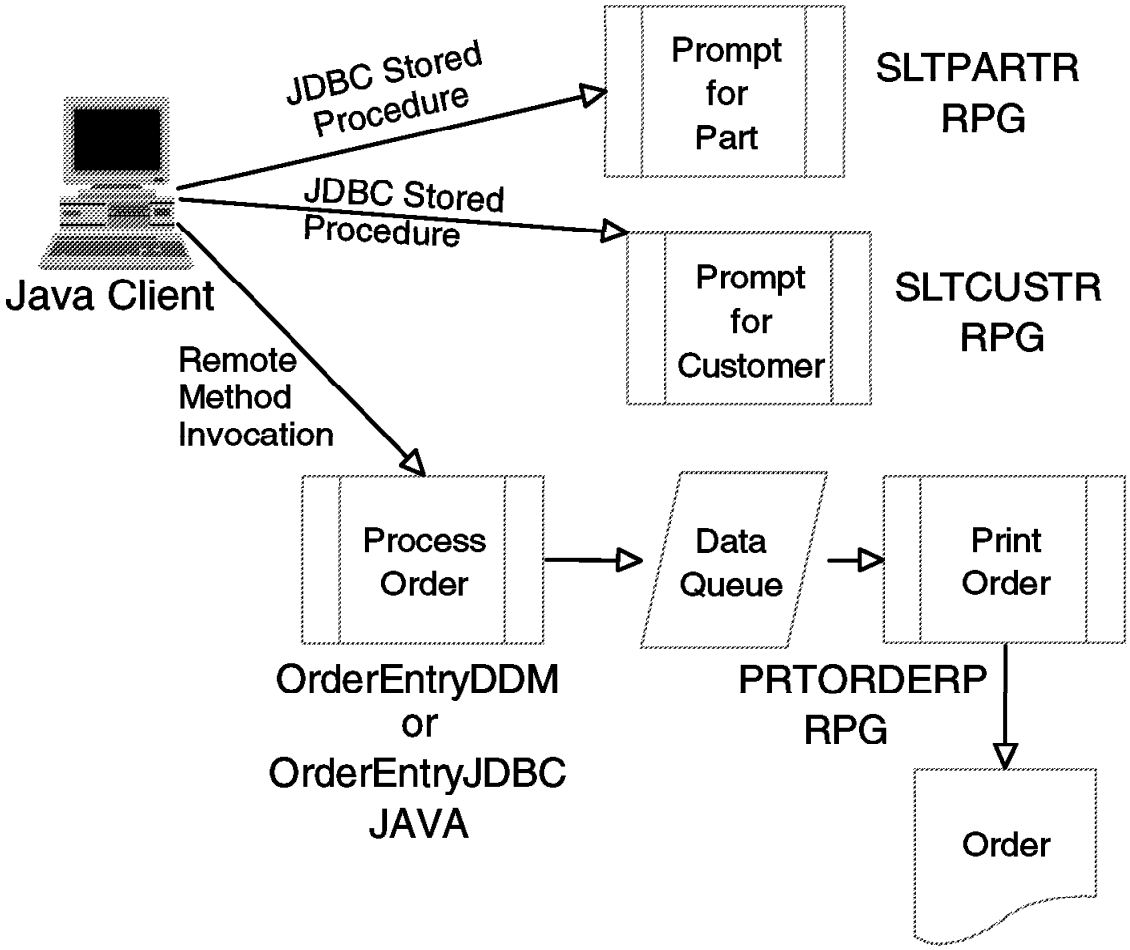


Figure 3. Java Client/Java Server Order Entry Application

To maximize the benefits from this book, certain skills are assumed. First, to install Java on the AS/400 system and on your workstation, we assume that you have general AS/400 operational skills and Windows 95 skills. Also, you need to be proficient with a Web browser. Second, if you plan on developing Java based applications, you need to be proficient in Java programming, object-oriented design techniques, and AS/400 application development methods as well as DB2/400 skills.

This book is not designed to teach basic Java skills. There are many books freely available and translated into many languages that teach Java. See Appendix E, “Related Publications” on page 215 for details.

Chapter 2. Java Overview and AS/400 Implementation

In this chapter, we discuss the Java Platform architecture. First we look at the Java Platform as it is defined by Sun Microsystems' JavaSoft Division and implemented in their Java Development Kit (JDK). Then we present the Java implementation on the AS/400 system and some of the AS/400 specific aspects of this implementation.

2.1 Java Platform

Java is a full fledged Object-Oriented (O-O) 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, easy to learn, simple to debug, and with a reduced maintenance cost, the main advantage of Java is its cross-platform portability. Java's portability is achieved due to the core Java APIs or Java classes that provide a rich set of platform neutral Application Programming Interfaces (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/server Internet enabled applications that can be deployed and run on any Java enabled platform. Hence, Java is not only a new promising programming language, it is a new software platform that can be implemented and run on any of the existing hardware/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 is the center piece of the Java platform; it is the "engine" of Java. As the JVM is responsible for running Java applications or applets in a given hardware/software environment, it is platform dependent. Every company that wants to implement Java on a given platform must implement the Java Virtual Machine on its hardware/software platform. The Java Virtual Machine usually includes the following components.

2.1.1.1 The Class Loader

The class loader is capable of dynamically locating and loading the various classes that the application uses. This is a powerful feature as it allows programmers to develop applications or applets that are made up 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 Java Beans that greatly simplify the job of application developers. The dynamic nature of the class loader greatly simplifies the application packaging process as it is no longer required to go through the complex and error prone process of building the executable program. Each class can be compiled

separately from the other classes and can be loaded as required by the class loader. However, for performance reasons, the developers tend to package related classes together in what is known as **jar** files (Java Archive). 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 required when using more conventional languages such as C or C++. Often all the classes that make up an application are packaged in a single **jar** file. For instance, all 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

A main focus of Java is security. This is important in an open network environment where one may run an applet from an unknown source. Before running a Java program, this component of the JVM performs extensive checks to ensure that the Java bytecodes have not been altered and that they still conform to the Java security specifications. This involves such things as type matching. For instance, 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 run-time exception and the program is terminated.

2.1.1.3 Bytecode Interpreter

This component of the JVM 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. This part of the Java Virtual Machine is essential as it is key to the overall performance of the implementation. As new versions of the JDK are introduced, the overall performance of the bytecode interpreter is improved, thanks to new algorithms being developed.

2.1.1.4 Garbage Collector

Another feature of the Java Virtual Machine is the automatic garbage collection. Unlike C++ where the programmer is responsible for allocating memory to store new objects and freeing unused memory when objects are being discarded, Java provides a fully-automated memory allocation and de-allocation known as automatic garbage collection. This solves one of the main problems found in many C++ applications 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, it is the JVM that allocates the necessary memory when a new object is created while a background task running in a separate thread continuously scans the memory and de-allocates space occupied by objects without an active reference in any of the running classes. This is an important part of the Java Virtual Machine, and it is key to both the performance and the reliability of Java programs.

2.1.1.5 Java To Native Method Interface (JNI)

This component can be viewed as some "glue" code that allows a Java program to invoke a method written in another language than Java. Usually, the supported languages are C or C++. This interface allows for inter-operation between Java applications and legacy applications. However, using Native methods means losing portability. By construction, Native methods are written to a specific execution environment and are platform dependent. As soon as a Java application uses a

piece of code 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 corresponding to the bytecodes in such a way that any further reference to a bytecode that was executed once results in the execution of the corresponding real machine instruction that already exists. This technique greatly enhances the overall performance of the Java Virtual Machine. Just-In-Time compilers can also perform code optimization functions to further improve the performances. Functions such as inlining (that is, including 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 Just-In-Time compilers.

Other functions such as serialization or reflection are also found in a Java Virtual Machine.

Reflection in Java refers to the ability of a Java class to reflect upon itself (that is, 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 are supported by a bean. Reflection can be used to query and set the values of fields, to invoke methods, or to create new objects. Java does not allow methods to be passed directly as data values but the reflection technique makes it possible for methods passed by name to be invoked 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 re-create 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. It can also be used by Java Beans 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 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 Application Programming Interfaces (APIs) that mimic a complete modern, yet platform neutral operating system. The current version of the Java Platform, which is based on the Java Development Kit (JDK) version 1.1.4, is made of two sorts of APIs, the Core library and the Standard Extension library.

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.

The Standard Extension library is a set of APIs outside 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 is made of the following packages:

- **java.applet**: This defines the environment in which applets are running. It provides all the controls required for the Browser to manage the applet's execution.
- **java.awt**: The Abstract Windowing Toolkit (AWT) provides the basic constructs required to build and manage a graphical user interface (GUI).
- **java.beans**: Java Beans are reusable software components that conform to the bean specification and are designed to be directly manipulated by visual development tools.
- **java.io**: This provides all the constructs required to perform standard input and output operations on UNIX style stream files.
- **java.lang**: This is the basic Java language package. This includes the definition of all the Java data types as well as the execution behavior such as multi-threading and exception handling.
- **java.math**: This enhances the basic language constructs by providing a BigDecimal data type and a BigInteger data type and associated operations.
- **java.net**: This provides a TCP/IP connectivity environment that includes sockets, URL, HTTP, and datagram management.
- **java.rmi**: (Remote Method Invocation) allows for inter-operation among distributed Java objects.
- **java.security**: This provides the classes 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 the constructs required to handle relational database accesses.
- **java.text**: This includes all the classes necessary to develop National Language enabled applications.
- **java.util**: This 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 the previous 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/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 (implemented in the java.sql package)

- RMI, Remote Method Invocation (implemented in java.rmi)
- IDL, Interface Definition Language, 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 has been decoupled from the JDK 1.1 release and will be available on a slightly delayed schedule.
- JNDI, Java Naming and Directory Interface provides a unified interface to multiple naming and directory services in the enterprise.

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

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

- New high-level graphical user interface (GUI) components
- Pluggable look and feel
- Accessibility support for people with disabilities
- 2D APIs
- Drag-and-drop

The Java Foundation classes will be part of the Core library of the Java platform and will extend the original Abstract Windowing Toolkit (AWT) by adding a comprehensive set of graphical user interface (GUI) class libraries that is portable and compatible with all AWT-based applications.

2.1.2.2 The Standard Extension Library

The Standard Extension library includes all the Java APIs that have been defined by JavaSoft and that are not part of the Java Core library. The following set of APIs are currently defined:

- The 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 required for developers to quickly develop their own Java servers.
- The Java **Servlet APIs** enable the creation of Java Servlets. This API enables 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.*.
- The 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.
- The **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 including Java components, applications, applets, desktops, and HTML pages.
- The 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 are made of the Java 2D APIs, Java 3D APIs, Java Media Framework APIs, Java Sound APIs, Java Speech APIs, and JavaTelephony APIs.

- The **Java Management APIs** provide a rich set of extensible Java objects and methods for building applets that can manage an enterprise network over Internet. It has been developed in collaboration with SunSoft and a broad range of industry leaders including 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.
- The **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.
- The **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 a flavor of the extensive set of functions provided by today's Java platform and some of the directions where Java is going. It demonstrates that Java is not another simple and easy to use programming language but a sophisticated programming environment for developing today's and tomorrow's applications.

2.1.3 Java Utilities

The Java Utilities is a set of programmer's aids that covers the programming side of the application development cycle. The following tools are provided:

2.1.3.1 java

This command allows you to run a Java application. Note that 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 **javac**. 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 follows:

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

2.1.3.2 javac

This tool invokes the Java compiler. The Java compiler checks the syntax of a Java source file (**.java** file). It then compiles it and creates a Java bytecodes file (**.class** file) that can be run using the previous **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 defined in each source file compiled by **javac**, the compiler stores the resulting bytecodes in a class file with a name of the form **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 jdb

This command invokes the Java language debugger that helps 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. The same as dbx, there are two ways **jdb** can be used for debugging. The most frequently used way is to have the debugger start the Java interpreter with the class to be debugged. This is done by typing **jdb** instead of **java** 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 javah

This tools reads a Java **.class** file and creates a C language header file and stub file to be included in a C source file. This utility provides the "glue code" that allows the programmer 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 an object's 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 new native method interface, Java Native Interface (JNI), does not require header information or stub files. The **javah** utility can still be used with the **-jni** option to generate native method function prototypes needed for JNI-style native methods. The result is placed in the **.h** file. Using native methods means that the application is not 100% pure Java and, thus, is not directly portable across platforms. Native methods are, by nature, platform or system specific.

2.1.3.5 javap

This command invokes the Java disassembler. This Java tool disassembles a **.class** file. Its output depends on the options used. If no options are used, javap prints out the public fields and methods of the classes passed to it. **javap** 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. The use of this tool may violate the license agreement for the class you are disassembling.

2.1.3.6 javadoc

This command produces a standard documentation for your Java classes. The documentation is in HTML format and can be viewed by any Browser. The **javadoc** utility 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** tool to produce additional information, you must use the special form of comments in your Java source file. The javadoc type comments start with **/**** and ends with ***/**. All the text included between the opening **/**** tag and the ending ***/** tag is added to the generated documentation.

2.1.3.7 jar

The **jar** tool combines several Java **.class** files into a single Java ARchive **(.jar** file). Java ARchive files are used to minimize Java applets download time and to simplify the Java application's installation on distributed clients and servers. The **jar** utility was designed mainly to facilitate 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 times. The **jar** utility also compresses files using a **zip** like algorithm and so further improves download time.

2.1.3.8 javakey

This tool adds a digital signature to a jar file. This allows you to improve the security of your Java applets as the users of your applets know that they are using authenticated Java code originating 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/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 associated with it one or more certificates authenticating its public key. A certificate is a digitally signed statement from one entity, saying 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 authenticating its public key.

2.1.3.9 appletviewer

This tool allows you to run a Java applet without using a browser. The tool fully supports the 1.1.4 level of the JDK whereas most browsers do not support it yet. If the HTML page you are viewing makes reference to several applets, each applet is displayed in a separate window.

2.1.3.10 rmic

This command generates a stub class file and a skeleton class file for Java objects that implement the Remote Method Invocation 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's 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's 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 rmiregistry

This 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. Clients can then look up remote objects and make remote method invocations. The registry is typically used to locate the first remote object on which an application needs to invoke methods. That object, in turn, provides application-specific support for finding other objects.

2.1.3.12 serialver

This command returns the serial version ID for one or more classes.

2.1.3.13 native2ascii

This utility 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 containing Latin-1 and Unicode-encoded charaters.

For more details on this set of utilities and for the exact command syntax, please refer to the 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 AS/400 Java Implementation

Starting with V4R2 of OS/400, the AS/400 system implements the Java Platform. This implementation fully complies with the Java Development Kit (JDK) 1.1.4 as defined by JavaSoft. As in any other Java platform implementation, the AS/400 system provides the following components:

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

The AS/400 Java Implementation has the following enhancements:

- The JVM is integrated into the Systems Licensed Internal Code (SLIC).
- Static compilation of class files
- Dynamic class loading
- Remote Abstract Windowing Toolkit (AWT)
- Scaleable garbage collector

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

Figure 4 shows a high level diagram of the AS/400 Java implementation.

2.2.1 AS/400 Java Virtual Machine

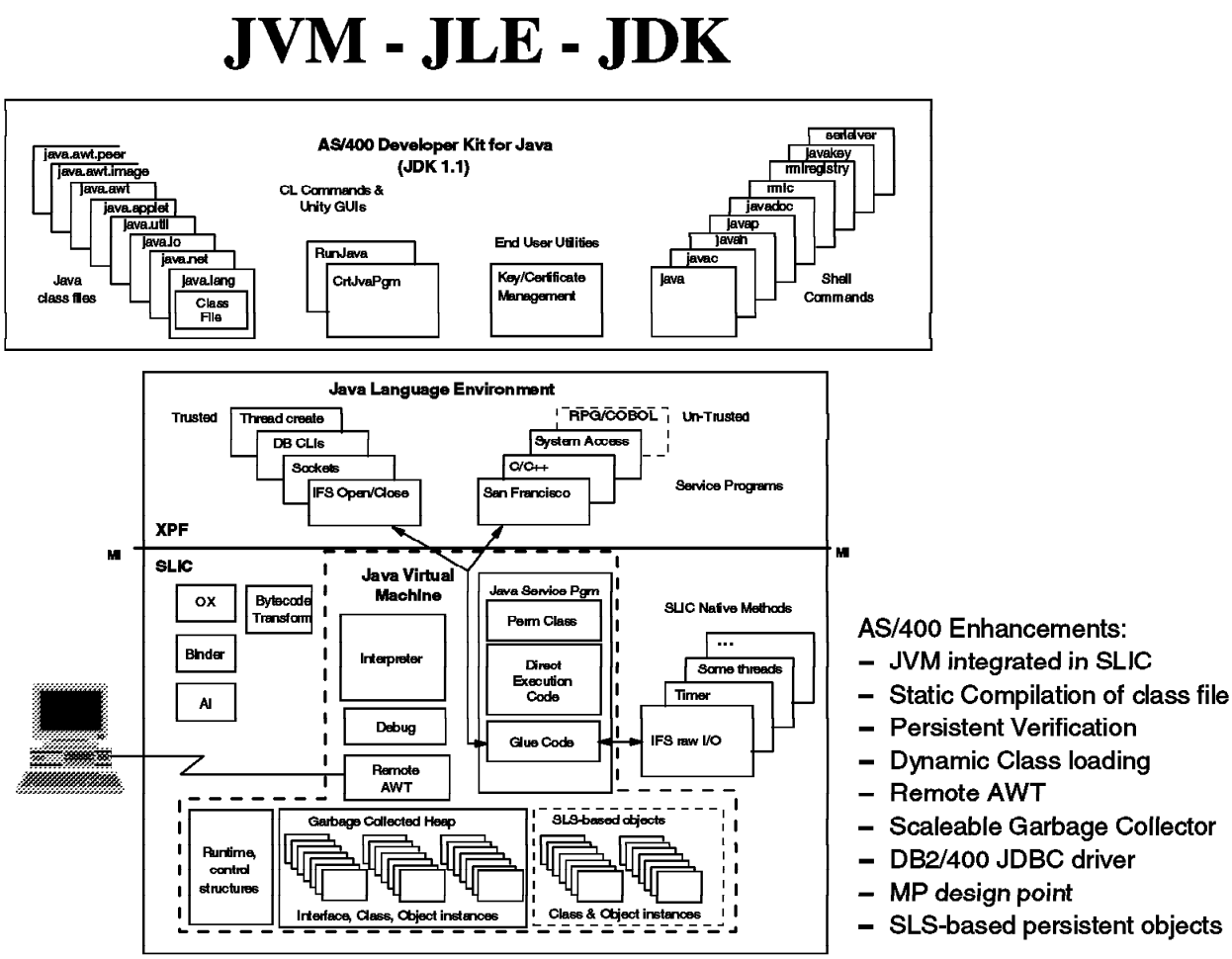


Figure 4. Schematic of AS/400 Java Virtual Machine and AS/400 Developer Kit for Java

On the AS/400 system, the Java Virtual Machine is implemented within the System Licensed Internal Code (SLIC) below the Technology Independent Machine Interface (TIMI), and it is an integral part of OS/400. This means that as soon as you install OS/400 V4R2 on your machine, you have installed a standard Java Virtual Machine on your system. The OS/400 Java Virtual Machine includes all the components of a standard Java Virtual Machine described earlier in this chapter:

- The class loader
- The bytecode verifier
- The bytecode interpreter
- The garbage collector

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

Important Note

Although there is no way to prevent a Java program from invoking a native ILE C method, which, in turn, calls an RPG or COBOL program, this is not supported in V4R2 as 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. Future releases of the RPG and COBOL compilers may eventually be able to produce a thread enabled RPG or COBOL program.

As of V4R2, not all OS/400 system functions are thread safe. Thus, 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 V4R2.

2.2.2 AS/400 Java APIs

The Java Core Library APIs as defined in JavaSoft's Java Development Kit (JDK) 1.1.4 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. This licensed program is called AS/400 Developer Kit for Java (5769-JV1) and can be installed using the standard OS/400 Install Licensed Program procedures. Chapter 3, “Installation” on page 31 contains all the details on how to install this licensed program on your system.

This is a "skip release" licensed program product that follows Sun's 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 (IFS) support of OS/400 and provide access to any UNIX or PC style stream file within the IFS. The **java.net** APIs use the standard TCP/IP support provided by the TCP/IP Connectivity Utilities for AS/400 (5769-TC1) licensed program 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 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 Java Virtual Machine to route any GUI operation to a workstation. This is because there are no graphical user interface capable devices on the AS/400 system. Please see Section 2.3, “AS/400 Specific Implementation” on page 19 for more details on Remote AWT.

2.2.3 AS/400 Java Utilities

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

The QShell Interpreter is a new option of OS/400 that you must install on your AS/400 system to run any Java Utility on your system. Please refer to Chapter 3, “Installation” on page 31 for more details on how to install this option on your AS/400 system. See also Section 2.3, “AS/400 Specific Implementation” on page 19 for more details on the QShell Interpreter.

As of V4R2, **qsh** supports the following Java utilities:

- java
- javac
- javah
- javap
- javadoc
- rmic
- rmiregistry
- serialver
- jar
- javakey

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

2.2.3.1 java

The Java utility runs Java programs associated with the specified Java class. The AS/400 Developer Kit for Java version of this utility **does not** support the following 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 OS/400 command (RUNJVA or JAVA) to debug a Java program. The VisualAge for Java AS/400 feature provides a cooperative debugger capable of debugging AS/400 Server Java applications from a workstation. This debugger 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 in lieu of the standard JDK garbage collector. The AS/400 garbage collector can be tuned using the new special options.
- **noclassgc**: As previously mentioned, the AS/400 garbage collector uses its own options. The standard **java** utility options do not apply.
- **-help**: To get help on how to use the **java** utility, use the OS/400 JAVA (or RUNJVA) command from any OS/400 command entry line and use the standard OS/400 prompt (F4) and help (F1) support.
- **-prof**: Output profiling information to a specified file or to the file java.prof 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 run-time 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 run-time 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 CRTJVAPGM or the RUNJVA (JAVA) commands, tracing may not be available.

- **-verify**: The Java bytecodes are verified, then translated into AS/400 Risc PowerPC machine instructions when you run the CRTJVAPGM or the RUNJVA (JAVA) OS/400 commands. All the standard **java** utility verify options do not apply to the AS/400 Java run-time environment.
- **-verifyremote**: See the preceding **-verify** option.
- **-noverify**: See the preceding **-verify** option.
- **-verbosegc**: The AS/400 garbage collector does not support this option as it can result in a huge number of messages being put out given the large amount of objects that can potentially be handled in the AS/400 system.

The AS/400 version of the Java tool supports specific AS/400 options as follows:

- **-secure**: This option tells the AS/400 Java Virtual Machine to check for public write access to the directories specified in the CLASSPATH. A directory in the CLASSPATH that has public write authority is a security exposure because somebody may store a class file with the same name as the one 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.

We expect that AS/400 programmers will prefer to use the OS/400 equivalent command (RUNJVA or JAVA) to benefit from the standard OS/400 environment such as prompting and online help they are used to. See Section 2.3.1, “The OS/400 Java Commands” on page 19 for more details on OS/400 Java related commands.

2.2.3.2 javah

The AS/400 implementation supports only the JNI type of native method invocations.

- **-jni**: This option causes **javah** 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 these **-jni** type files are always produced.
- **-td.**: If this option is specified, 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 this option is specified, it is ignored by the AS/400 implementation.
- **-trace**: If this option is specified, it is ignored by the AS/400 implementation.
- **-v**: Verbose. This option is not supported.

2.2.3.3 javap

The AS/400 implementation ignores the following 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.
- **-verify**: This option of **javap** will be removed in the next release of the JDK. Thus, it did not make sense to support it on the AS/400 system.

2.2.3.4 Java Utilities Not Supported by AS/400 System

The V4R2 implementation of the JDK does not support the following Java utilities:

- **jdb**: To debug a Java program on the AS/400 system, you must use the RUNJVA (or JAVA) command from any OS/400 command entry display. Specify OPTION(*DEBUG) to start the OS/400 system debugger. Make sure that you used the **-g** option when you created the Java bytecodes with the **javac** utility. Do not use the optimization option **-o** when you compile your Java source program into bytecodes. This causes the Java compiler to place static, final and private methods inline; thus, the bytecodes are not the exact representation of the source code. Also make sure that both the Java source code (**.java** file) and the bytecodes (**.class** file) are stored in the same directory in the AS/400 Integrated File System. This ensures that the AS/400 system debugger finds all the source files associated with every class that is used. The debugger assumes that the source file is contained in a **.java** file with the same name as the class it is debugging.
- **appletviewer**: The AS/400 system cannot run applets as it does not have any graphical user interface 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. First, we look at the OS/400 commands that are provided to help you perform some Java related functions in a standard AS/400 way where command prompting and online help text is available. Then we teach you how to start the QShell Interpreter environment and how to use the qsh window. Finally, we explain the Remote AWT support.

2.3.1 The OS/400 Java Commands

With V4R2 of OS/400, you can use OS/400 commands to perform certain Java related functions. Some commands such as RUNJVA (or JAVA) are OS/400 equivalents to existing Java utilities such as **java**. The other commands (CRTJVAPGM, DSPJVAPGM, and DTLJVAPGM) are specific to the AS/400 implementation.

2.3.1.1 CRTJVAPGM

The AS/400 implementation of Java provides a unique component called the bytecodes transformer. This system function preprocesses Java bytecodes produced by any Java compiler on any platform and contained in a **.class** file, a **.jar** file, or a **.zip** file to prepare them to run using the OS/400 Java Virtual machine. The Java Transformer creates an optimized Java program object that is persistent and is associated with the **.class** file, the **.jar** file, or the **.zip** file. This program object contains Risc PowerPC 64-bit machine instructions. The optimized program object is not interpreted by the bytecodes interpreter at run time, but directly executes when the class is loaded.

No action is required to start the bytecodes 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 OS/400 command.

It is especially important to use CRTJVAPGM on .jar and .zip files. Unless the entire **.jar** or **.zip** file has been optimized using the CRTJVAPGM, each individual class is optimized at run time and the resulting program objects are temporary.

Using the CRTJVAPGM command on a **.jar** or a **.zip** file causes all the classes contained in that file to be optimized and the resulting optimized Java program object to be persistent. This results in much better run-time performance.

The Create Java Program (CRTJVAPGM) command creates an AS/400 Java program from a Java **.class**, **.jar**, or a **.zip** file. The resulting Java program object becomes part of the class, jar, or zip file object, but cannot be modified directly. When invoked by the RUNJVA or Run Java Program (JAVA) command, the Java program is run. The size and performance of the Java program can be controlled through the use of the OPTIMIZE parameter.

On this command, you specify the name of the Java **.class** file containing the bytecodes of the Java program you want to create. You may also specify the name of a **.jar** or **.zip** file that contains several classes packaged together. 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 invoked. For other optimization levels, the Java program contains

machine instruction sequences that are run when the Java program is invoked. The possible values for the *OPTIMIZE* parameter are as follows:

- ***INTERPRET**: The Java program created is not optimized. When invoked, the Java program interprets the class file byte codes. Variables can be displayed and modified while debugging.
- **10**: The Java program contains a compiled version of the class file bytecodes but has only minimal additional compiler optimization. Variables can be displayed and modified while debugging. This is the default value for **OPTIMIZE**.
- **20**: The Java program contains a compiled version of the class file bytecodes and has some additional compiler optimization. Variables can be displayed but not modified while debugging.
- **30**: The Java program contains a compiled version of the class file bytecodes and has more compiler optimization than optimization level 20. During a debug session, user variables cannot be changed, but can be displayed. The given values may not be the current values of the variables.
- **40**: The Java program contains a compiled version of the class file bytecodes and has more compiler optimization than optimization level 30. All call and instruction tracing is disabled.

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

The possible values for this parameter are:

- ***NONE**: The collection of performance data is not enabled. No performance data is to be collected.
- ***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 REPLACE parameter allows you to specify whether an existing AS/400 Java program should be replaced or not.

Figure 5 on page 21 shows the OS/400 prompt for the CRTJVAPGM command.

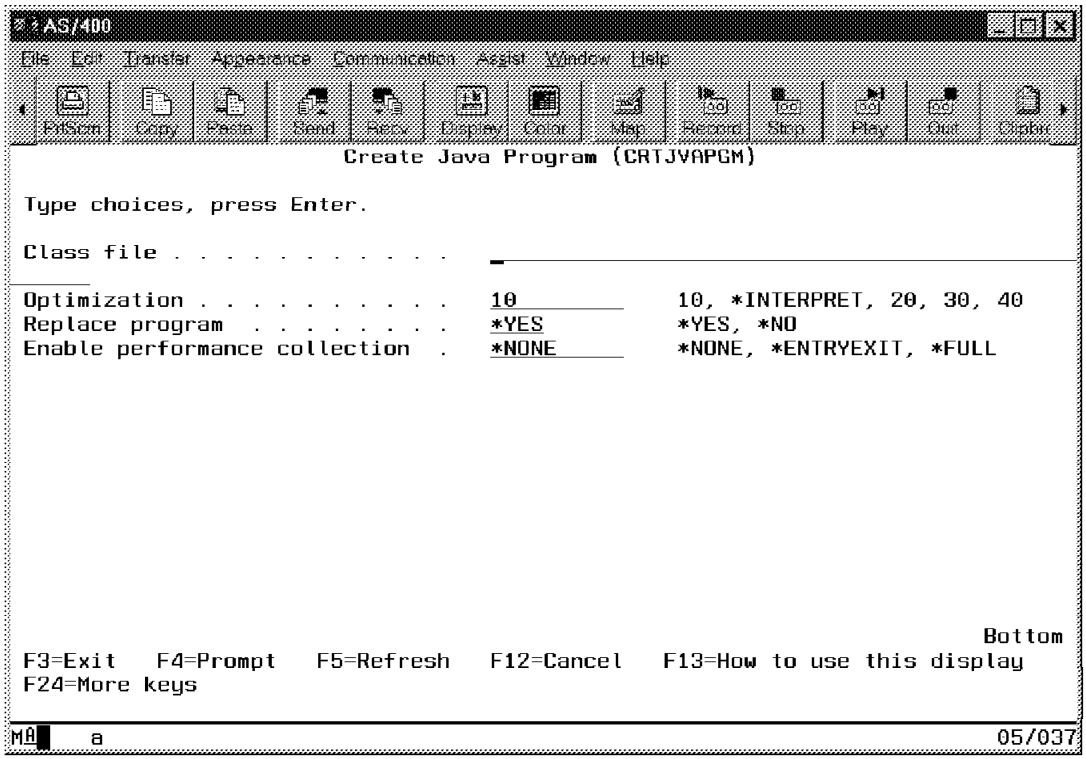


Figure 5. Create Java Program (CRTJVAPGM) Command

2.3.1.2 DLTJVAPGM

The Delete Java Program (DLTJVAPGM) command deletes an AS/400 Java program associated with a Java **.class**, **.jar** or **.zip** file. If no Java program is associated with the class file specified, an informational message is sent and command processing continues.

2.3.1.3 DSPJVAPGM

The Display Java Program (DSPJVAPGM) command displays information about the AS/400 Java program associated with a Java class file. If no Java program is associated with the class file specified, an error message is sent and the command is cancelled. The OUTPUT parameter allows you to specify where the output should be directed to. Specify ***** to display the results and ***PRINT** to send the results to a spooled file. You can specify the name of a **.class**, **.jar**, or **.zip** file.

Figure 6 on page 22 shows the output of the Display Java Program command.

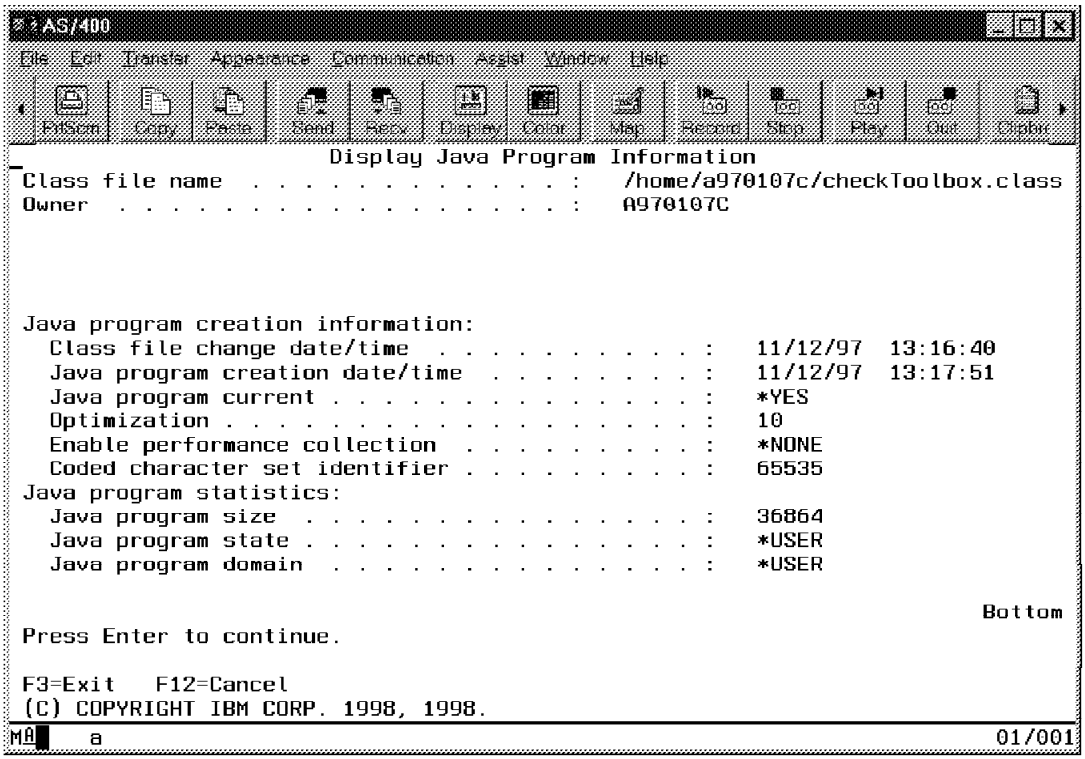


Figure 6. Display Java Program (DSPJVAPGM) Command

2.3.1.4 RUNJVA (or JAVA)

The Run Java Program (JAVA) command runs the AS/400 Java program associated with the specified Java class. If there is no *JVAPGM object associated with the class file, one is created and associated permanently with the .class file. It is used for executing the class file rather than interpreting the bytecodes.

If you specify the special value *VERSION instead of a valid Java class name, the build version information for the Java Development Kit (JDK) and the Java Virtual Machine (JVM) is displayed. No Java program is run.

The following parameters can be specified on the RUNJVA command:

- **PARM**: Specifies one or more parameter values that are passed to the Java program. A maximum of 200 parameter values can be passed.
- **CLASSPATH**: Specifies the path used to locate classes. Directories are separated by colons. If the special value *ENVVAR is used, the class path is determined by the environment variable **CLASSPATH**. The **CLASSPATH** environment variable can be set by the Add Environment Variable (ADDENVVAR) command, be part of an **export** directive in the system wide **/etc/profile** file, or specified at the user profile level with an **expor**t directive contained in the **.profile** file in the home directory of each user. See Chapter 3, “Installation” on page 31 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. Whichever class file is found first 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.
  - *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 - ignore 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 on the CRTJVAPGM command. You can disable optimization by specifying OPTIMIZE(*INTERPRET) on the RUNJVA command. This requires that the classes be interpreted regardless of the optimization level 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) that the garbage collection heap can grow to. This is used to prevent 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. This may adversely affect performance.
- **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.

- **OPTION**: Specifies special options used when running the Java class. The possible values are *NONE (no special options are used when running the Java class), *VERBOSE (a message is displayed each time a class file is loaded), or *DEBUG (allows the AS/400 system debugger to be used for this Java program).

Figure 7 and Figure 8 on page 25 show the RUNJVA command prompts.

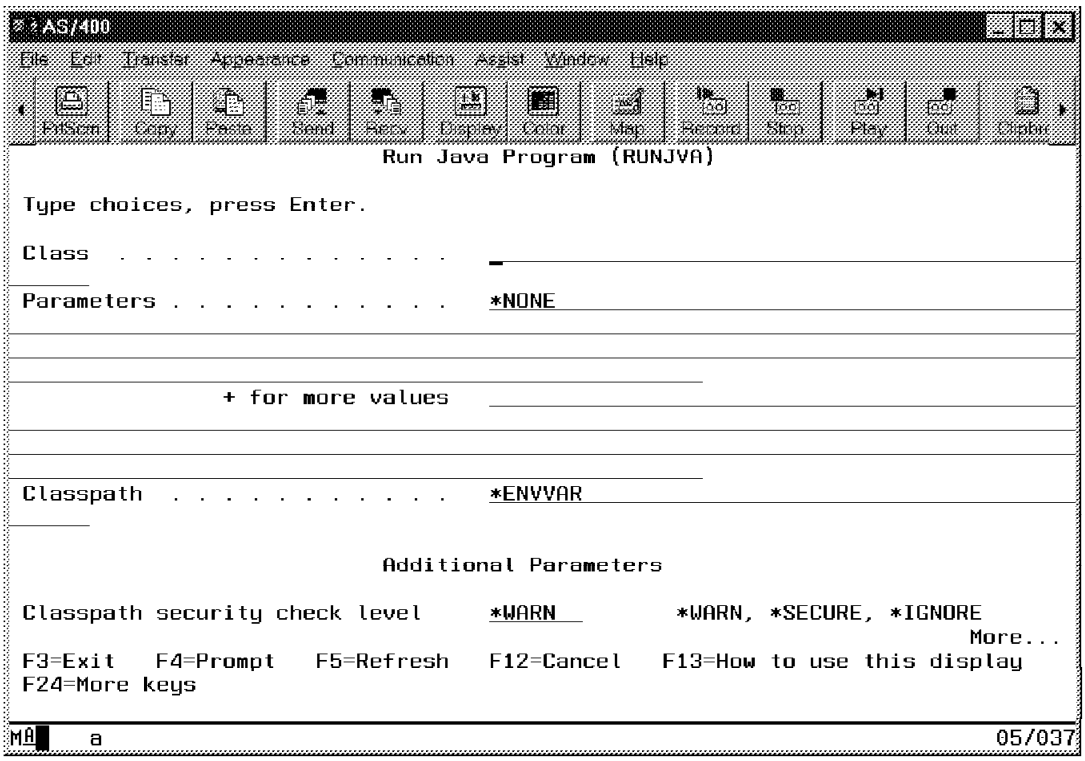


Figure 7. Run Java Program (RUNJVA) Command (Display 1 of 2)

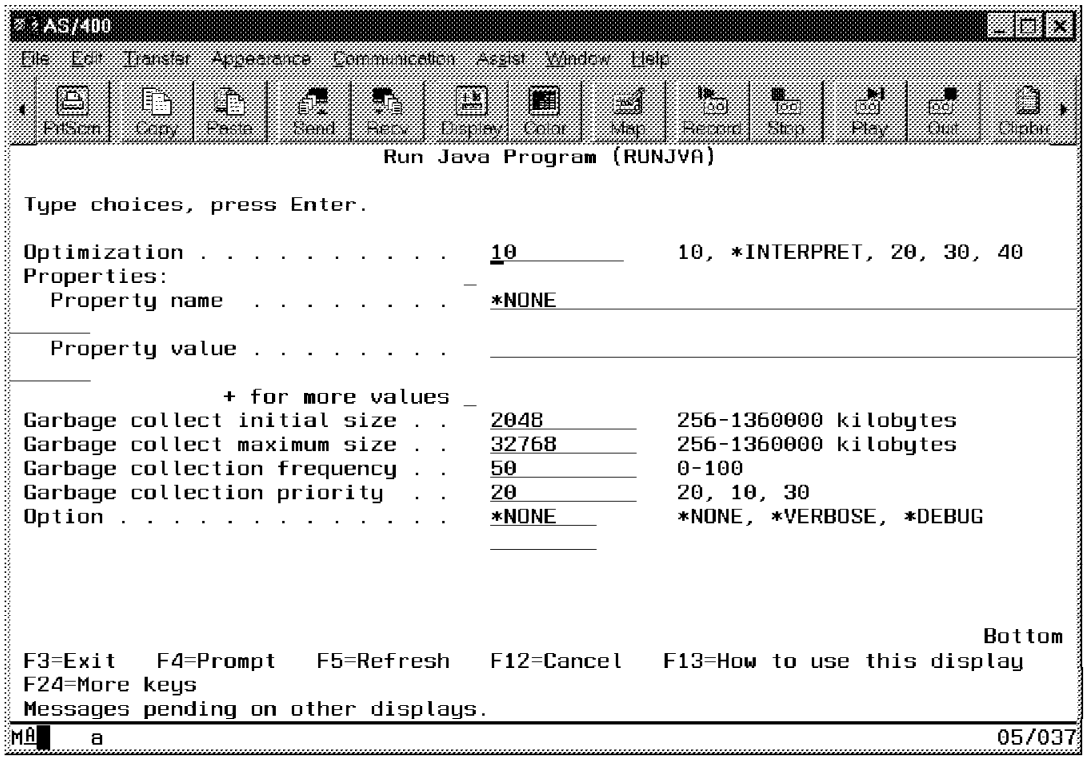


Figure 8. Run Java Program (RUNJVA) Command (Display 2 of 2)

2.3.2 The QShell Interpreter

The QShell Interpreter, **qsh**, is a command interpreter for the AS/400 system that is based on POSIX and X/Open standards. The QShell Interpreter is based on the POSIX 1003.2 standard and X/Open CAE Specification for Shell and Utilities. It has many features that make it the same as the Korn shell (**ksh**) and it is upwardly compatible with the Bourne shell (**sh**).

With **qsh**, you can:

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

qsh provides the following features:

- Quoting, including the escape character, literal quotes, and grouping quotes
- Parameters and variables including positional parameters and special parameters
- Word expansion including tilde ( ) expansion, parameter expansion, command substitution, arithmetic expansion, field splitting, path name expansion, and quote removal
- Input/output redirection
- Commands, including 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.

To start the QShell Interpreter, use the STRQSH or the QSH OS/400 commands. You can specify a QShell command to be 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.

Figure 9 shows the STRQSH command prompt.

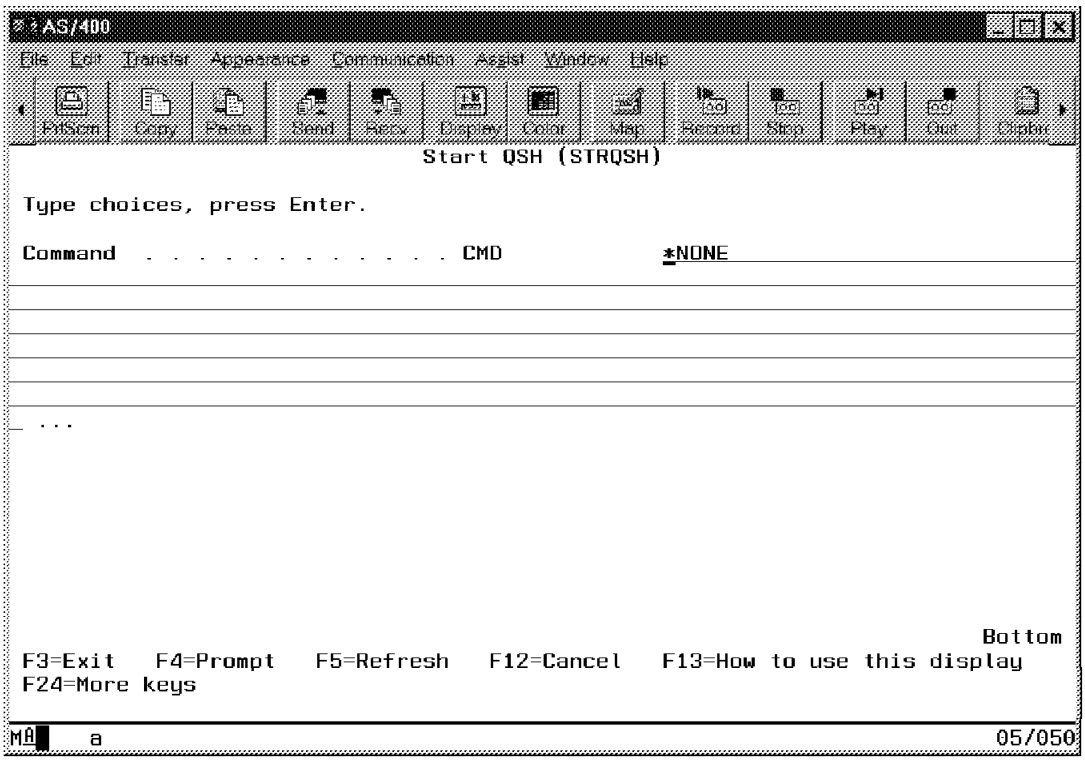


Figure 9. Starting the Qshell Interpreter

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 does the following actions:

- Start a new shell session for your job.
- Display the shell terminal window on your display.
- Run the commands contained in the **profile** file of the **/etc** directory (**/etc/profile**) if such a file exists in the /etc directory.
- Run the commands 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 you run. The terminal window has two parts, 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 entered on the input line and any output generated by the commands you entered. The output area can be scrolled backward and forward.

Figure 10 shows the shell terminal window.

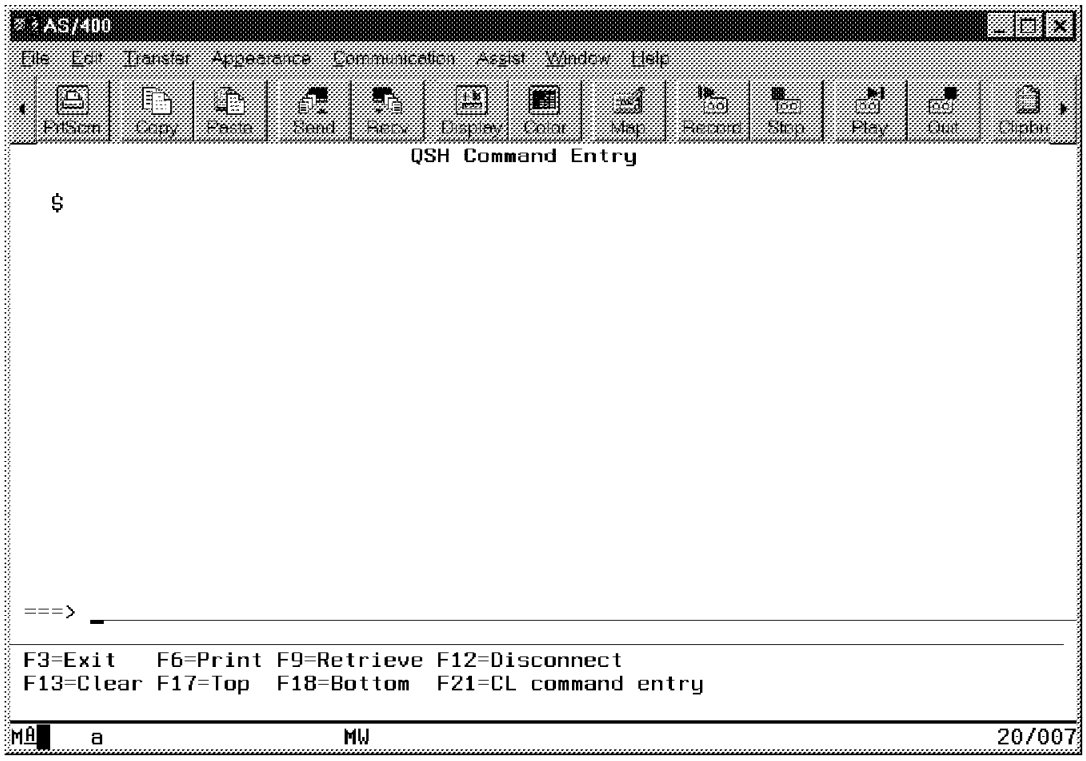


Figure 10. QSH Command Entry

The following 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: Retrieve a previous command. You can press this key multiple times to retrieve any previous command. You can also select a specific command to be 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/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 The Remote AWT Support

The Remote Abstract Windowing Toolkit (Remote AWT) is an implementation of the Java Abstract Windowing Toolkit (AWT) that allows Java applications that use the Java AWT support to run unchanged on a host that does not have a graphical user interface (GUI) such as the AS/400 system.

The Remote AWT support is a set of Java classes that use the remote method invocation (RMI) feature of the JDK. As such, this support can also be used 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 graphical user interface (GUI). Every call to any AWT API is passed to the Remote AWT support on the "source" system. The Remote AWT function uses RMI to communicate with its equivalent function on the "target" (remote) system. On the "target" (remote) system, the Remote AWT support passes all the AWT requests 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 Remote AWT support. The remote AWT support sends all the requests to its peer component on the "source" system using RMI. 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 11 on page 29 shows a Remote AWT implementation.

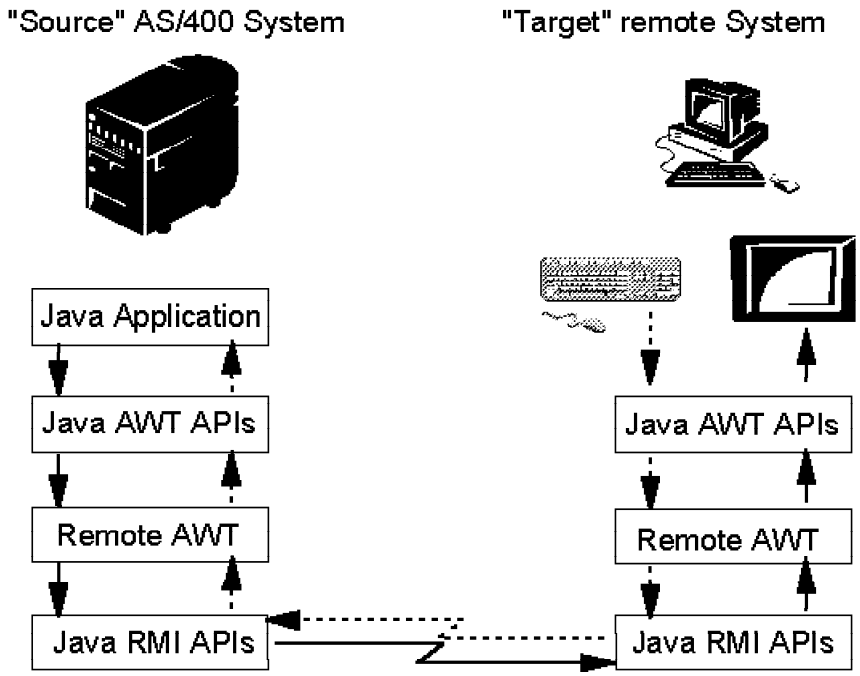


Figure 11. Source and Target Systems

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 graphical user interface 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 interactions and is performed on the server.

Remote AWT is not intended to be used as a way to support client/server applications involving heavy graphical user interface (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 12 on page 30 shows the architecture of a standard client/server application.

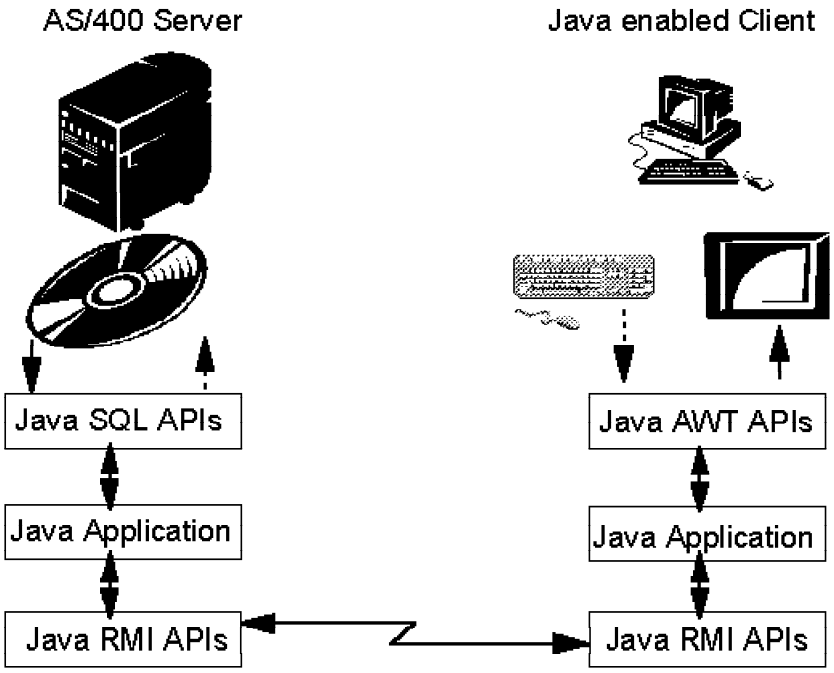


Figure 12. AS/400 Server and Java-Enabled Client

As you can see, 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 graphical user interface. 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. Care should be taken when placing enterprise data on client workstations.

Chapter 3. Installation

This chapter provides the information needed to successfully install various software components required to run Java on the AS/400 system. We cover:

1. Installing the Java support on the AS/400 system itself.
2. The necessary steps required to install the JavaSoft Java Development Kit (JDK) on a workstation.
3. Installing the AS/400 Toolbox for Java on the workstation.
4. How to set up the run-time environment variables such as the **PATH** and **CLASSPATH** directives.
5. How to configure and run Remote AWT.

In this book, we use a PC based workstation running Microsoft's Windows 95 Operating System as an example. Any Java Virtual Machine (JVM) enabled workstation can be used in lieu of a Windows 95 system. This includes RS/6000 with AIX workstations, PCs running IBM's OS/2 Warp Version 4 or Microsoft's Windows NT, Apple's Macintosh, or any JVM capable Network Computer such as IBM's Network Station.

3.1 Installing Java on AS/400 System

To install Java on the AS/400 system, sign on to the system using the QSECOFR user profile. At the **Sign On** display, enter **QSECOFR** on the User prompt and type the corresponding password on the Password prompt shown in Figure 13 on page 32.

Pressing the Enter key displays the **AS/400 Main Menu**.

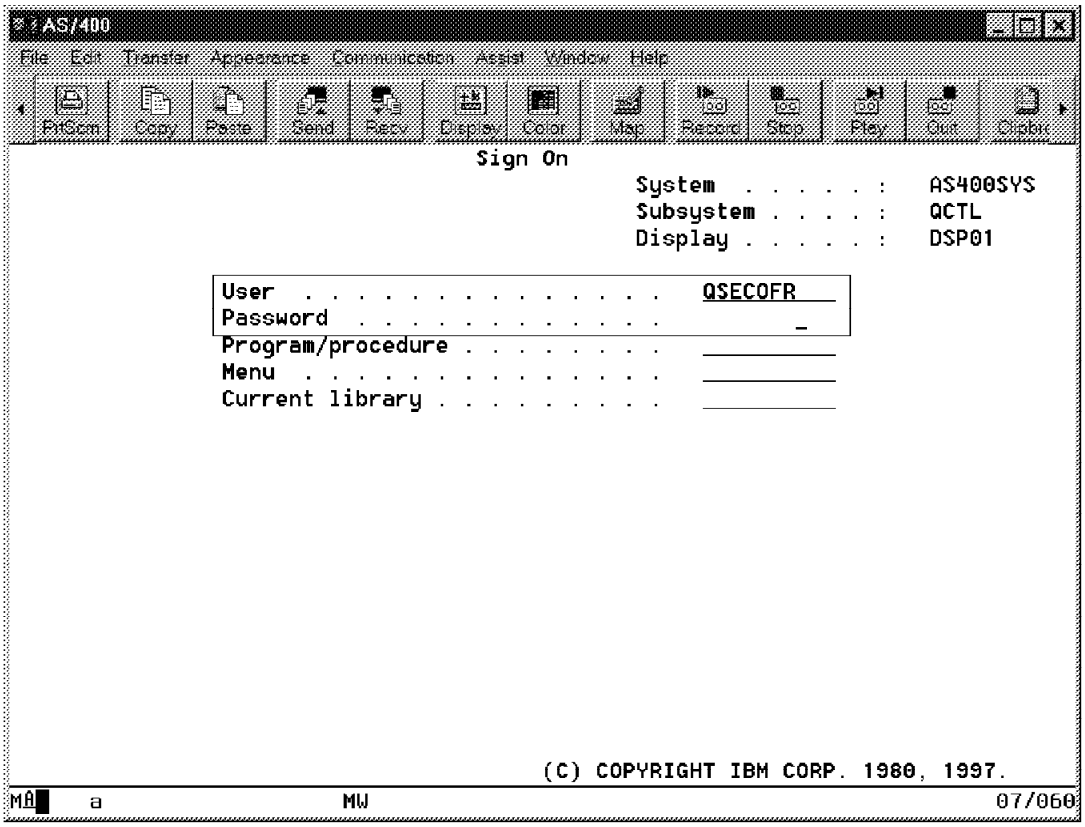


Figure 13. Sign On to Your AS/400 using QSECOFR User Profile

On the AS/400 Main Menu command line, type **GO LICPGM** as shown in Figure 14 on page 33. The system then displays the **Work with Licensed Programs** menu.

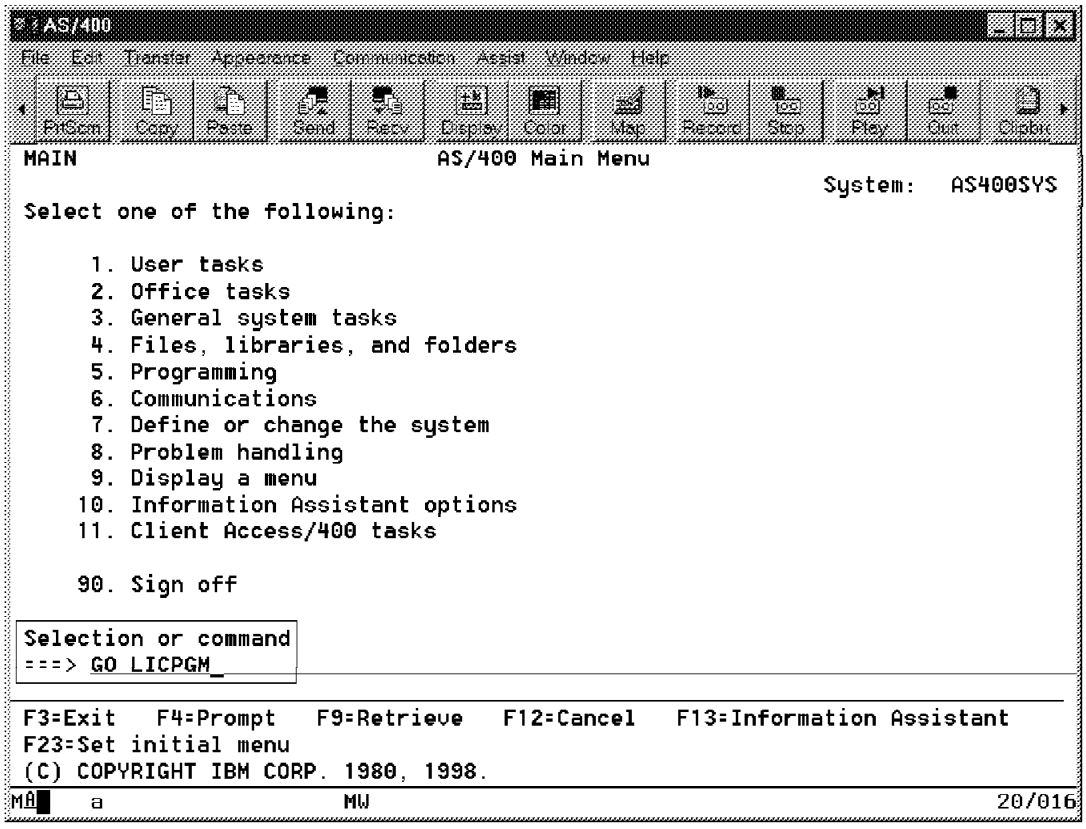


Figure 14. AS/400 Main Menu

Now check to see if all the required software is already installed on your AS/400 system. All AS/400 systems shipped from IBM with OS/400 Version 4 Release 2 (V4R2) installed should have all the required support already pre-loaded. 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 AS/400 System” on page 41.

3.1.1 Checking What Software is Installed

On the **Work with Licensed Programs** menu command line, type option **10** as shown in Figure 15 on page 34. All of the licensed programs currently installed on your system are displayed.

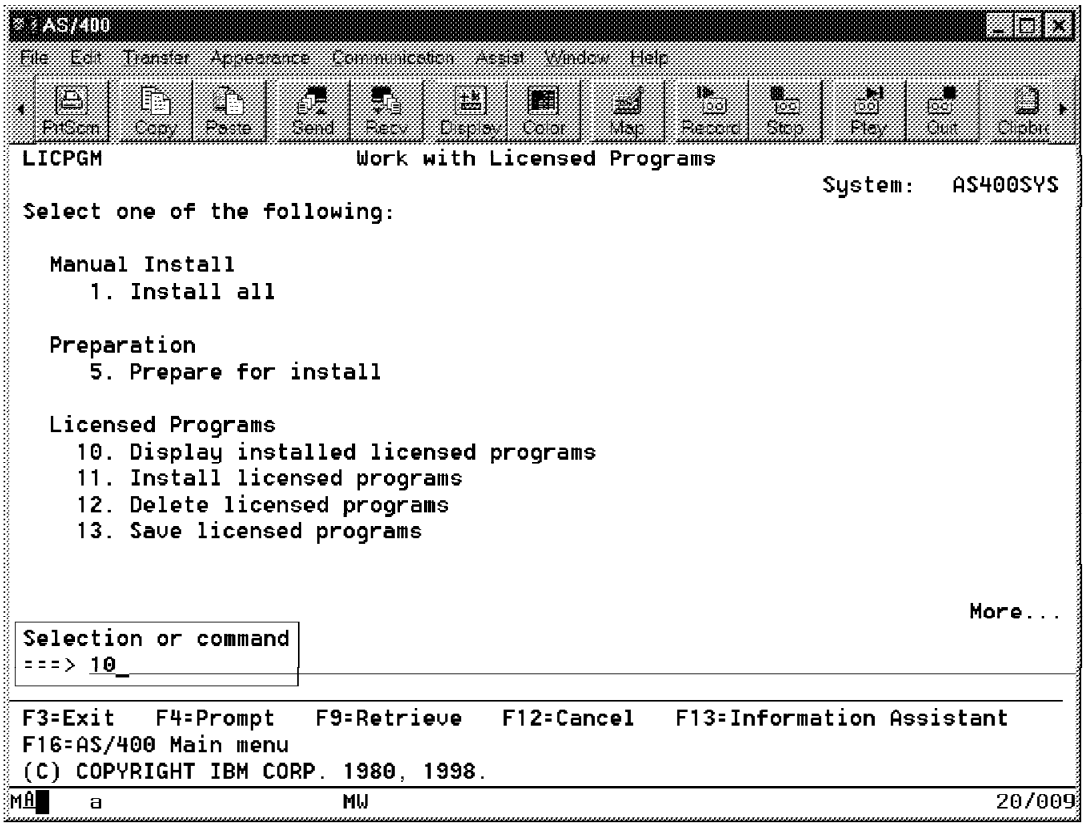


Figure 15. Work with Licensed Program Menu

Pressing the Enter key shows the **Display Installed Licensed Programs** display. On this display, you can choose the type of information shown in the second column of the display. The default setting is to display the compatibility status of the various licensed programs. Pressing **F11** one time displays the Installed Release information. Pressing **F11** again displays the Product Option information. Since we are looking for some OS/400 options, select the Product Option information shown in Figure 16 on page 35.

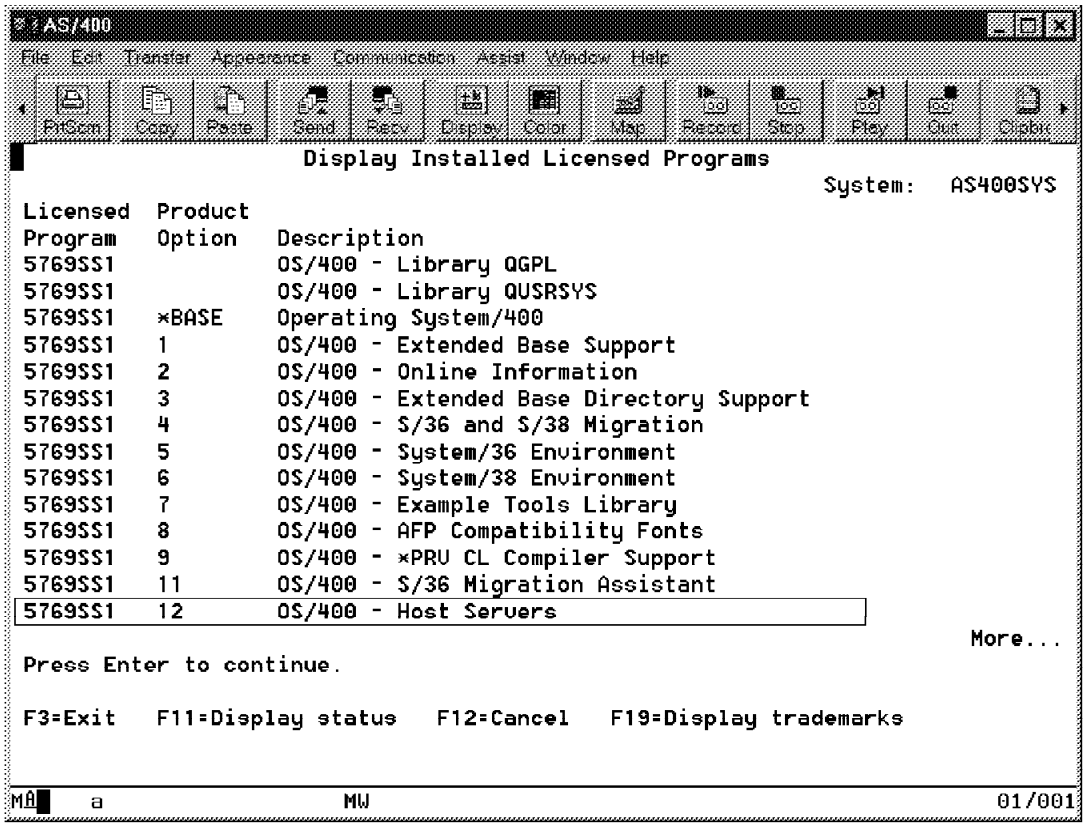


Figure 16. Display Licensed Programs with Product Options

If you are planning to develop client/server Java applications on the AS/400 system, you may require all of the following licensed programs and operating system options:

- OS/400 - Host Servers (5769-SS1 Option 12)
- OS/400 - QShell Interpreter (5769-SS1 Option 30)
- AS/400 Toolbox for Java (5763-JC1)
- AS/400 Developer Kit for Java (5769-JV1)
- TCP/IP Connectivity Utilities for AS/400 (5769-TC1)

The Java classes provided in the AS/400 Toolbox for Java can be used to develop client-based applications and applets that access AS/400 resources. These applications or applets can access any AS/400 system running OS/400 V3R2, V3R7, V4R1, or V4R2 (in earlier releases such as V3R2, not all AS/400 Toolbox for Java features may be available due to certain host server functions not being available. For example, on OS/400 V3R2, DDM is not supported over a TCP/IP connection. This feature can be enabled on V3R7 and V4R1 by applying the appropriate PTFs.) These classes use TCP/IP sockets to connect to the host servers running on the AS/400 system. This is the reason why you must have the OS/400 Host Servers and TCP/IP Connectivity Utilities installed.

Running Java applications on the AS/400 system requires OS/400 Version 4 Release 2 (V4R2).

With V4R2, Java applications running on the AS/400 system must also use the AS/400 Toolbox for Java classes to access AS/400 resources such as data

queues, print and spool support, execute OS/400 commands, or call AS/400 programs.

For this reason, you must make sure that the OS/400 Host Servers, TCP/IP Connectivity Utilities, and AS/400 Toolbox for Java are installed on your AS/400 system.

With V4R2 of OS/400, you have two new licensed programs:

- The AS/400 Developer Kit for Java (5769-JV1)
- The AS/400 Toolbox for Java (5763-JC1)

The AS/400 Developer Kit for Java (5769-JV1) provides the Java classes or Application Programming Interfaces (APIs) defined by JavaSoft. The current implementation conforms to level 1.1.4 of the Java Development Kit (JDK) specification. The QShell Interpreter (OS/400 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.

The AS/400 Toolbox for Java (5769-JC1) provides a set of Java classes that simplify the access of AS/400 resources from a Java application. These classes can be used in a client Java application or in a server Java application.

Using the Page Down key (or the Scroll Up key), page through the **Display Installed Licensed Programs** displays and look for the required OS/400 options and licensed programs. Figure 17 on page 37 shows **5769-SS1 Option 30, OS/400 - QShell Interpreter**.

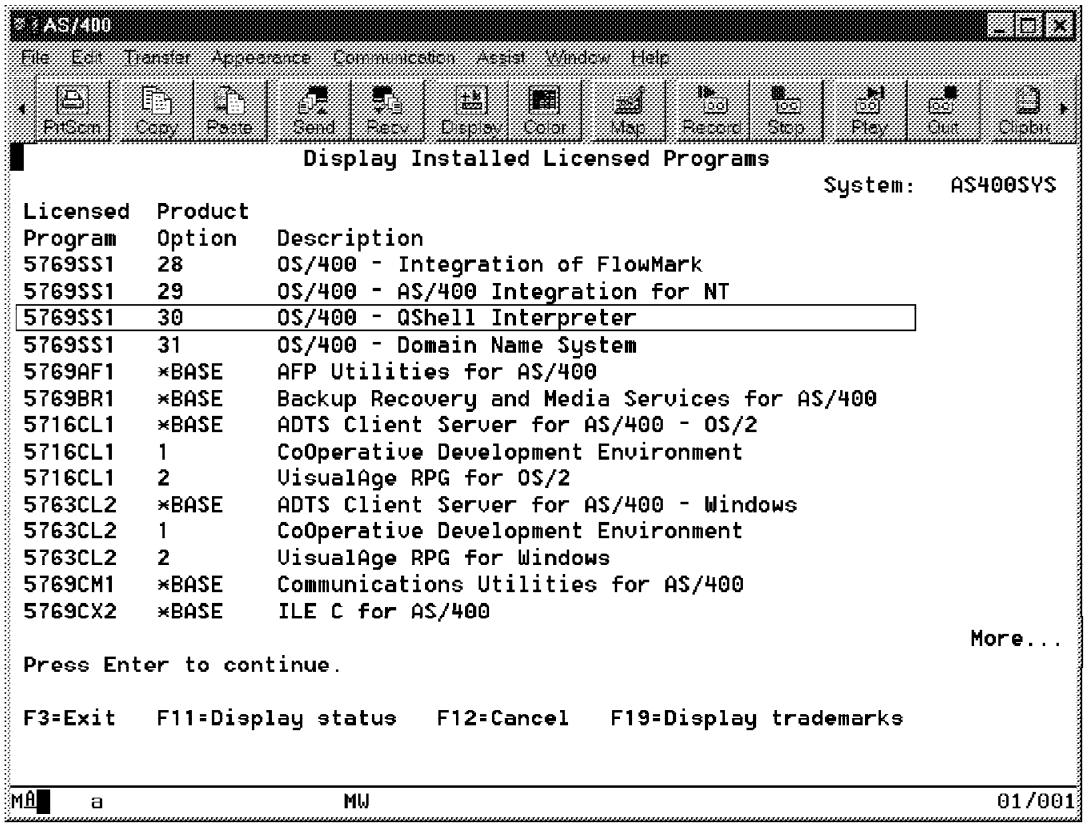


Figure 17. Display Installed Licensed Programs and Product Options (Option 30)

Browse through the various pages of the displayed licensed programs and look for all the programs previously listed. Make sure you find the two new V4R2 licensed programs, **5763-JC1 AS/400 Toolbox for Java** and **5769-JV1 AS/400 Developer Kit for Java**, shown in Figure 18 on page 38.

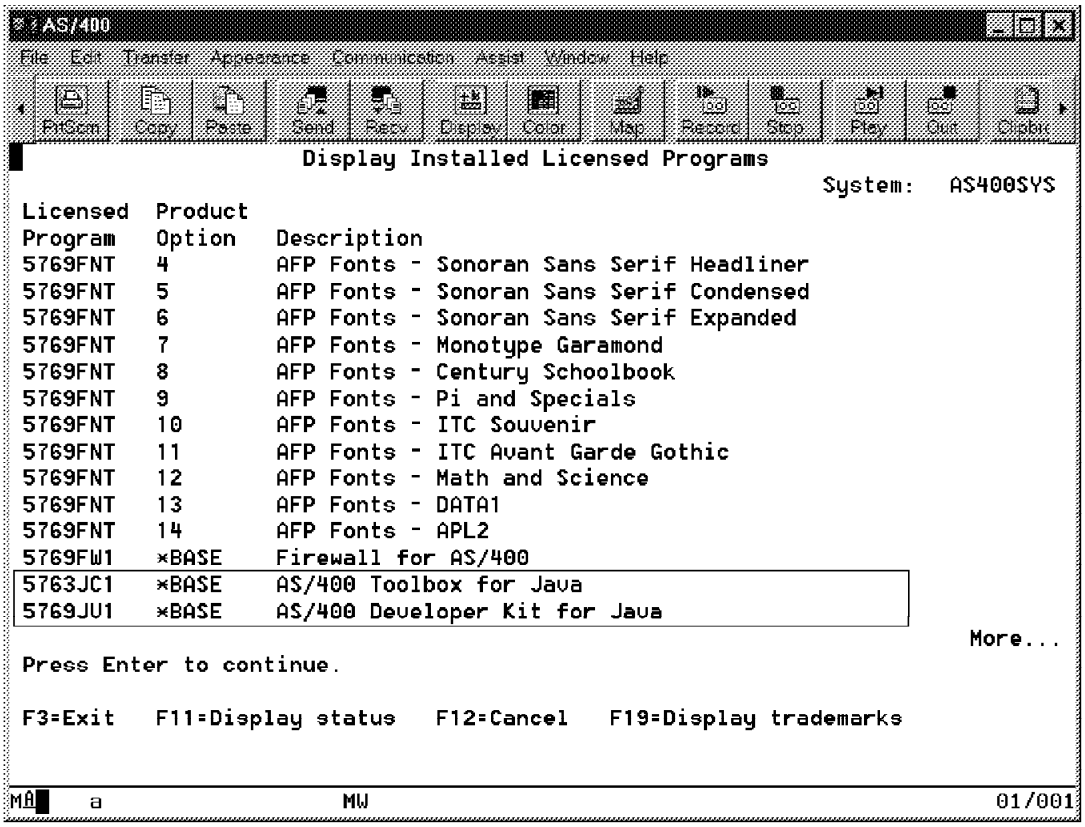


Figure 18. Display Installed Licensed Programs (5769-JC1 and 5769-JV1)

If you find that **5763-JC1 AS/400 Toolbox for Java** or **5769-JV1 AS/400 Developer Kit for Java** are not installed then follow the procedures in 3.2, “Manually Installing Java Support on AS/400 System” on page 41 to install them.

Continue paging down until you find the **5769-TC1 TCP/IP Connectivity Utilities for AS/400** licensed program shown in Figure 19 on page 39.

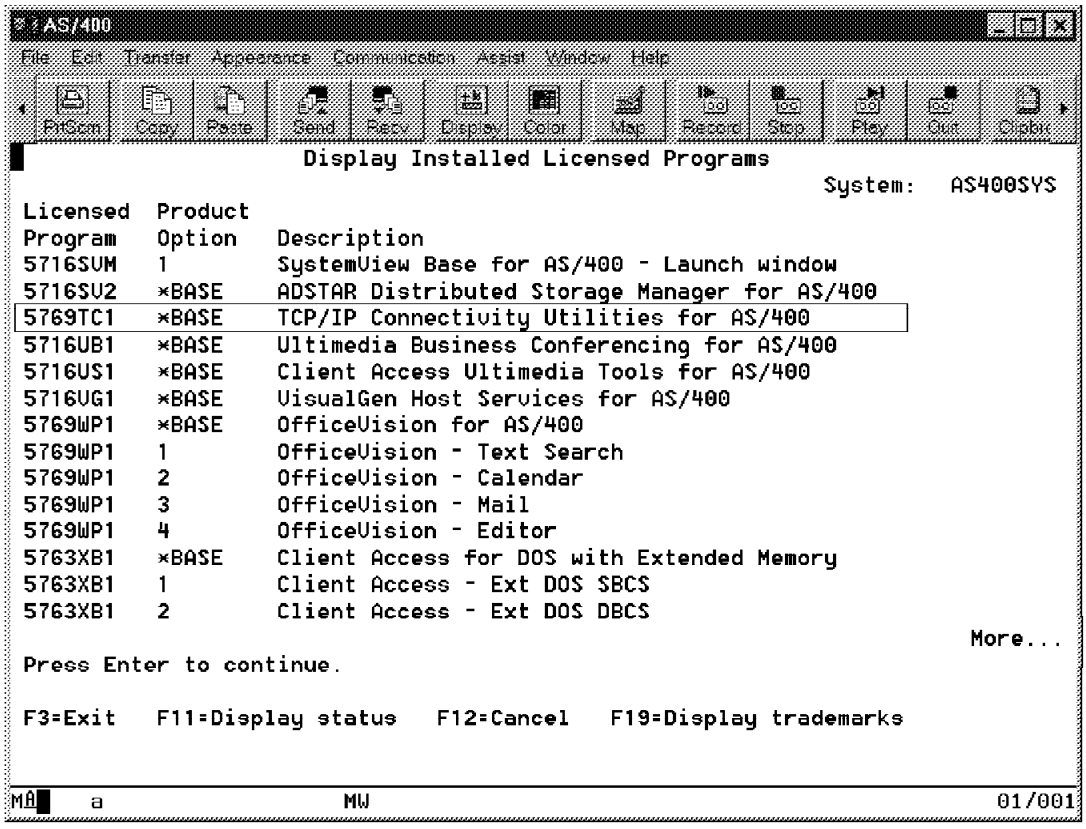


Figure 19. Display Installed Licensed Programs (Host Servers)

If you plan on using PCs running Microsoft's 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 IFS to store Java source code and Java bytecodes from your PC. In this case, look for the **5763-XD1 Client Access for Windows 95/NT** licensed program shown in Figure 20 on page 40.

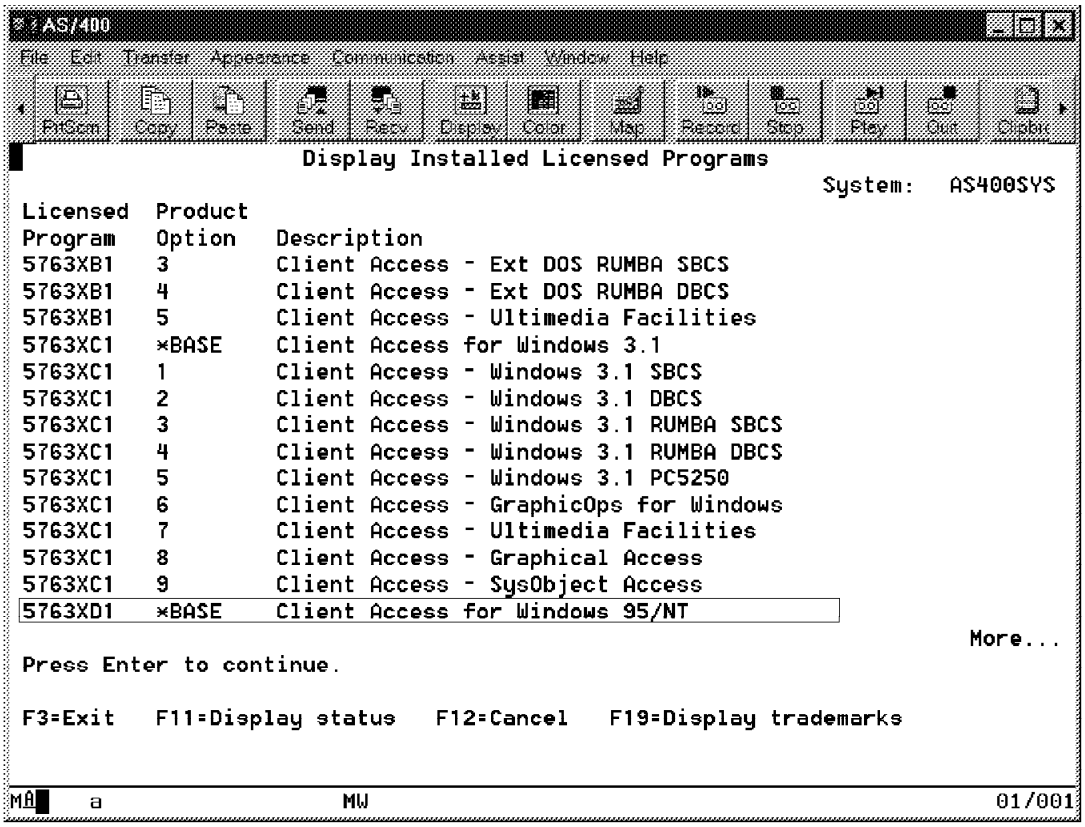


Figure 20. Display Installed Licensed Programs (5763-XD1)

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 **AS/400 Client Access Family for Windows, 5769-XW1**. Make sure this licensed program is installed as shown in Figure 21 on page 41.

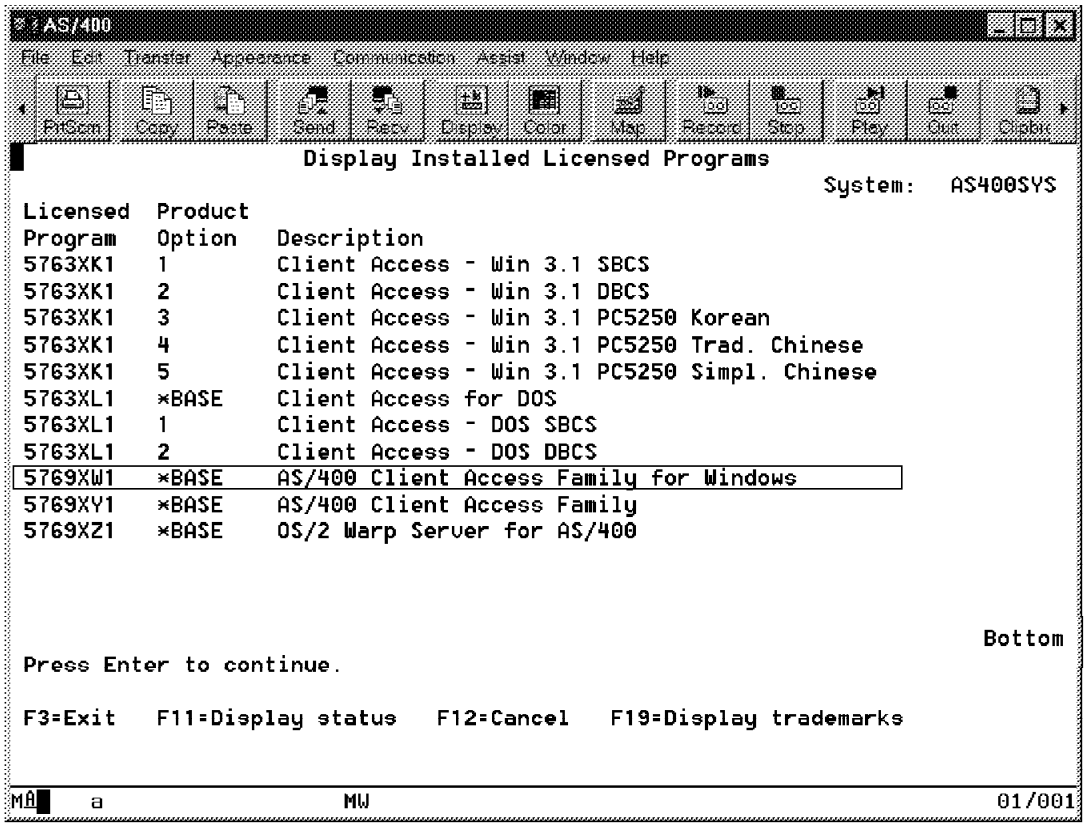


Figure 21. Display Installed Licensed Programs (5769-XW1)

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

3.2 Manually Installing Java Support on AS/400 System

In this section, we go through the steps required to install one or more of the software functions needed to run Java on your AS/400 system.

First, we guide you step-by-step in the installation process; then we cover loading and applying the latest PTF Cumulative Package.

3.2.1.1 Install OS/400 Options And Licensed Programs

Return to the **Work with Licensed Programs** menu using the **F3=Exit** or the **F12=Cancel** key from the **Display Installed Licensed Programs** display. On the **Work with Licensed Programs** menu, type option **11** as shown in Figure 22 on page 42.

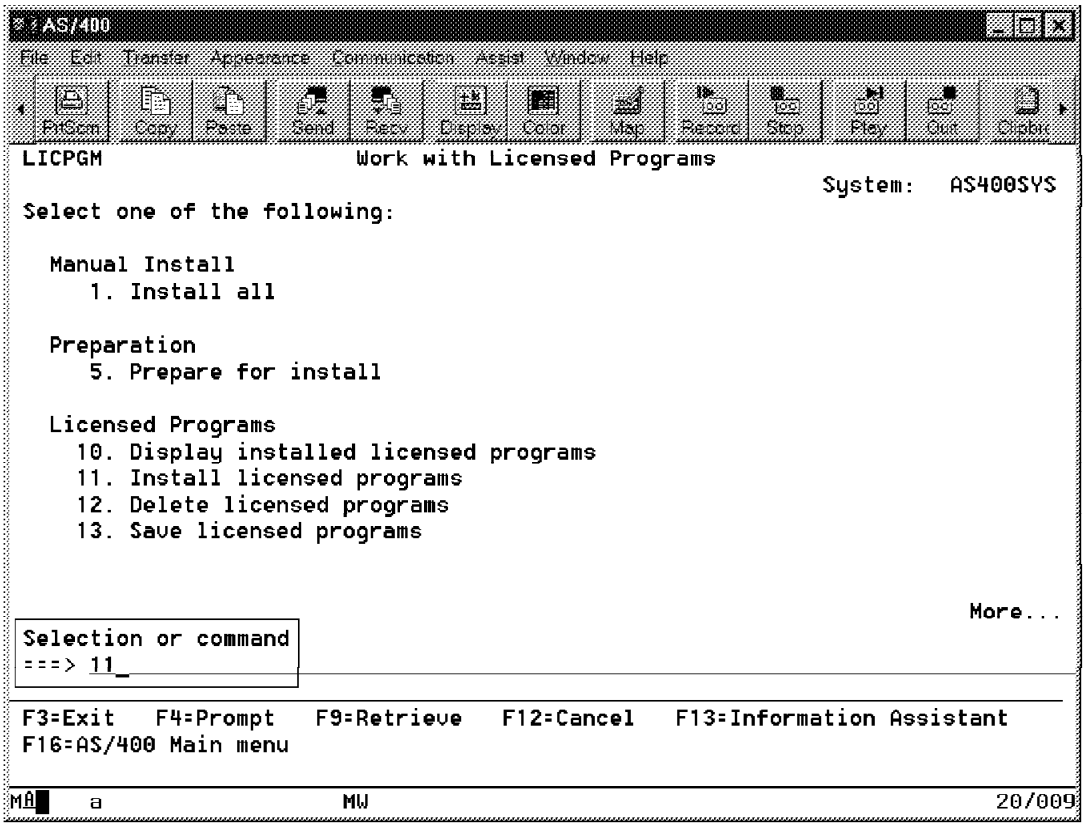


Figure 22. Work with Licensed Programs Display

Press the **Enter** key. The **Install Licensed Programs** display is shown as in Figure 23 on page 43.

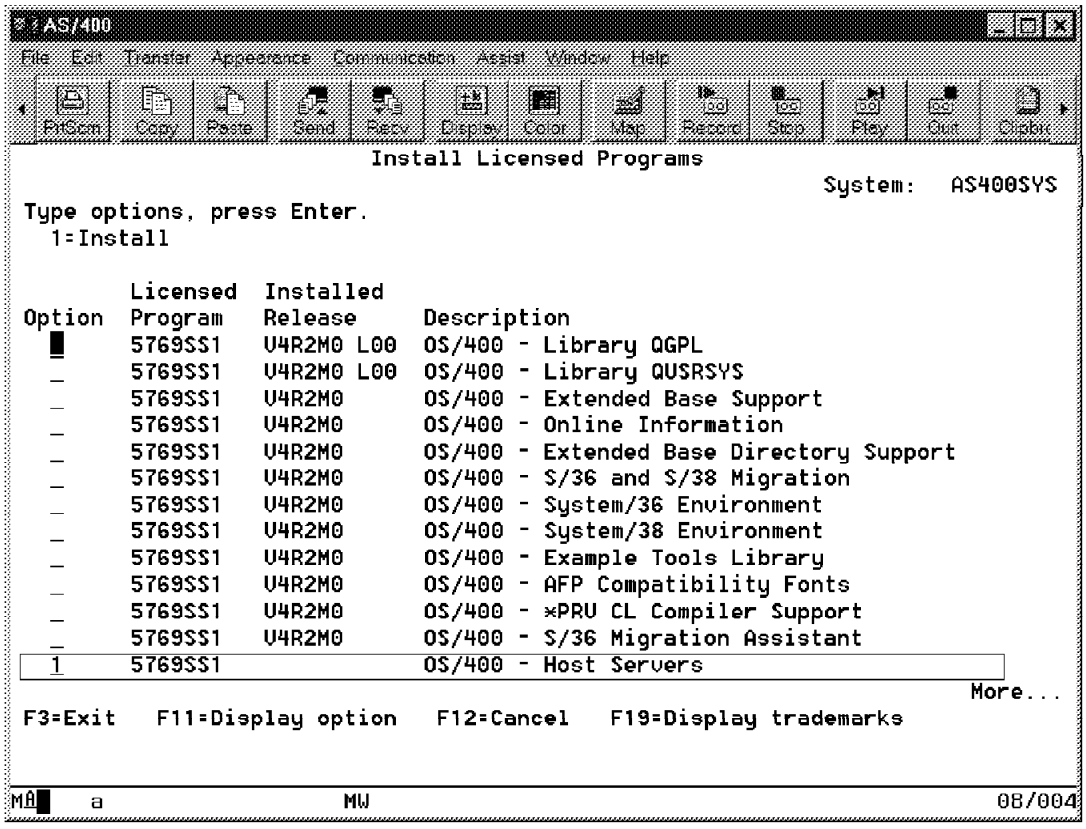


Figure 23. Install Licensed Programs Display

On the appropriate line in the list, enter a **1** next to **5769-SS1 OS/400 - Host Servers** if this option is not already installed on your system. Use the Page Down or the Scroll Up key until you see **5769-SS1 OS/400 - QShell Interpreter** shown in Figure 24 on page 44.

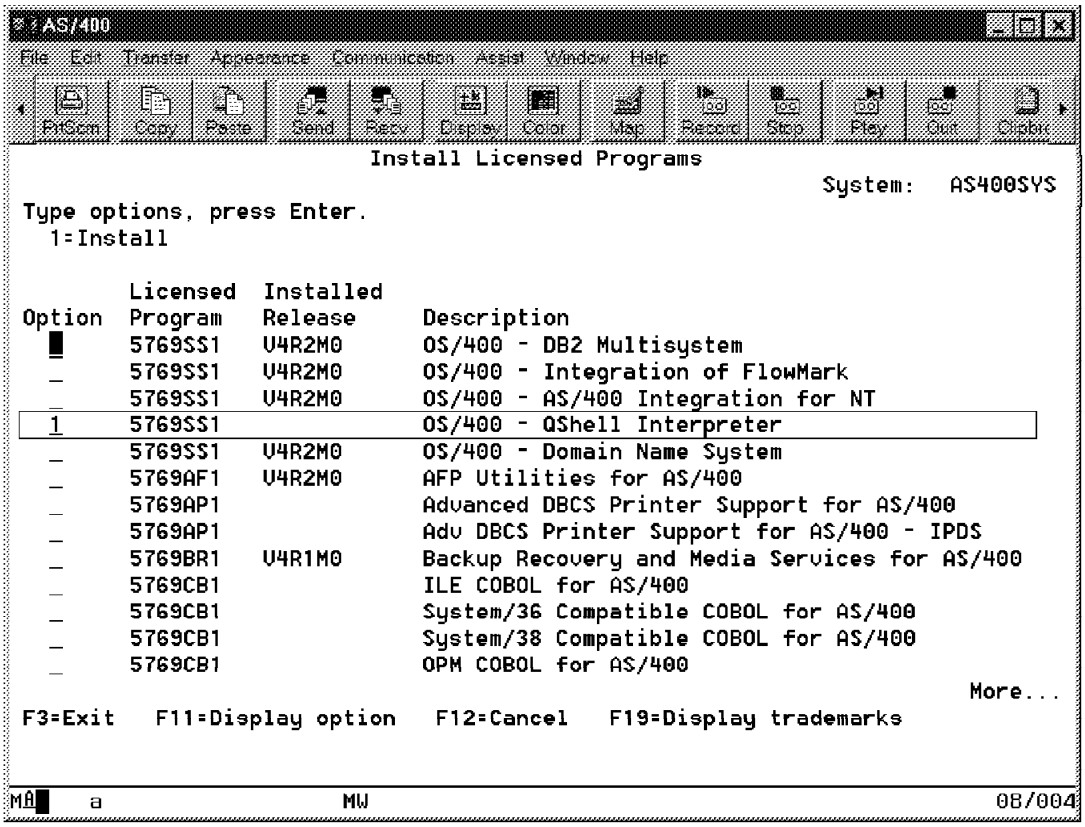


Figure 24. Install Licensed Programs Display (Installing OS/400 Host Servers)

Type a **1** to select the *5769-SS1 OS/400 - QShell Interpreter* option if it is not already installed on your system. This is a new option of OS/400 V4R2 that provides a text-based command entry environment designed to allow you to use most Java standard commands from any AS/400 workstation the same as from a DOS command entry window on a PC.

This is a required option as an OS/400 style command is not provided for every Java command.

In V4R2 of OS/400, the current Java CL commands are:

- **CRTJVAPGM** to compile a Java program on the AS/400 system. This command converts Java bytecodes 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 containing Java bytecodes. Java bytecodes can be produced by any PC based Java compiler and then copied to the AS/400 IFS. Or you can run the **javac** command on the AS/400 system using the OS/400 Java QShell Interpreter support; then create the AS/400 executable program from an OS/400 command entry display.
- **RUNJVA** (or **JAVA**) to execute 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 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** displays some details about the AS/400 executable Java program produced by the CRTJVAPGM command.
- **DLTJVAPGM** deletes the AS/400 executable Java program and removes the link from the associated **.class** file.

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

Use the Page Down or the Scroll Up key until you see the new V4R2 licensed programs required for running Java within the AS/400 system. These are **5763-JC1 AS/400 Toolbox for Java** and **5769-JV1 AS/400 Developer Kit for Java**. Type a **1** next to each one on these two new licensed programs to select them for installation, as shown in Figure 25.

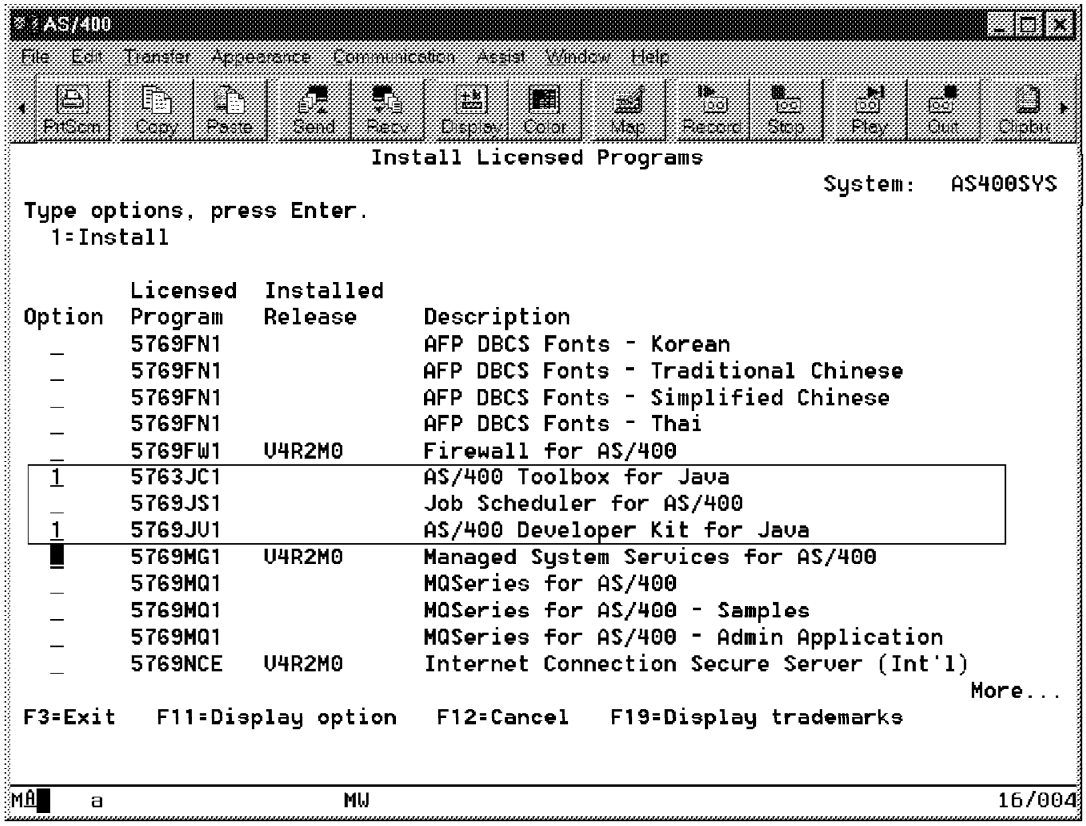


Figure 25. Install Licensed Programs (5763-JC1 and 5769-JV1)

The **AS/400 Toolbox for Java** provides a set of Java classes aimed at simplifying the access to AS/400 resources from a Java application. These classes are used on the AS/400 server to allow AS/400 native Java applications access to data queues, printer and spooling functions, executing 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 IFS, and JDBC access to DB2/400 Database.

The **AS/400 Toolbox for Java** classes can also be used on the client to develop Java applications or applets that access AS/400 resources. Please refer to *Accessing the AS/400 System 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 or applets that access the AS/400 system.

The **AS/400 Developer Kit for Java (5769-JV1)** provides all the Java classes as defined in the Java Development Kit Version 1.1.4 and published by JavaSoft. This set of application programming interfaces (APIs) allows for running any Java compliant program on the AS/400 system. This new licensed program requires OS/400 Version 4 Release 2 (V4R2).

Then, you need to scroll further down the list until you find **5769-TC1 TCP/IP Connectivity Utilities for OS/400**. Type **1** next to this licensed program to install if it is not already installed on your AS/400 system. This is shown in Figure 26.

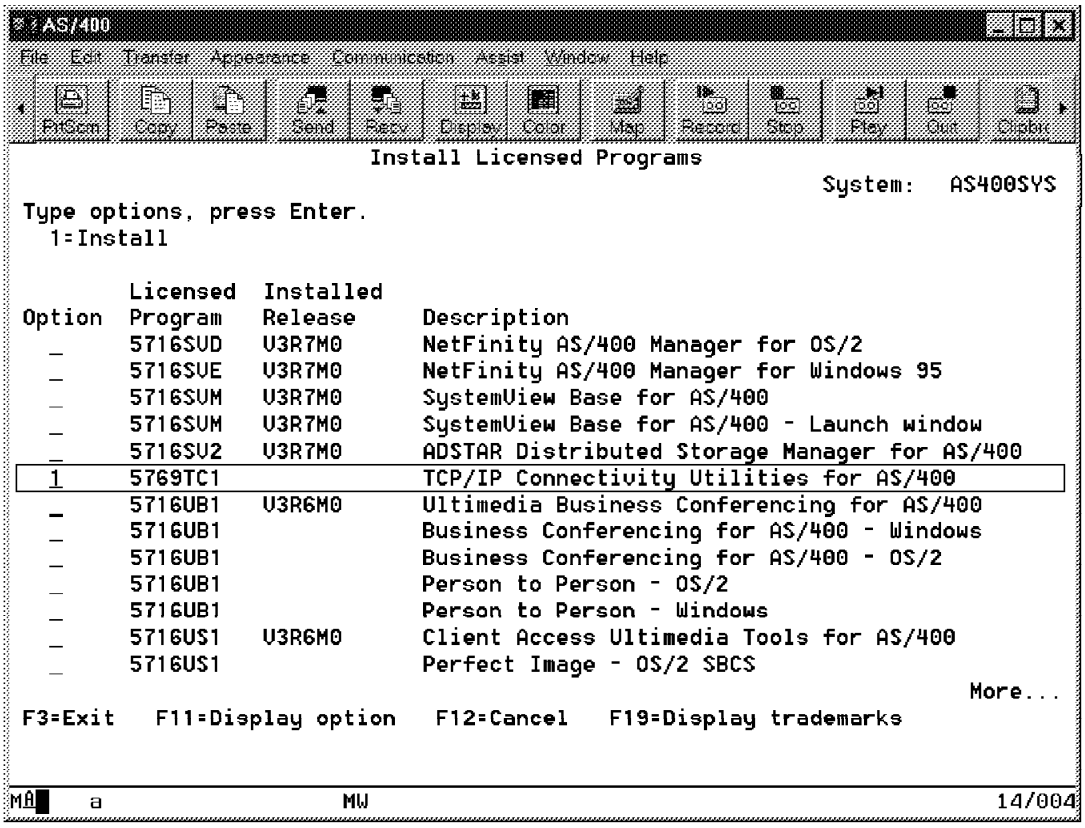


Figure 26. Install Licensed Programs (5769-TC1 TCP/IP Connection Utilities)

If you plan on using PCs running Microsoft's Windows 95 or Windows NT Workstation as Java clients or as Java Remote AWT clients for your AS/400 server based Java applications, you might consider installing **Client Access for Windows 95/NT** on your AS/400 system. To do this, you need to scroll down the **Install Licensed Programs** display until you see the **5763-XD1 Client Access for Windows 95/NT** licensed program shown in Figure 27 on page 47.

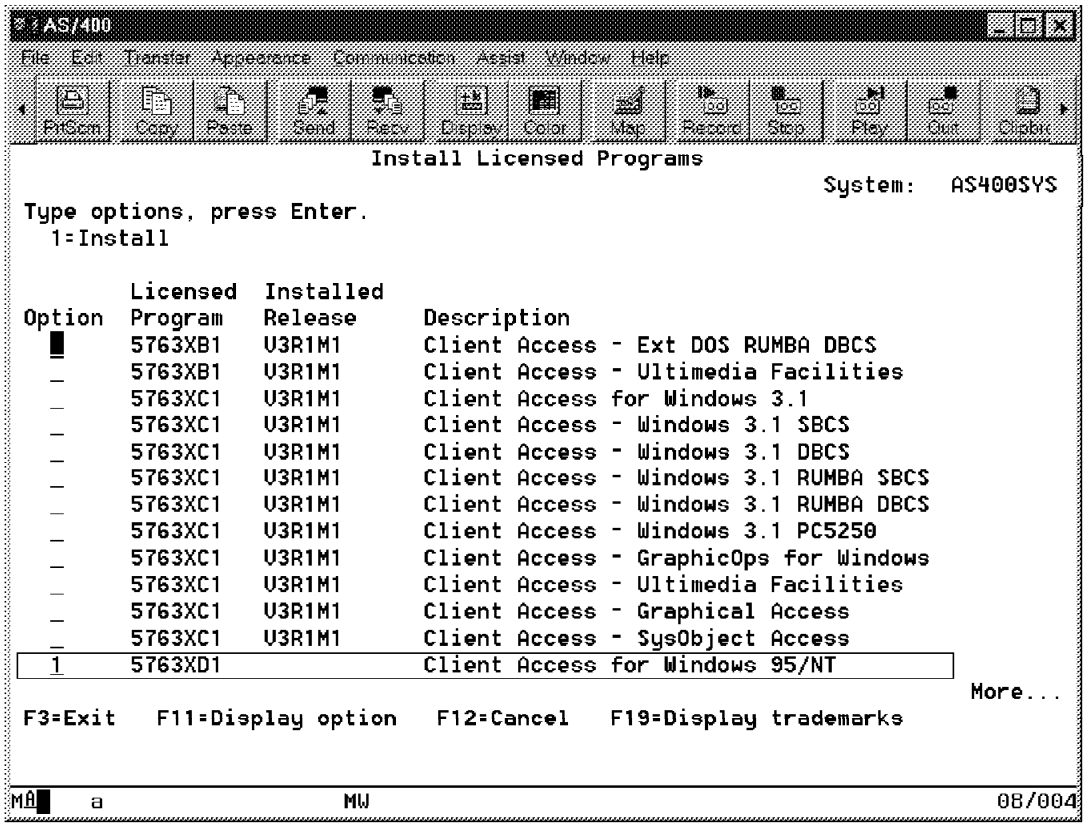


Figure 27. Install Licensed Programs (5763-XD1 Client Access for Windows 95/NT)

Type a **1** next to the **5763-XD1 Client Access for Windows 95/NT** entry on the display if this licensed program is not already installed on your AS/400 system. Then scroll down the display using the Page Down key or the Scroll Up key until you see **5769-XW1 AS/400 Client Access Family for Windows**. Type a **1** on the corresponding line to select this licensed program for installation if it is not currently installed on your machine. Figure 28 on page 48 shows this selection.

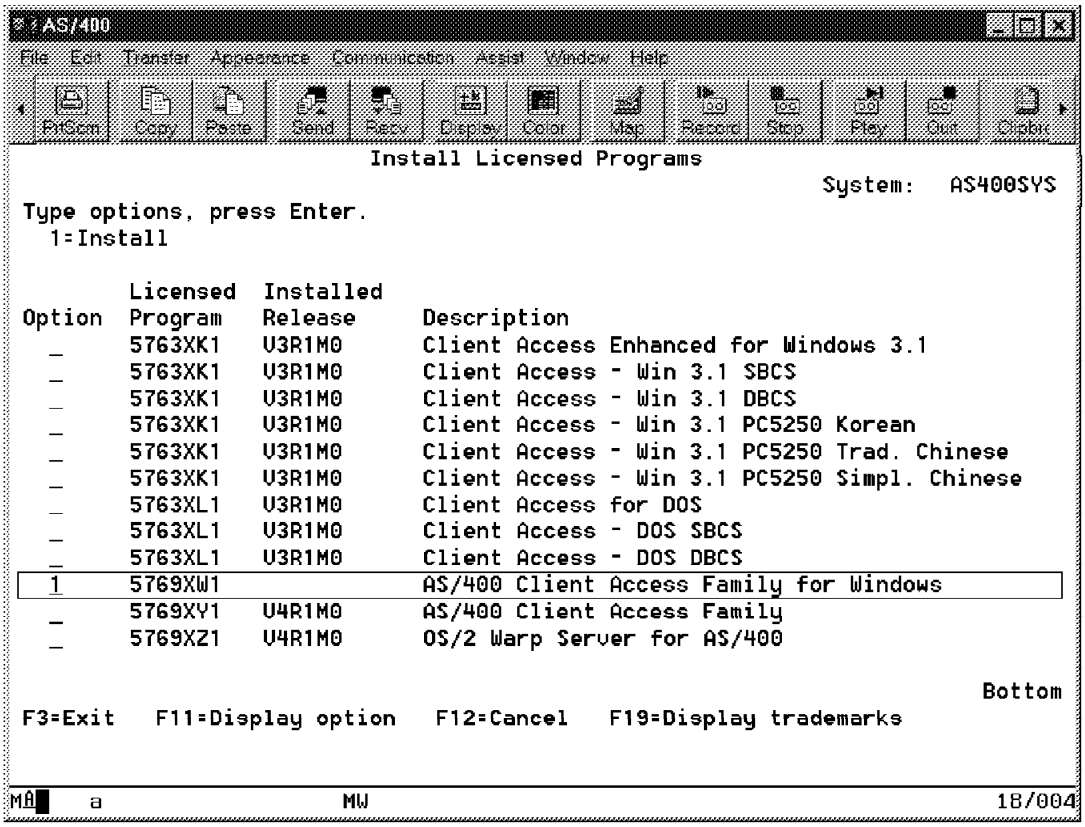


Figure 28. Install Licensed Programs (5769-XW1 Client Access Family for Windows)

You have now completed selecting the required OS/400 options and licensed programs. Press Enter to see the **Confirm Install of Licensed Programs** display shown in Figure 29 on page 49.

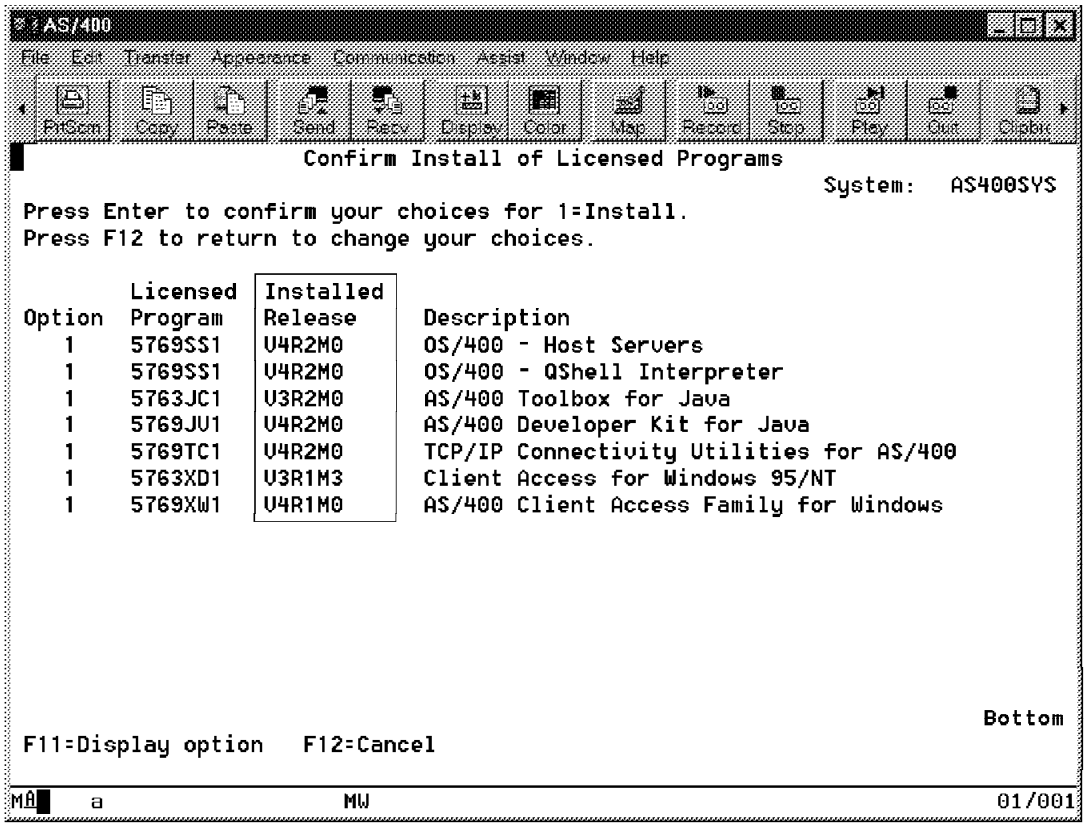


Figure 29. Confirm Install of Licensed Programs Display

As you can see on this display, most of the OS/400 options and licensed programs are at the V4R2 level. However, as we have previously mentioned, the AS/400 Toolbox for Java Licensed Program can be installed on any AS/400 system running OS/400 V3R2, V3R7, V4R1, or V4R2. This is depicted by the name of the licensed program (5763-JC1) that conforms to OS/400 V3R2 standards and by the V3R2M0 information shown in the **Installed Release column**.

The **Client Access for Windows 95/NT Licensed Program** is now at the V3R1M3 level. This level can be installed on any AS/400 system running OS/400 V3R1, V3R2, V3R6, V3R7, V4R1, or V4R2. *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 licensed programs are not installed on your system, the corresponding Installed Release information is blank.

Press **Enter** to see the **Install Options** display shown in Figure 30 on page 50.

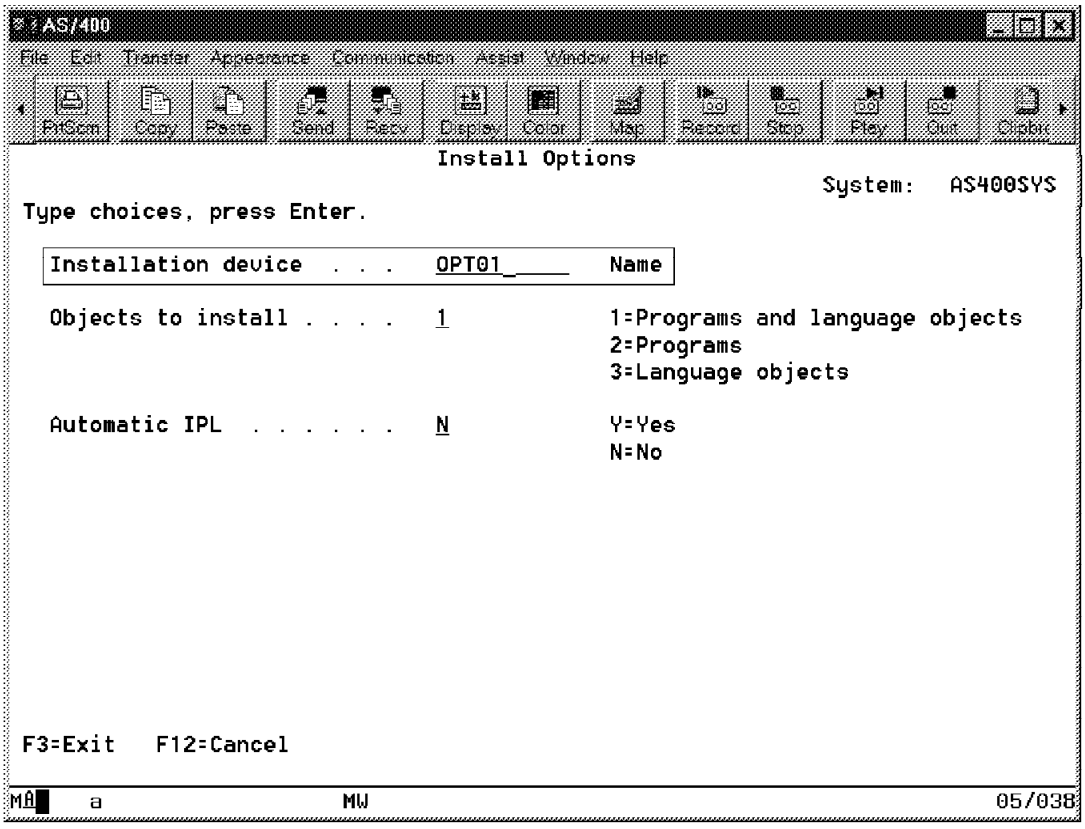


Figure 30. PTF Install Options

On this display, type the device name of the CD-ROM drive of your AS/400 system. This is usually OPT01. Type **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 you insert the OS/400 Volume 1 in the CD-ROM drive; then 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**).

3.2.1.2 Install Cumulative PTF Package

When the installation completes, you must install the latest cumulative PTF package to load and apply all the required PTFs to the newly installed OS/400 options and licensed programs. To proceed with installing the PTFs, use the **F3=Exit** key or the **F12=Cancel** key to return to the **AS/400 Main Menu**; then type **GO PTF** on the **AS/400 Main Menu** command line as shown in Figure 31 on page 51.

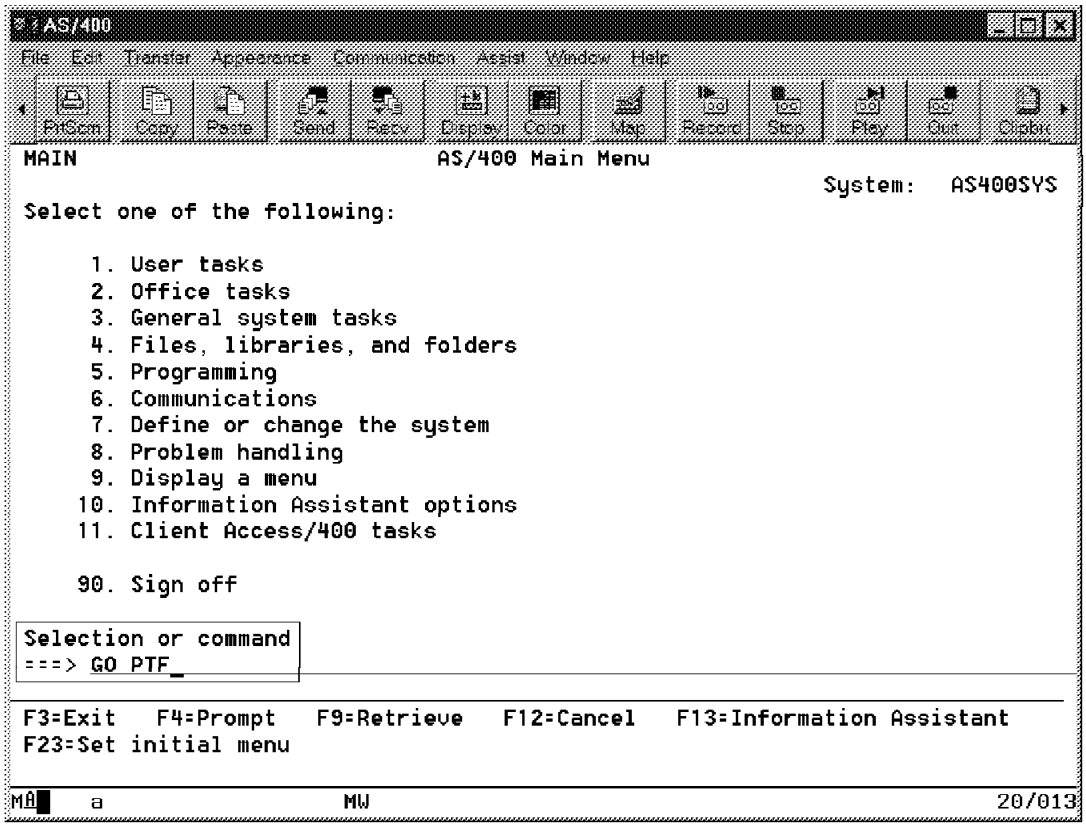


Figure 31. AS/400 Main Menu Display

Then press the **Enter** key. The **Program Temporary Fix** menu is shown in Figure 32 on page 52.

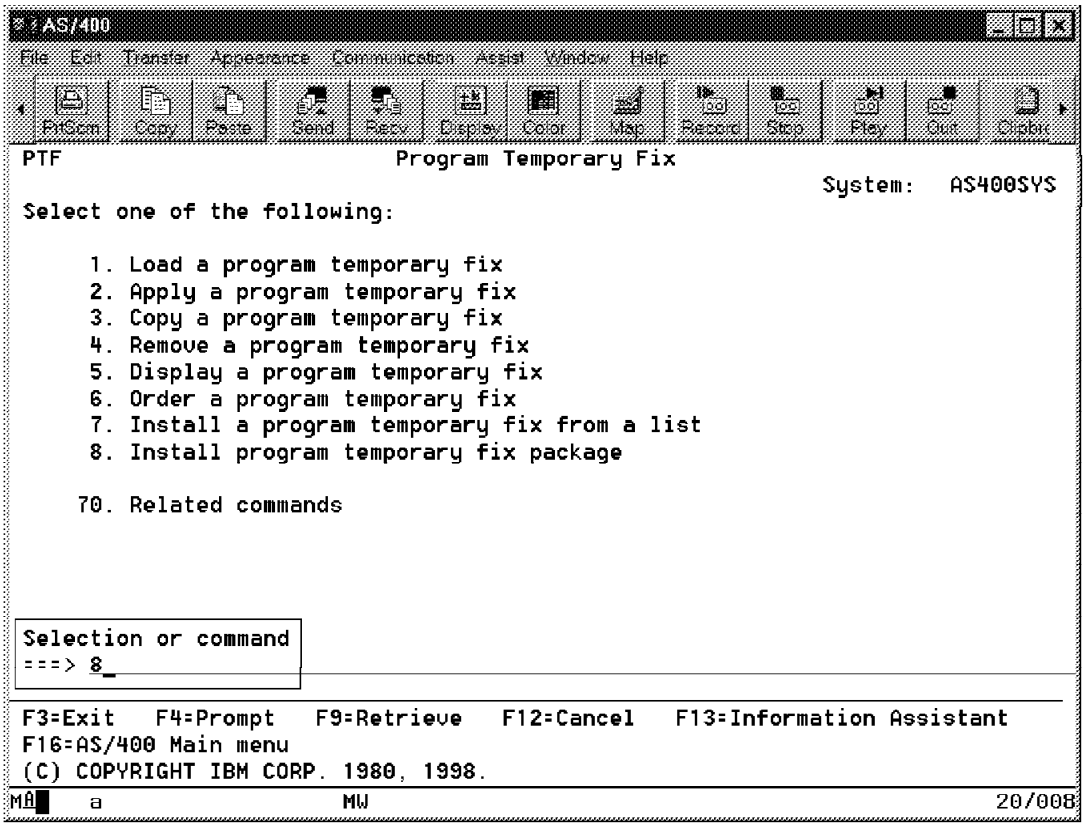


Figure 32. Program Temporary Fix Display

On the **Program Temporary Fix** menu, type 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 33 on page 53.

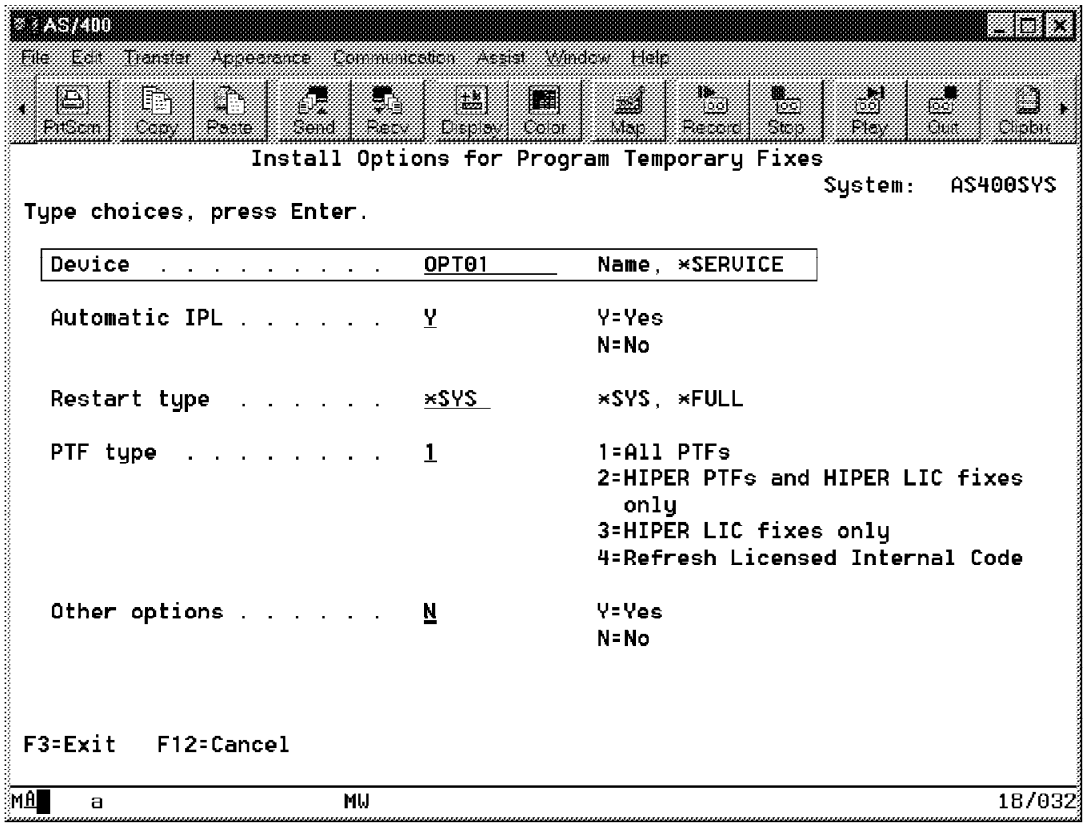


Figure 33. Install Options for Program Temporary Fixes

On this display, type **OPT01** (or the name of your AS/400 system CD-ROM drive or tape device if it differs from OPT01) and keep the default values for all the other options shown in the previous picture. Make sure 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. *But beware, 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

You now need to install a Java run time on your workstation. In this redbook, we show you how to install the JavaSoft Java Development Kit (JDK) on a PC running the Microsoft Windows 95 operating system.

You may choose to use any other workstation environment capable of running a Java Virtual Machine (JVM). Such environments include RS/6000 with AIX workstations, PCs running IBM's OS/2 Warp Version 4 or Microsoft's Windows NT Operating Systems, Apple's Macintosh, or any JVM enabled Network Computer such as IBM's 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 new programming language designed to develop applications that you can easily deploy in a network computing environment.

3.3.1 Downloading JavaSoft JDK from Internet

To install a Java environment on your workstation, you must obtain the required software. The Java Development Kit can be downloaded free of charge from the JavaSoft World Wide Web (WWW) 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 such a CD-ROM from JavaSoft. For more details, please visit JavaSoft Internet site at the URL shown in the following example and Figure 34.

```
http://www.javasoft.com/products/jdk/1.1/
```

Figure 34. JavaSoft JDK 1.1 Home Page

To download the Java Development Kit 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 1.1.4. In the future, there will be later releases of the JDK (for example, JDK 1.1.5). In the documentation provided by JavaSoft, it is stated that these 1.1.n releases will be

upward compatible. So when a later release becomes available for your PC, it should work the same or similar to the JDK 1.1.4 examples shown here.

From this site, simply follow the instructions as they are listed on the HTML page (shown in Figure 34 on page 54):

Select **Microsoft Windows 95 / NT 4.0** from the drop-down list under Download JDK 1.1.4 software.

Click on the **Download Software** push button next to it as shown in Figure 35.

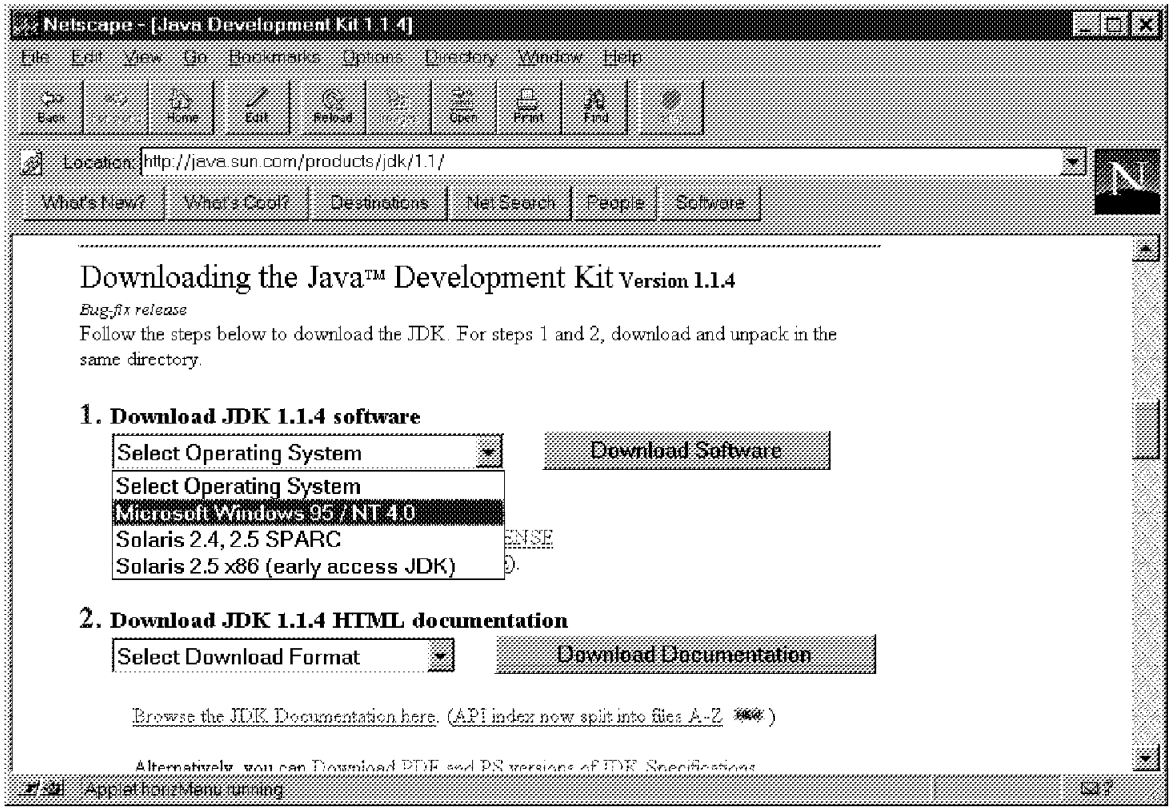


Figure 35. Downloading JDK 1.1.4 from the Internet

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 36 on page 56.

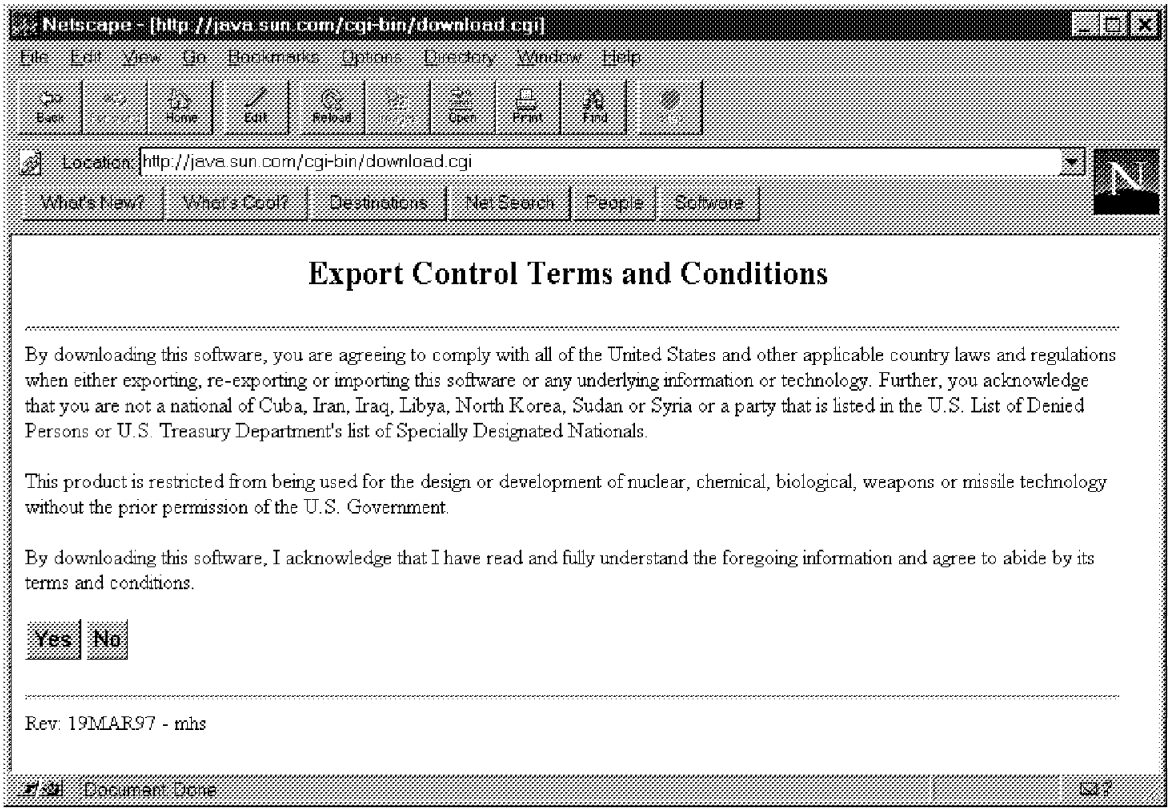


Figure 36. Export Control Terms and Conditions

If you are eligible to download the JDK, click on the **Yes** push button. The JDK Download Page is shown in Figure 37 on page 57.

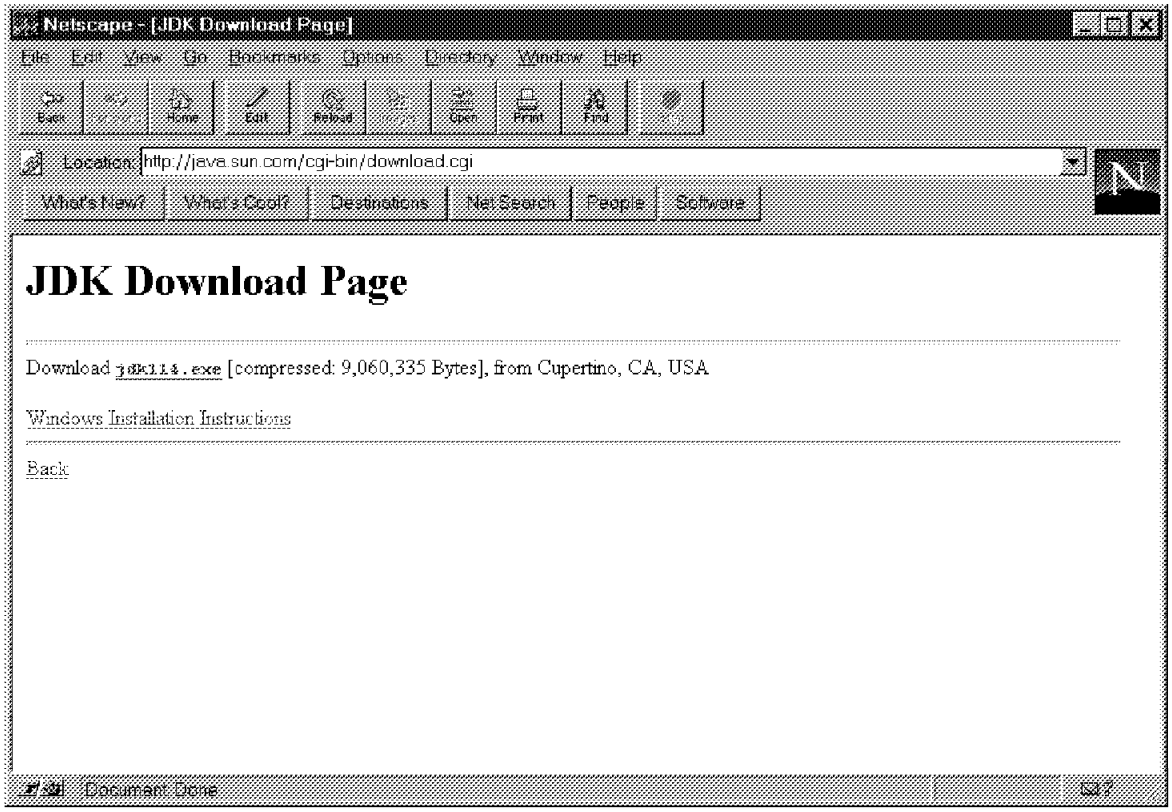


Figure 37. JDK Download Page (Windows Version)

Click on the Download **jdk114.exe** link to copy the install program named JDK114.EXE onto the hard disk of your PC. You are prompted to specify a path name (directory) and file name for saving the file shown in Figure 38. 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\
```

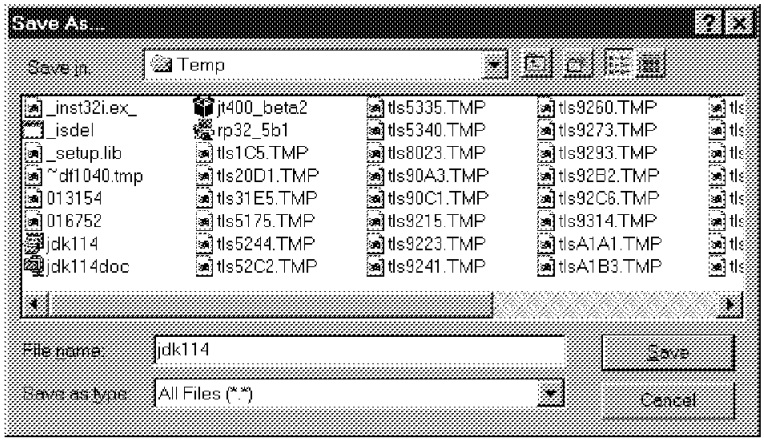


Figure 38. Directory Trees

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 9 060 335 bytes (compressed).

After copying the JDK114.EXE, return to the **Downloading the Java Development Kit Version 1.1.4** page at the following URL:

`http://www.javasoft.com/products/jdk/1.1/`

You may use the **Back** push button from your browser's toolbar to do so. Again, simply follow the instructions as they are listed on the HTML page:

Select **ZIP file for Windows** from the drop-down list under Download JDK 1.1.4 HTML documentation.

Click on the **Download Documentation** push button next to it as shown in Figure 39.

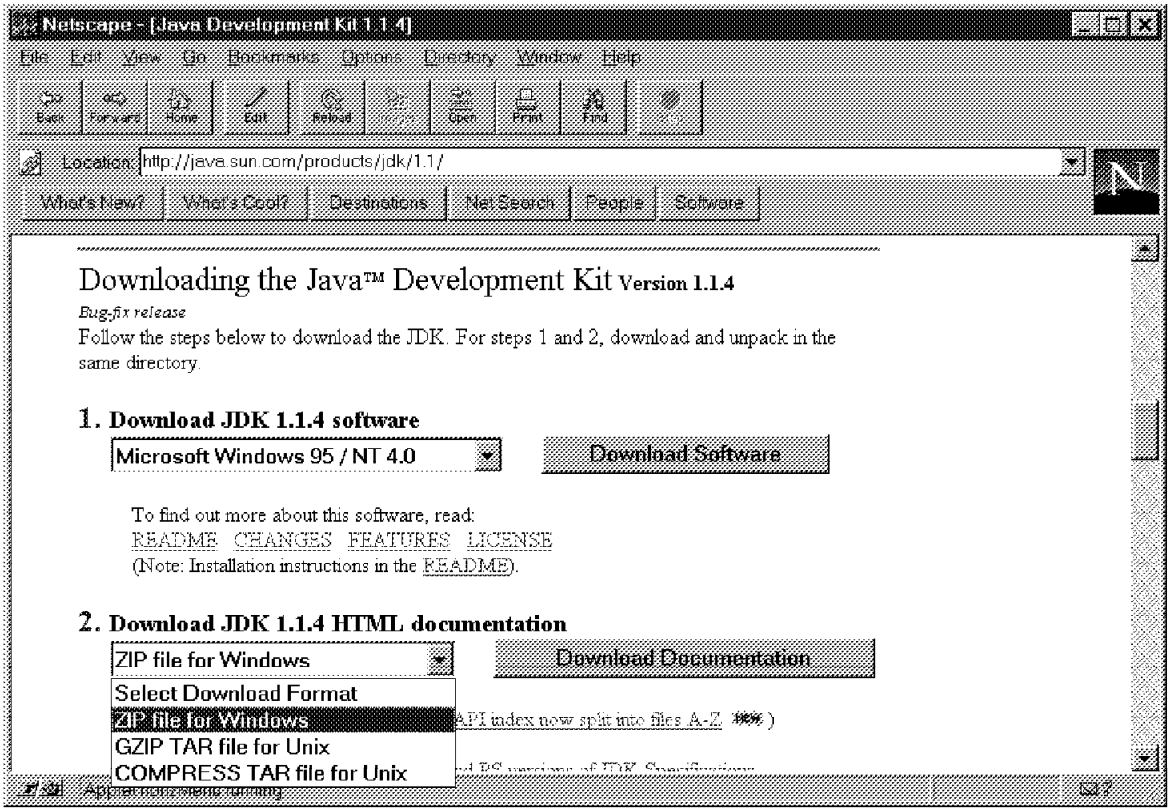


Figure 39. Downloading JDK 1.1.4 Documentation

This takes you to the JDK Download Page shown in Figure 40 on page 59.

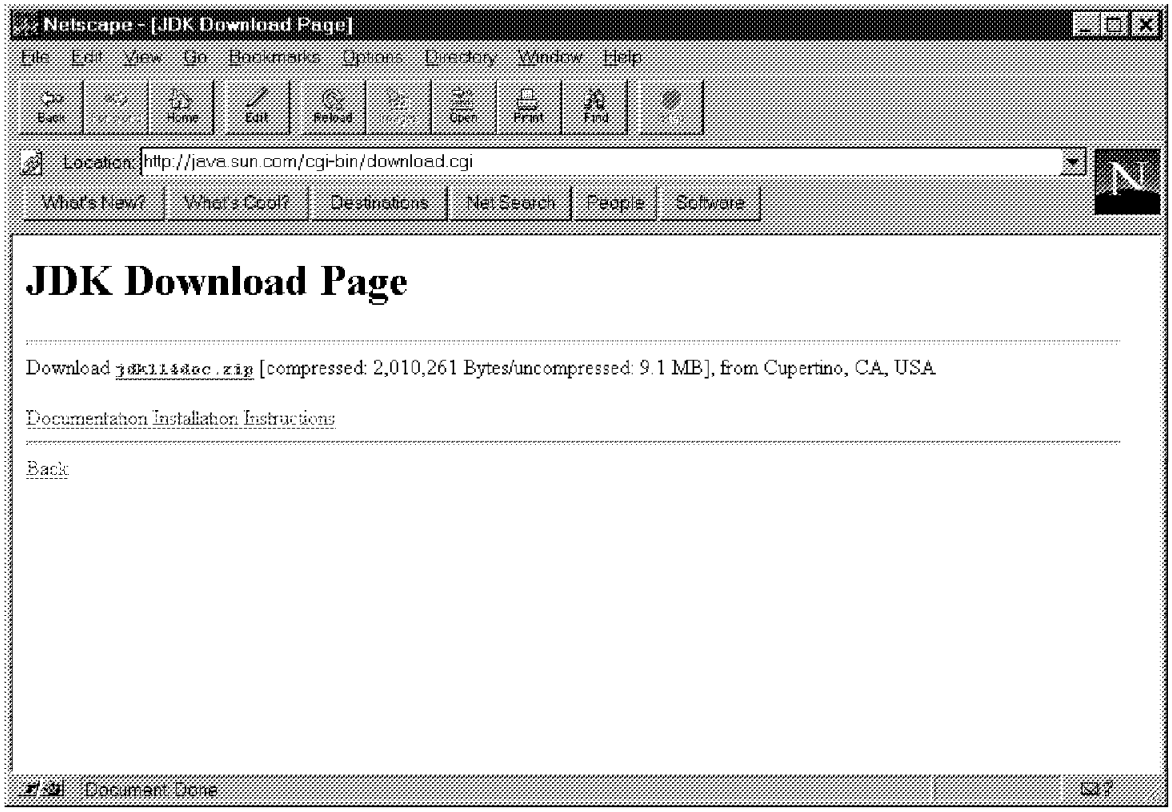


Figure 40. JDK Download Page (Documentation)

Click on the Download **jdk114doc.zip** link to copy the ZIP file containing the JDK documentation to your PC hard disk.

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

You are then prompted to specify a path name (folder) and file name as shown in Figure 41.

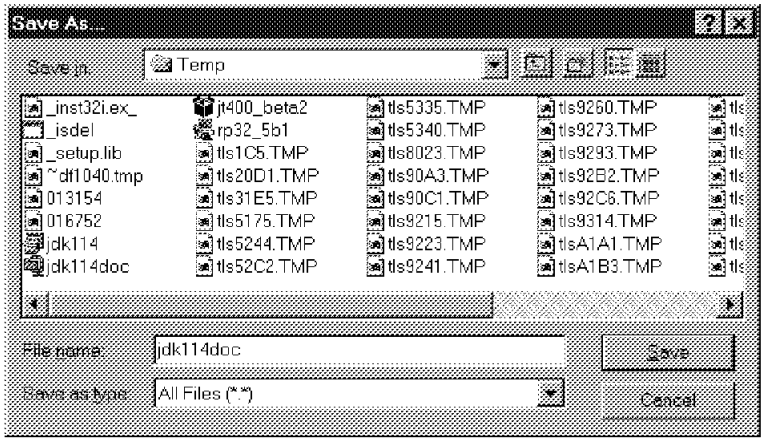


Figure 41. Downloading the .zip File to Correct Folder

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

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

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 as this file is 2 010 261 bytes (compressed).

You are now ready to begin installing the JDK and associated documentation on your PC. Start the Windows Explorer and find the two files you just downloaded. If you used the proposed folder and file names, look for the program **C:\WINDOWS\TEMP\JDK114.EXE**. Double-click on it to run the installation program. This creates a folder called **JDK1.1.4** on your disk and installs all the files and sub-folders composing the JDK.

Figure 42 shows the Windows 95 Explorer frame.

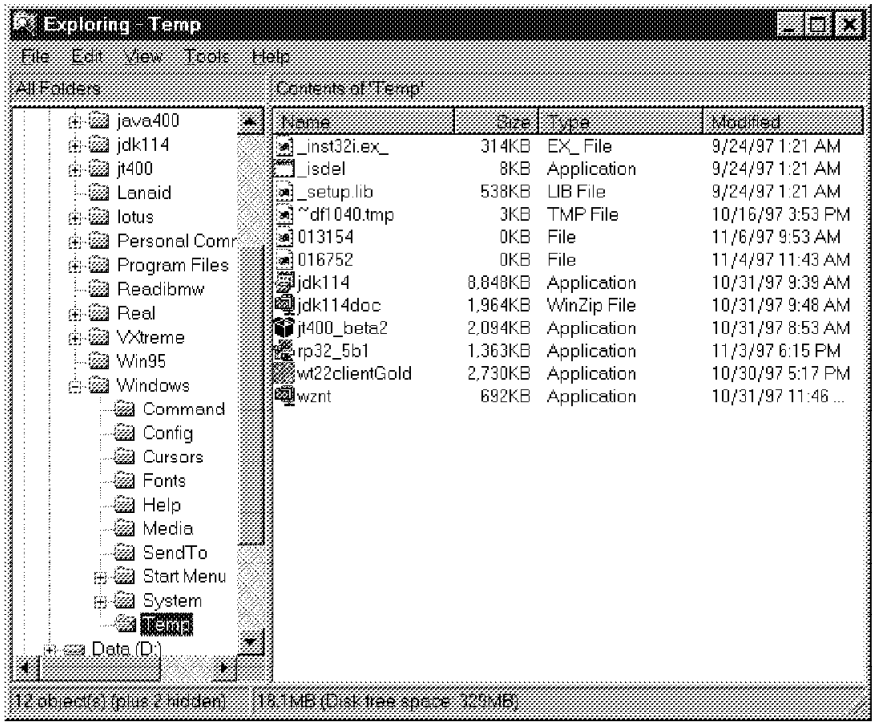


Figure 42. Locating JDK Documentation File for Installation

Then look for the documentation self extracting program. Again, if you used the proposed folder and file names, look for **C:\WINDOWS\TEMP\JDK114DOC.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 sub-folder named **docs** in the previously created **JDK1.1.4** folder.

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

Important Note

The Java core classes of the JavaSoft JDK 1.1.4 are contained in a compressed (ZIP) file named **CLASSES.ZIP** in **JDK1.1.4\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 various steps you need to follow to set up the proper Java environment both on your PC and on the AS/400 system.

3.4.1 Setting Up the Environment On Your PC

The first thing you need to do is to set up the **PATH** and the **CLASSPATH** environment variables on your workstation.

It is possible for you to run the JDK without modifying any system environment variables (such as **PATH** or **CLASSPATH**) or modifying the **AUTOEXEC.BAT** file. The path variable is merely a convenience to the developer and 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 instance, if you want to compile a Java program named **myclass.java**, you need to enter the following command at the DOS prompt:

```
C:\WINDOWS> C:\JDK1.1.4\BIN\javac myclass.java
```

This can become tedious and error prone, so usually you 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 simply by entering the following command at the DOS prompt:

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

3.4.1.1 Setting Up the PATH Variable

To change the path information in your **AUTOEXEC.BAT** file, you need to do the following steps:

- Start a text editor such as WordPad or Notepad.
- From the File menu open **c:\autoexec.bat**.
- Look for the **PATH** statement. Notice that the **PATH** statement is a series of directories separated by semi-colons (;). Windows looks for programs in the **PATH** directories in order, from left to right.
- Put the Java directory at the end of the path statement.
- Choose Save from the File menu to update your **AUTOEXEC.BAT** file.
- Exit from the text editor.

For example, in the following **PATH** statement, we have added the Java directory at the end:

```
SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\;C:\DOS;C:\JDK1.1.4\BIN
```

Figure 43 on page 62 shows the contents of a typical **AUTOEXEC.BAT** file.

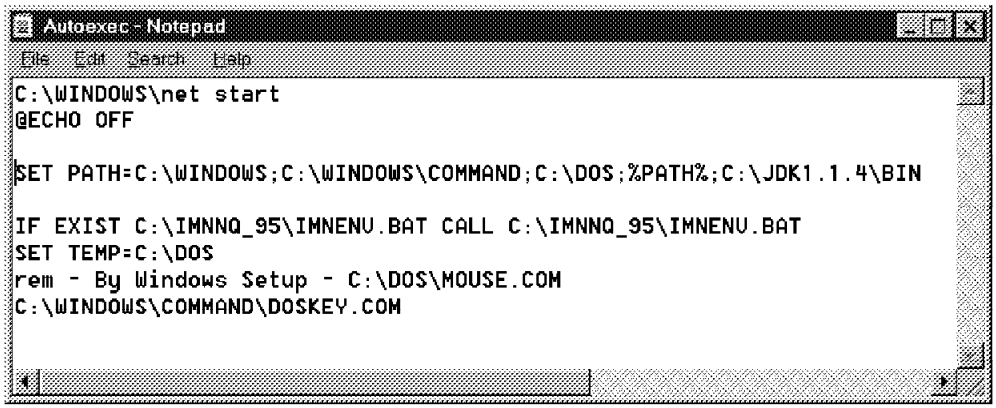


Figure 43. Editing AUTOEXEC.BAT Path Variable

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. It is preferable to re-boot your system, but it can be run interactively (care must be exercised as to what else will run again) by using the following commands at the **DOS prompt**:

```
C:\WINDOWS>cd \

C:>autoexec.bat
```

If while running the **AUTOEXEC.BAT** file, you get an "Out of environment space" error, you must increase the initial environment memory for your DOS session.

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. On the pop-up menu, select **Properties** to open the **MS-DOS Prompt Properties** window. Then select the **Memory** tab (the one in the middle). On the Memory tab, select 4096 from the drop-down list next to **Initial** environment as shown in Figure 44 on page 63.

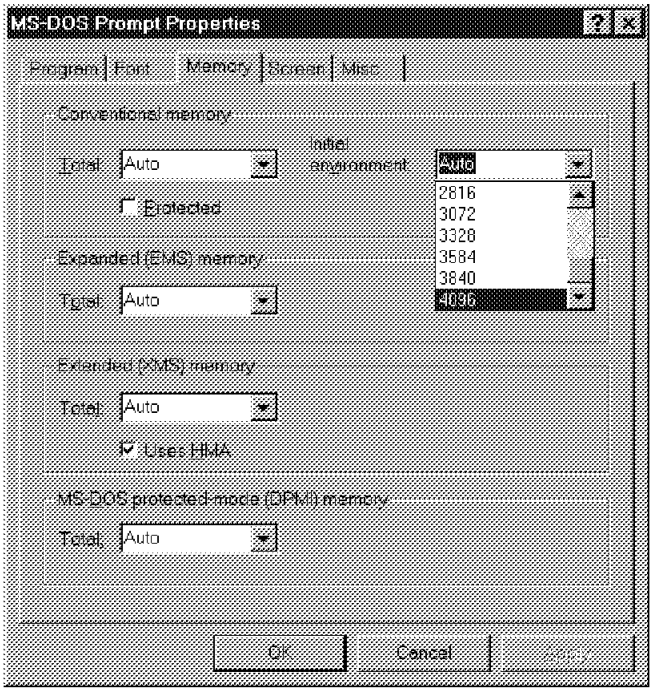


Figure 44. Changing Initial Memory Environment for DOS

Then click on the **Apply** push button to save your changes, and finally 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.

Now you can 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 64.

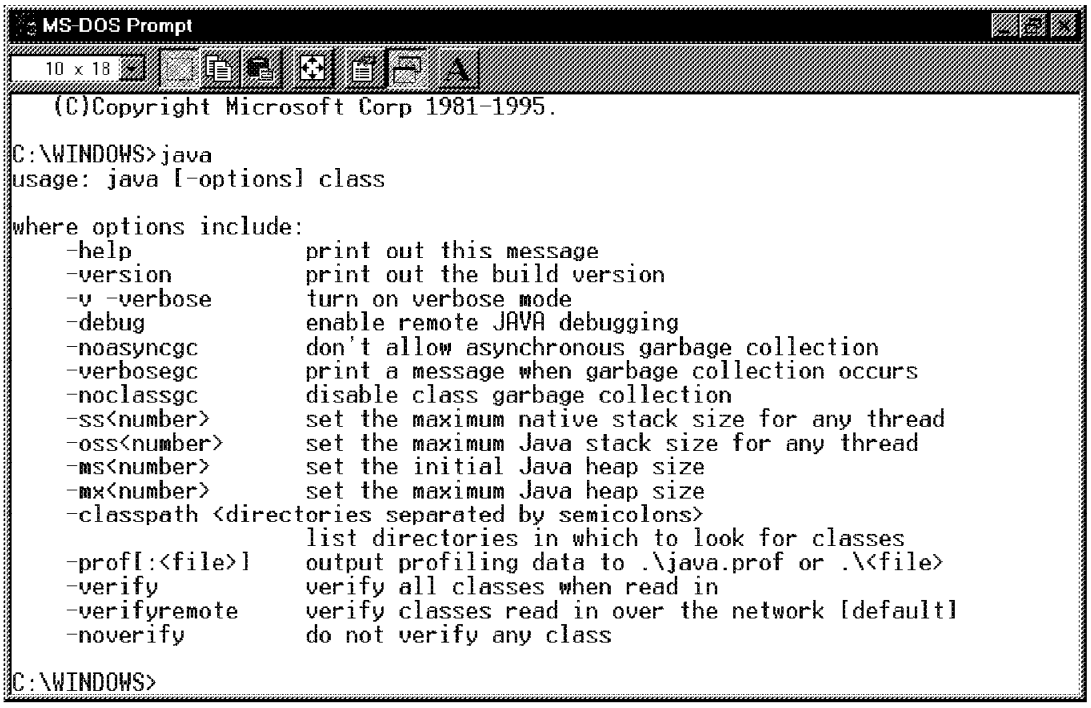


Figure 45. Testing the Path Variable for Java

You have now completed setting up the **PATH** environment variable.

3.4.1.2 Setting Up the CLASSPATH Variable

Now you must change the **CLASSPATH** environment variable to locate the AS/400 Toolbox for Java classes.

The JDK does not require you to change the **CLASSPATH** environment variable as it can find the core Java classes as long as they are in the **CLASSES.ZIP** file located in the **LIB** sub-directory and the **LIB** sub-directory is located in the same directory as the **BIN** sub-directory indicated in the **PATH** statement. This is because the JDK automatically appends the following CLASSPATH data to whatever you have explicitly set in your **AUTOEXEC.BAT** file:

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

where [bin] is substituted by the absolute path to the **JDK1.1.4\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 one you develop or the **AS/400 Toolbox for Java**.

The **AS/400 Toolbox for Java** classes are located on the AS/400 IFS in the following directory structure:

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

Important Note

The Java language is **extremely** case sensitive! Make sure you type all the path names (directory names) exactly as shown in the examples.

To use the **AS/400 Toolbox for Java** classes, we must tell the JVM where to look for these classes. We do this by means of a **CLASSPATH** directive that we add to the **AUTOEXEC.BAT** file. To change the **AUTOEXEC.BAT** file, you need to do the following steps:

- Start a text editor such as WordPad or Notepad.
- From the File menu, open "c:\autoexec.bat".
- Look at the **SET CLASSPATH=** statement. Notice that the **CLASSPATH** statement is a series of directories separated by semi-colons (;). The JVM looks for Java classes in the **CLASSPATH** directories in order, from left to right.
- If a **SET CLASSPATH=** statement does not exist, insert a new one.
- Put the **AS/400 Toolbox for Java** directory at the end of the **SET CLASSPATH=** statement.
- Choose Save from the File menu to update your **AUTOEXEC.BAT** file.
- Exit from the text editor.

For example, on the following SET statements, we have 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 IFS. This assumes that you have mapped a network drive "S:" to your AS/400 system. We then set the **CLASSPATH** environment variable to instruct the JVM that the AS/400 Toolbox for Java classes are contained in the **jt400.zip** file located in the **lib** sub-directory. We also indicate how to find utilities such as the **AS400ToolboxInstaller.class** file located in the **utilities** sub directory. 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 46 on page 66 shows a typical AUTOEXEC.BAT file edited with the Notepad utility.

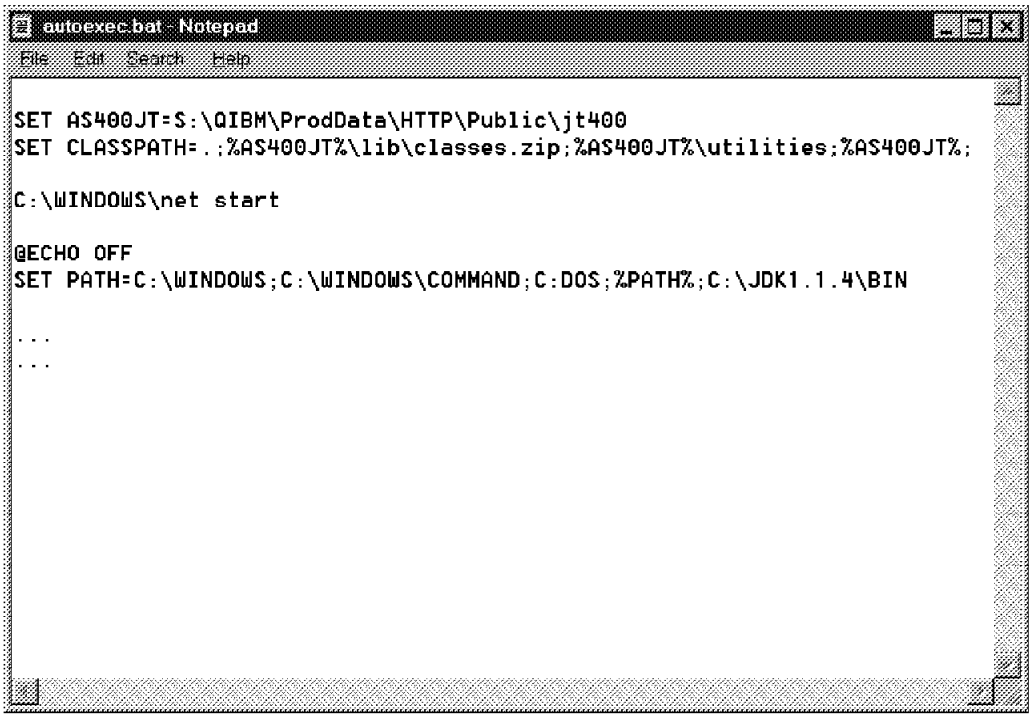


Figure 46. Setting up the CLASSPATH Variable for Java

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 native Java support provided in the JDK. There is no such thing as an **AUTOEXEC.BAT** file with a **PATH** environment variable. Similar to any other standard JDKs, the AS/400 native JDK does not require you to change the **CLASSPATH** environment variable as it can find the core Java classes as long as they are stored in the **java.zip** and **sun.zip** files located in the **lib** sub-directory and that the **lib** sub-directory is in the same directory as the **bin** sub-directory.

This is because the AS/400 Java Virtual Machine (JVM) automatically searches all the classes it needs to load in the **lib** sub-directory located in the **jdk** directory on the AS/400 IFS.

Therefore, if you do not modify anything in the **jdk** directory and associated sub-directories (as shipped with the 5769-JV1 Licensed Program), the Java executables find the classes.

You need to do some setup if you want to load additional class libraries such as one you develop yourself or the **AS/400 Toolbox for Java**.

The AS/400 Toolbox for Java classes are stored on the AS/400 IFS in two files named **jt400.zip** and **jt400.jar**. These two files are located in the following 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 the examples that follow, 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 the symbolic link **AS400JT** is shown in Figure 47.

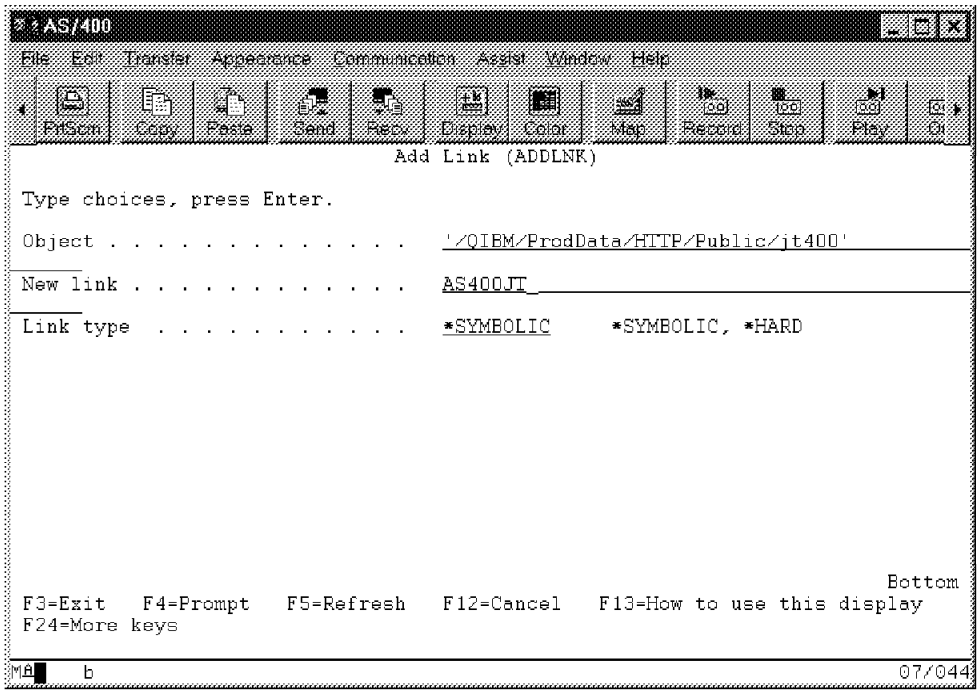


Figure 47. Adding a Symbolic Link

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 Setup on Every Java CL Command

You can specify the proper path on the **CLASSPATH** parameter on the CRTJVAPGM and RUNJVA (or the JAVA) commands as shown in Figure 48 on page 68.

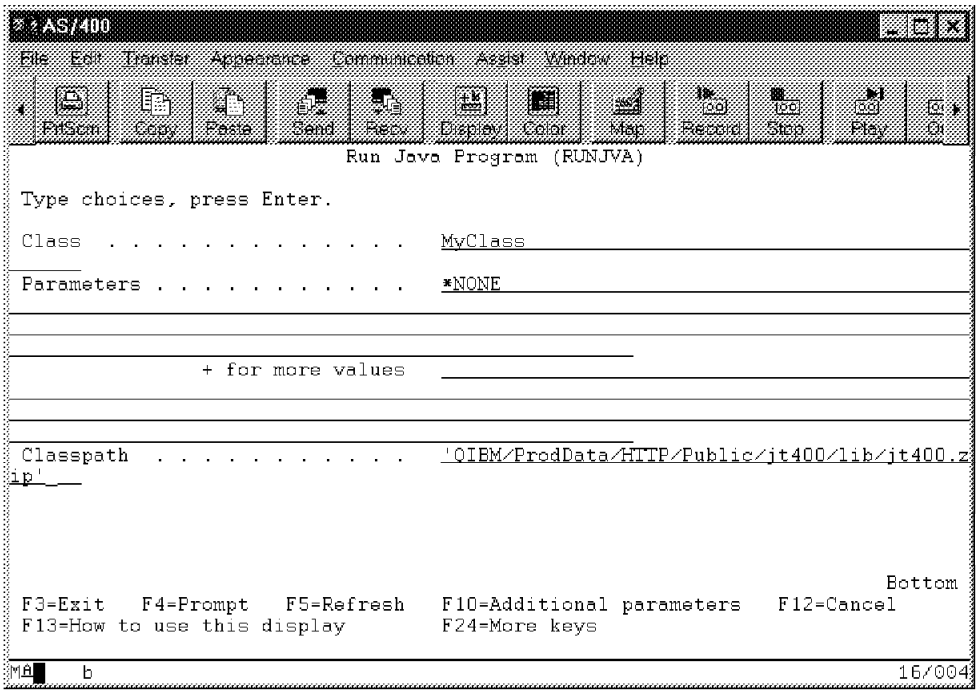


Figure 48. Run Java Program Display

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 can be used to set the CLASSPATH variable. Two CL commands are available:

- ADDENVVAR - to add the environment variable for the session
- WRKENVVAR - 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 you sign on. For example, the following 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), the classpath parameter can be set to ***ENVVAR**. This is the default setting and 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 that this directive is a succession of directory names separated by a colon (:).

Several different techniques may be used to set the CLASSPATH for CL commands. The CLASSPATH used 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 QShell

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 described previously for CL commands.
- Use the export directive to set up a CLASSPATH dynamically.
- 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.

Several different techniques may be used to set the CLASSPATH for an AS/400 system. The CLASSPATH used depends on which technique applies to the user. The order of precedence for setting the CLASSPATH for the QShell environment is:

1. -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 Dynamically in QShell

To set the CLASSPATH variable after starting QShell, enter:

```
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 that this directive is a succession of directory names separated by a colon (:). The QShell interpreter searches the directories in the order 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 Dynamically Using -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

When using the QShell Interpreter, you can control the CLASSPATH information by individual user or system wide. In this section, we show 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's home directory. Do this by typing the following command on an AS/400 Command Entry line:

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

where "myusrprf" is the user profile for the user.

You may also use the PC-like alias of the **md** command or the **mkdir** command.

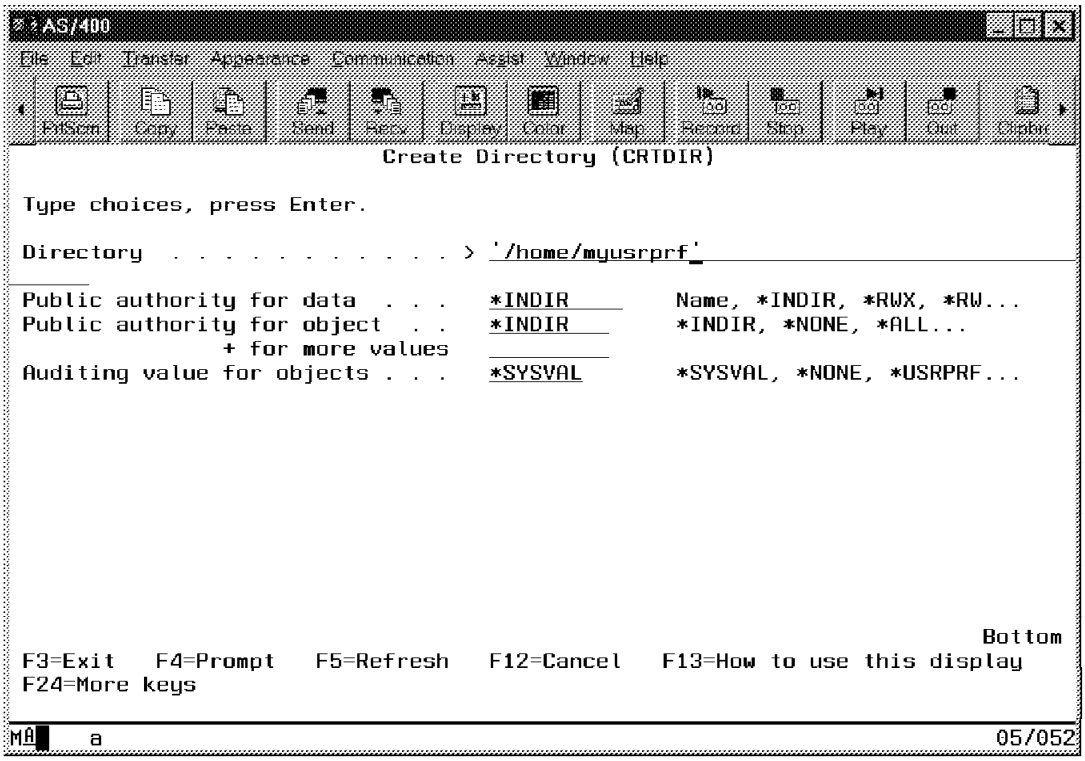


Figure 49. Creating an IFS Directory on AS/400 System

This creates a sub-directory named "myusrprf" in the **home** directory. You then need to create the **.profile** file for that user profile. Do this by entering the following command on an AS/400 Command Entry line:

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

The **EDTF** command invokes a stream file editor provided in the **QUSRTOOL** library that 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 the following 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 that this directive is a succession of directory names separated by a colon (:). The QShell interpreter searches the directories in the order 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's home directory, as created before. This statement allows you to find the AS400 Toolbox for Java classes and your own Java classes as long as your classes are stored in your home directory. Figure 50 on page 71 shows the **EDTF** window.

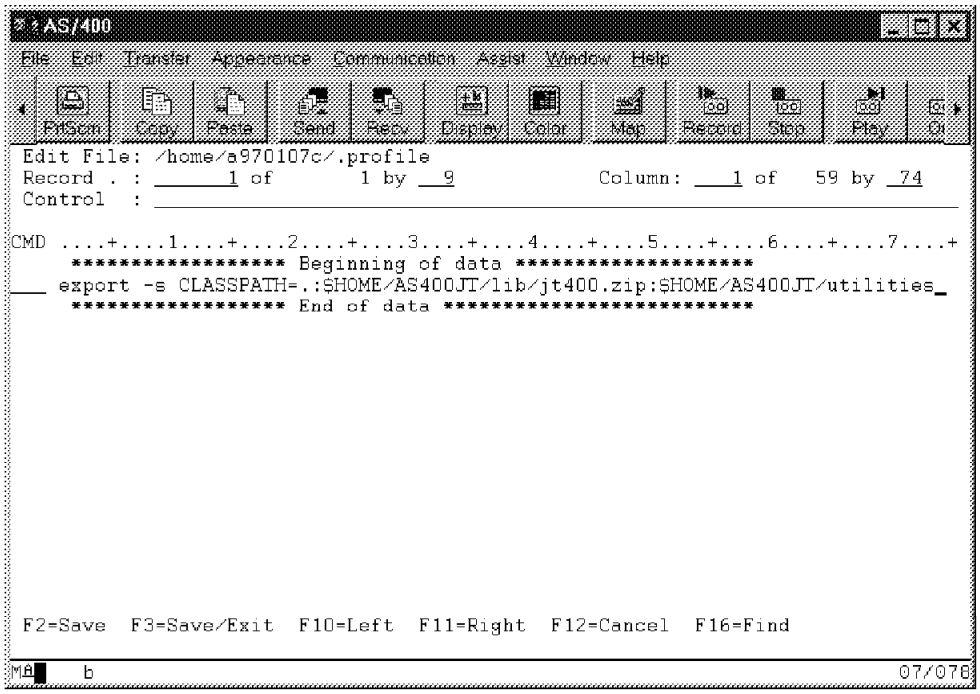


Figure 50. Using EDTF to Create .profile File

This setup allows you to create an identical **.profile** file in every user's 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's directory. We do this by signing on as the user and executing the **ADDLNK** command. We then create a link named **AS400JT** that points to the Toolbox 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 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 you used in creating the individual files.

```
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 to 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's 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 do it exactly as explained before by using the ADDLNK OS/400 IFS command.

3.4.4.5 Testing the AS/400 Java Environment

Important information

We provide an example Java program named **checkToolbox**. You can find the complete source listing of this program in Appendix C, “Java Source Code Samples” on page 209. This program is also available for downloading from our Internet site. Please refer to Appendix A, “Example Programs” on page 203 for download instructions.

To successfully compile the **checkToolbox** program as shown here, you must:

- Set up the AS/400 Java environment correctly (in our case, we use the QShell environment).
- Place the source code (checkToolbox.java) in your home directory.

Now you can test that the environment you just set up is working properly. First, compile the **checkToolbox** Java program. This program uses the **AS400ToolboxInstaller** class and provides the functions required to install (download) the AS/400 Toolbox for Java classes on your PC and check if the downloaded version needs to be updated in case PTFs have been applied to the AS/400 system. To compile the Java program, start the QShell Interpreter by typing **qsh** on any AS/400 command entry line. This opens the **QSH Command Entry** display. On the command entry line, enter the following QShell command:

```
javac checkToolbox.java
```

This is the standard Java command to compile a Java source file (**.java** file) into bytecodes (**.class** file). If your environment is set up as required and if the Java source file (**.java** file) is located in your home directory, the program should compile successfully and the corresponding **.class** file should be created in your home directory. You can check your environment by typing **export** on the QShell command entry line. Figure 51 on page 73 shows the javac command on the QShell command entry line and the results of the export command.

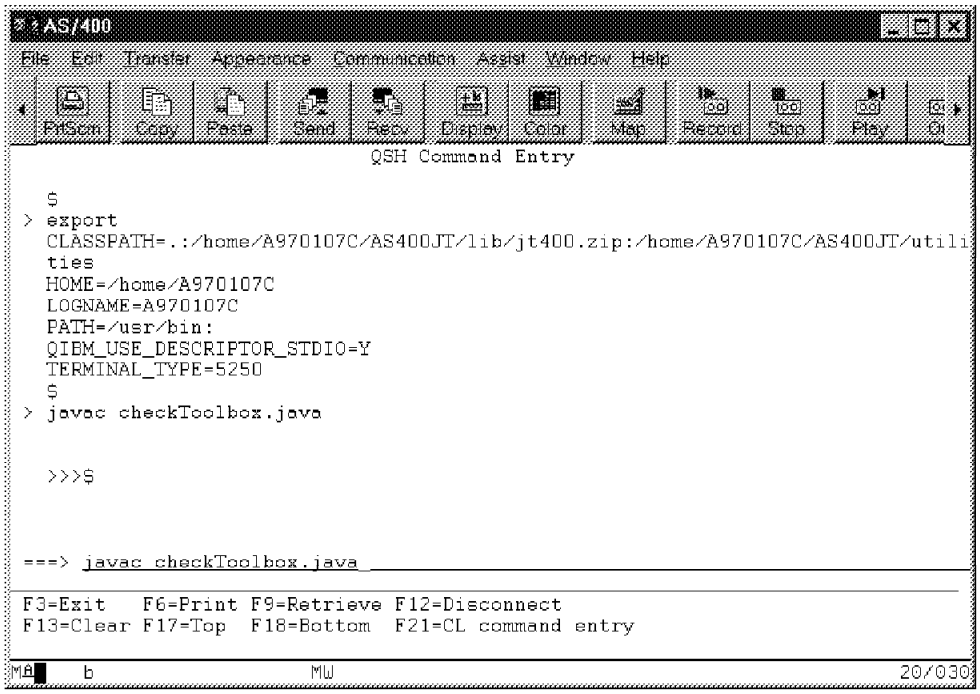


Figure 51. Compiling a Java Program in QShell Environment

The >>>**$** you see near the bottom of the output area is a "message" that is returned by the Java virtual machine to indicate completion of the previous request. This message is NLS enabled and requires no translation.

Now you can test the **checkToolbox** program by typing the following command on the QShell command entry line:

```
java checkToolbox
```

This is the standard Java command to run a program. If the program was compiled successfully and if your **CLASSPATH** is set up properly, the program responds by "printing" help information on the display regarding the required parameters it is expecting as shown in Figure 52 on page 74.

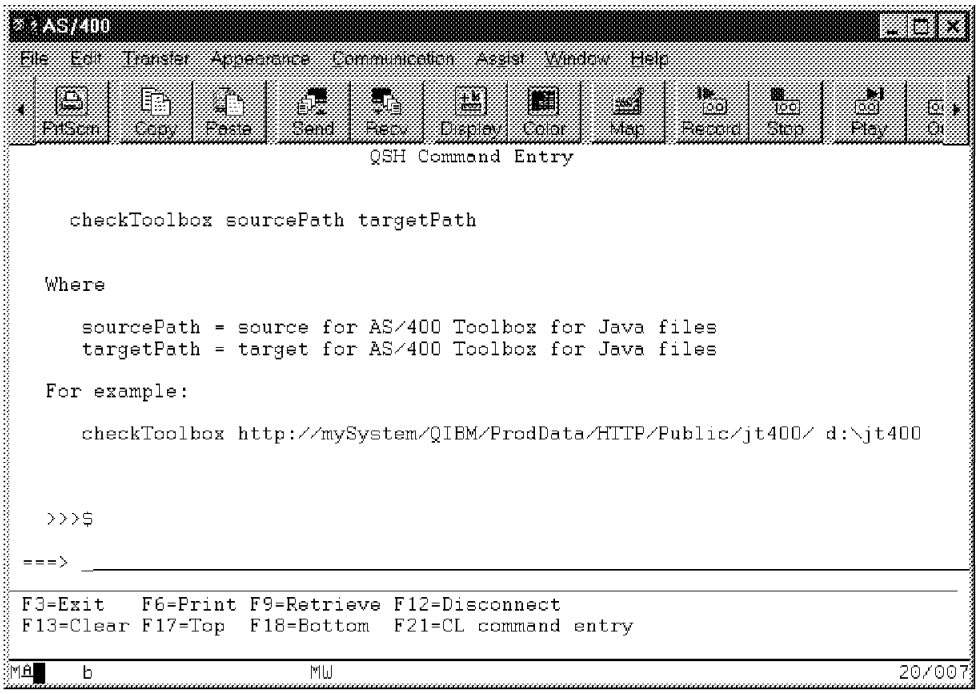


Figure 52. Running the checkToolbox Program from QShell

3.4.4.6 Setting the System Value QUTCOFFSET

The system value **QUTCOFFSET** 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 you need to subtract from local to obtain the UTC.

The JVM uses this value to keep track internally of the time in your system's locale. Java has the ability to track time in other locations using locales. The following example (the JVM uses the UTC) to derive 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 **QUTCOFFSET**, see Chapter 2 in *OS/400 Work Management*, SC41-5306-01.

In the next section, we show how to use the checkToolbox program to download the AS/400 Toolbox for Java classes to your Personal Computer.

3.4.5 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 Application Programming Interfaces (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
- IFS 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 5763-JC1 is an AS/400 licensed program and is shipped to you on the OS/400 Licensed Programs media (CD-ROMs). You install this licensed program on the AS/400 system using standard installation procedures as described in Section 3.2, “Manually Installing Java Support on AS/400 System” on page 41.

You can choose to use these classes directly from the AS/400 IFS. All you need to do is to map a network drive on your PC to the AS/400 IFS and direct the Java Virtual Machine on your PC to this drive with an appropriate **CLASSPATH** directive as described in Section 3.4.1.2, “Setting Up the CLASSPATH Variable” on page 64. This is the easiest way to use the 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 licensed program, 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 IFS ensures that you automatically get all the changes after they are applied to the AS/400 system.

However, for **performance** reasons, you may prefer to copy the AS/400 Toolbox for Java classes on your PC hard disk drive. As part of the AS/400 Toolbox for Java, we provide a Java class, **AS400ToolboxInstaller**, which is designed to handle installing, updating, and un-installing the AS/400 Toolbox for Java classes on your workstation. This is not a stand-alone application and it must be included in a user-written program. This class provides several methods, some of them being:

- **isInstalled()** to check if the AS/400 Toolbox for Java is currently installed on your workstation.
- **isUpdateNeeded()** to check if PTFs were applied on the AS/400 system that require your workstation to be updated.
- **install()** to install or update the AS/400 Toolbox for Java on your hard disk.

- **unInstall()** to remove the AS/400 Toolbox for Java package from your hard disk drive.

checkToolbox

Please refer to Section 3.4.4.5, “Testing the AS/400 Java Environment” on page 72 for details on how to install and compile the **checkToolbox** program.

We provide an example Java program named checkToolbox that uses the AS400ToolboxInstaller. Use this program to install the AS/400 Toolbox for Java classes on your PC. We have already set up the required **CLASSPATH** environment variables on the PC to allow the JVM on the PC to locate all the classes this program uses such as the AS400ToolboxInstaller class. First, you need to start an MS-DOS session on your PC. Make sure you have a network drive mapped to your AS/400 system. In this section, we assume that the S: drive is mapped to your AS/400 system.

At the DOS prompt, switch to your network drive. Enter S: and then press the Enter key. Then set your AS/400 home directory as the current directory on your PC. Enter the following command (where myusrprf is *your* user profile):

```
CD home\myusrprf
```

at the DOS prompt and press the **Enter** key. Then test if you can run the checkToolbox program. Type the following command:

```
java checkToolbox
```

at the DOS prompt and press the **Enter** key. You see the same messages as when you tried to run the program on the AS/400 system as shown in Figure 53 (notice that in this example we have set our user profile to **A970107C**).

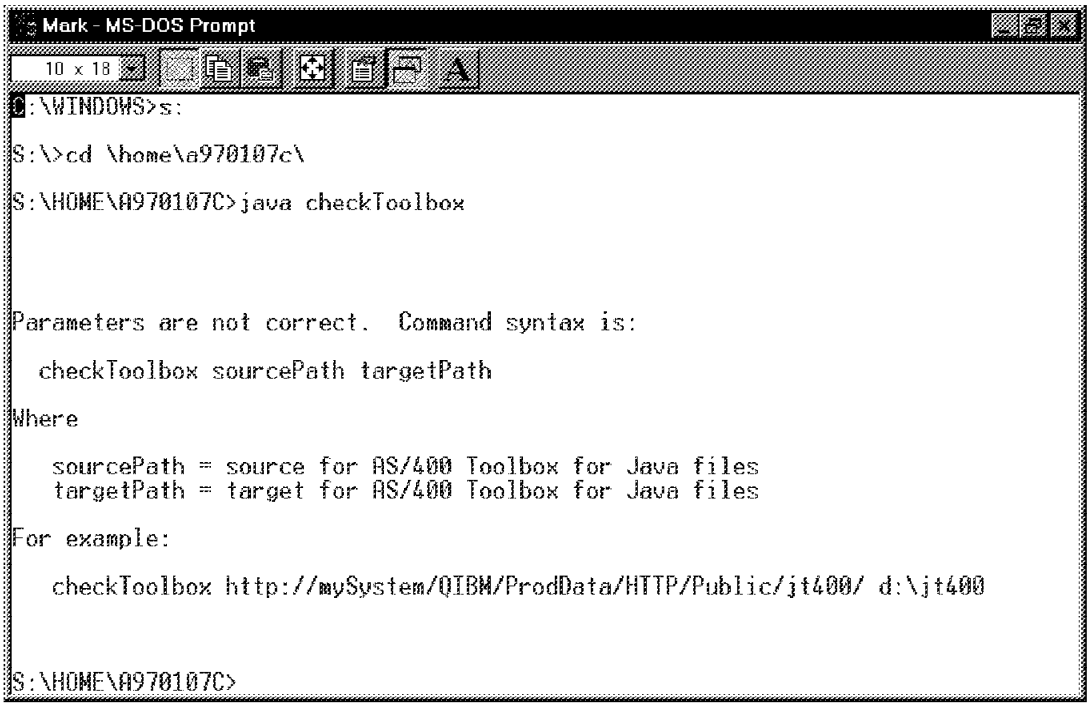


Figure 53. Testing the checkToolbox Program

Note: We compiled this program on the AS/400 system and we ran it on your PC without making any changes to the Java bytecodes! This same program can also run unchanged on any other Java enabled platform.

Now you need to run the program to actually download the AS/400 Toolbox for Java classes on your PC.

This program makes use of the **AS/400 HTTP server**, which must be running, and also in your HTTP configuration file, there must be a pass statement that allows us access to the AS/400 Toolbox for Java classes (this can be added using the **WRKHTTPCFG** CL command):

```
Pass /QIBM/ProdData/HTTP/Public/jt400/*
```

On your PC at the DOS prompt, type the following command:

```
java checkToolbox http://as400sys/QIBM/ProdData/HTTP/Public/jt400/ c:\jt400
```

Make sure you type in the slash (/) at the end of the source path parameter right after jt400. This is required as file names are appended to the path name by the program. If you forget to type in the slash (/), you receive a "File not found" exception error message.

The program checks if the AS/400 Toolbox classes are installed on your system. If the classes are not installed (or if they are out of date), the program asks if you want to install them now. Answer Y for Yes to the message to start downloading the classes to your PC. In a few seconds, you receive a message telling you that the installation was successful as shown in Figure 54.

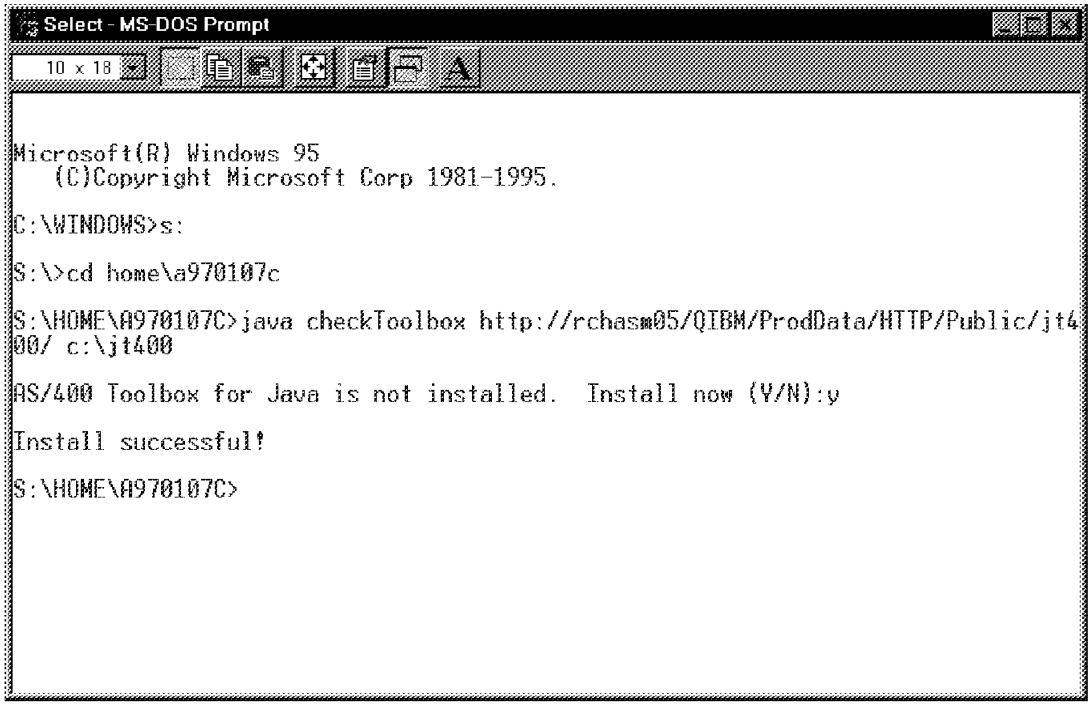


Figure 54. Installing AS/400 Toolbox for Java on Your PC

Now you have one final thing to do: modify the **AUTOEXEC.BAT** file on your PC to change the **CLASSPATH** environment variable so that the JVM on your PC loads the AS/400 Toolbox for Java classes from your PC's hard disk instead of loading them from the AS/400 IFS.

To change the **AUTOEXEC.BAT** file, do the following steps:

- Use a PC text editor such as Notepad or Work Pad.
- On the File menu, open the **autoexec.bat** file
- Look for the **SET CLASSPATH=AS400JT** line; if there is no line, you need to add one.
- Change the line to set our private environment variable to **c:\jt400**.
- Save your changes.
- 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 (Remote AWT) is a set of Java classes that use the Remote Method Invocation (RMI) feature of the Java Development Kit (JDK) to handle graphical user interface (GUI) operations on a server that 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 makes use of the Java AWT.

Typically, the only GUI operations that are performed on the server side of a client/server Java application are those related to application installation and configuration; that is, 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

You can use the Remote AWT support by running it from the AS/400 IFS or the PC hard drive. To run Remote AWT from the AS/400 IFS, all you need to do is to map a network drive on your workstation and set up the proper **CLASSPATH** environment variable so that the Java Virtual Machine running on your workstation can find the Remote AWT classes.

The Remote AWT classes are contained in the **rawt_PC_classes.zip** file located in the **/QIBM/ProdData/Java400/lib/** directory in the IFS. Assuming that you have assigned drive letter **S** to your AS/400 server, this is the **CLASSPATH** statement that you use:

```
SET CLASSPATH=S:/QIBM/ProdData/Java400/lib/rawt_PC_classes.zip
```

Make sure the CLASSPATH for the Remote AWT support comes before the one needed to access standard JDK classes as specified earlier in this chapter.

Because the Remote AWT support generates heavy network traffic, you may want to install (download) the Remote AWT classes on your workstation. You can do this by first creating a directory on your workstation and then copy the Remote AWT **rawt_PC_classes.zip** file from the AS/400 IFS to your workstation. You can do this by entering the following DOS commands at the DOS prompt:

```
C:\WINDOWS>cd ..

C:\>md rawt

C:\>cd rawt

C:\rawt>s:

S:\>cd QIBM\ProdData\Java400\lib

S:\QIBM\ProdData\Java400\lib>copy s:rawt_PC_classes.zip c:
```

This process is shown in Figure 55.

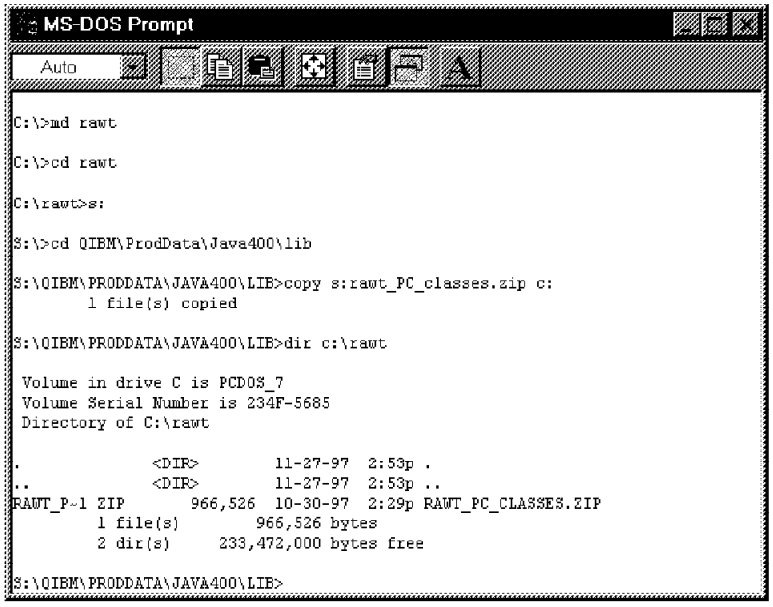


Figure 55. Downloading the Remote AWT Classes to Your Workstation

In this case, the **CLASSPATH** to find the Remote AWT classes needs to be set as follows:

```
SET CLASSPATH=C:\RAWT\rawt_PC_classes.zip
```

Now you are ready to start Remote AWT.

3.5.2 Starting Remote AWT Support On Your Workstation

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

We do this by starting a Java application named **java.awt.Awt400ServerImpl** on your workstation. You must pass your workstation's IP address and an arbitrary port number as a parameter to this application. Use a private port number that does not conflict with any TCP/IP application.

Currently, the port numbers are divided into three ranges: the well-known ports, the registered ports, and the dynamic and private ports. The well-known ports are those from 0 through 1023; the registered ports are those from 1024 through 49151 and the dynamic and private ports are those from 49152 through 65535. You can pick up any number in that range such as 55555, for example.

Normally, you want to start the Remote AWT daemon and have it run in the background. Do this by typing the following command:

```
start /m java java.awt.Awt400ServerImpl 192.168.21.7:55555
```

where **192.168.21.7** is the IP address of your workstation and **55555** is an arbitrary port number within the private port range. 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, do the following steps:

- Start an editor such as Word Pad or Notepad.
- Type in the **SET CLASSPATH** and the **start /m** statements previously shown.
- Save your file in the folder of your choice and give it a meaningful name such as **rmtawt.bat**.
- Exit from the editor.

You now can start the Remote AWT daemon on your workstation by typing **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. The example below shows the required SET CLASSPATH and start commands:

```
SET CLASSPATH=C:\RAWT\rawt_PC_classes.zip
start /m java java.awt.Awt400ServerImpl 192.168.21.7:55555
```

Note: If you add just these two lines, then when the Remote AWT program ends on the AS/400, your workstation daemon will end as well. To circumvent this, you may wish to place you listener daemon in a continuous loop, which will restart it each time after it is ended. Change you code so it looks like this:

```
SET CLASSPATH=C:\RAWT\rawt_PC_classes.zip
:loop
start /m java java.awt.Awt400ServerImpl 192.168.21.7:55555
goto loop
```

When you start the Remote AWT support on your workstation, you are told that the support is ready when the message shown on the next picture briefly pops up.

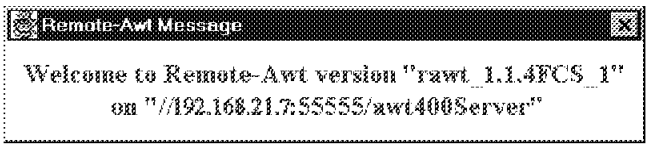


Figure 56. Remote AWT Startup Message

3.5.3 Starting Remote AWT Support on the AS/400 System

When you want to run a Java application that uses the AWT APIs and have the AS/400 application's GUI displayed on your workstation, you must set up the Java properties required to run the Remote AWT support. There are two properties that need to be set, **awt.toolkit** and **RmtAwtServer**. The **awt.toolkit** property must be set with the value **java.awt.CToolkit** and the **RmtAwtServer** property must be set to match the IP address and port number of the workstation where you want the AS/400 application GUI to be routed.

If you use the **java** tool from the QShell Interpreter to run your application, you must set the properties with the -D parameter on the **java** command as shown in Figure 57.

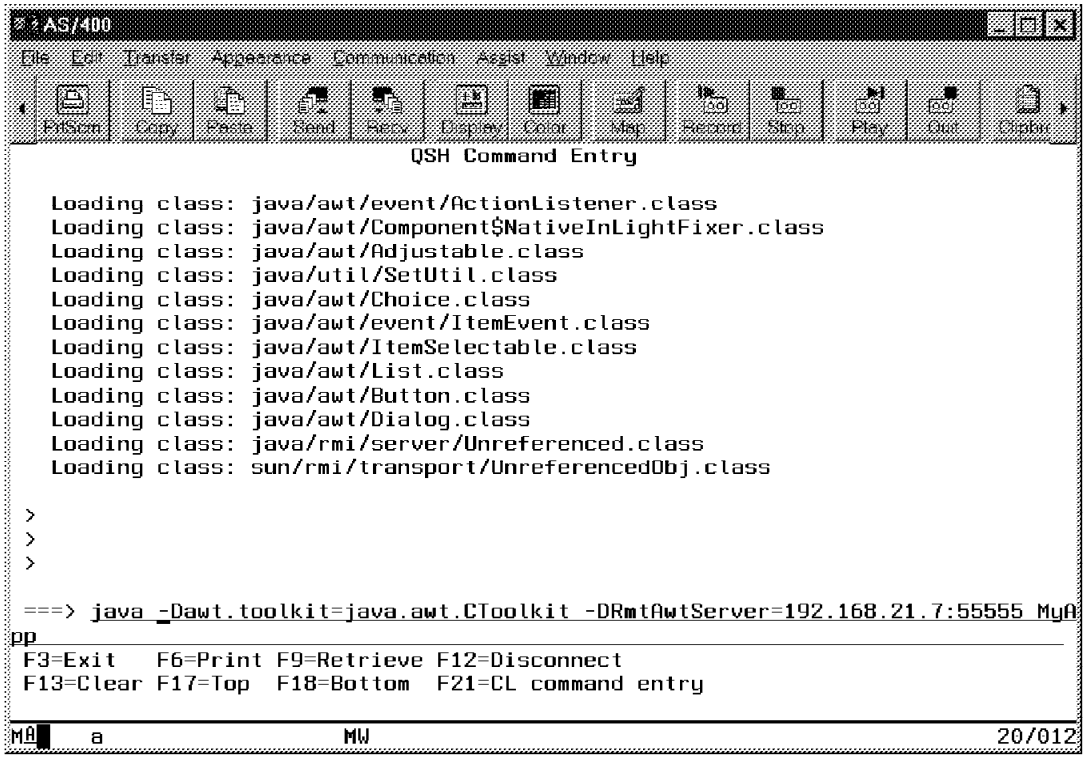


Figure 57. Starting a Remote AWT Java Program on AS/400 System

You may use your workstation's host name instead of the IP address. If you intend to do so, make sure that the host name is known to the AS/400 system, or the workstation name is on the DNS server known to the AS/400 system. (Test this by using **ping 'workstationname'** on a command line).

You can use **Work with TCP/IP host table entries** (option 10) of the *Configure TCP/IP (CFGTCP*) menu or the ADDTCPHTE command to add an entry for your workstation to the AS/400 TCP/IP host table shown in the following picture.

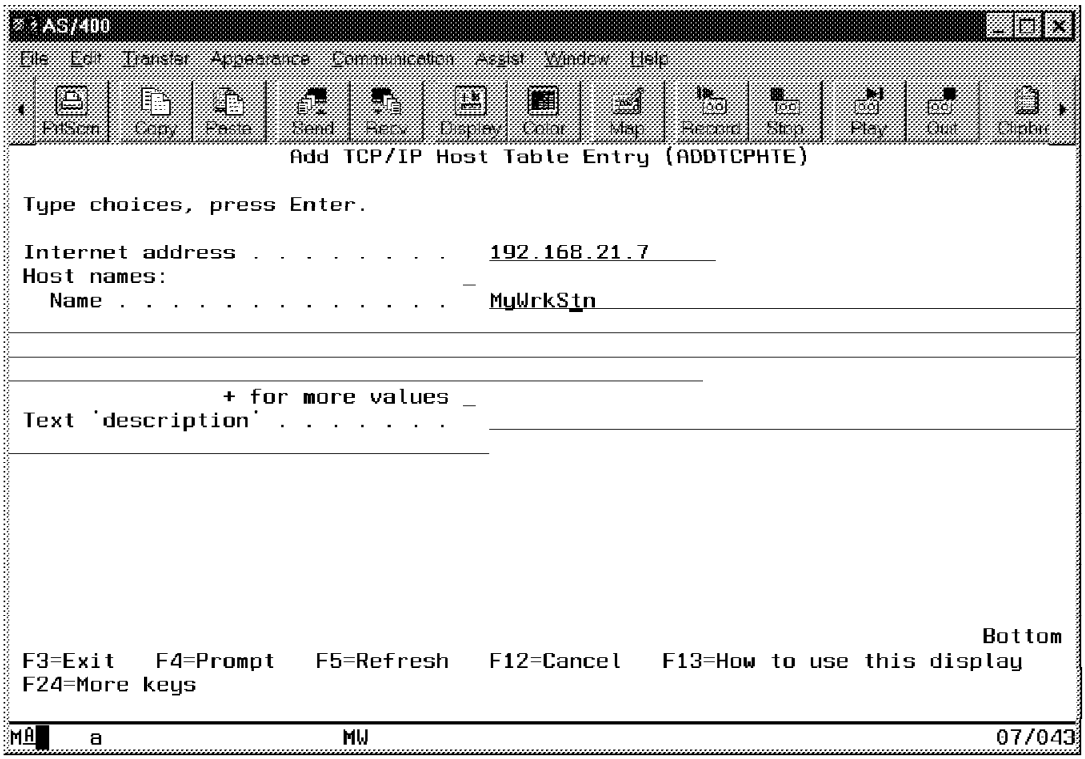


Figure 58. Putting Your Workstation Name in AS/400 Host Entry Table

If you are using a Domain Name Server (DNS) on your network, make sure that your workstation host name in the DNS matches the name in the Windows 95 configuration. If the name specified in the DNS does not match your real workstation host name as configured in Windows 95 TCP/IP properties and if you did not add an entry to the AS/400 host table, the AS/400 system tries to use the DNS to locate your workstation and the DNS provides invalid information to the AS/400 system.

You may also choose to use the OS/400 RUNJVA command (or JAVA) to start your application. If you plan to use this method, set up the Java properties as follows:

- Type **RUNJVA** or **JAVA** on any OS/400 command entry line.
- Press F4 to display the command prompt.
- Enter the name of the Java application (class name) you want to run.
- Press F10=Additional parameters for more prompting.
- Press Page Down (or Scroll Up) to go to the next page of the prompt.
- On the first line of Properties (PROP), type **awt.toolkit** for the name.
- Type **java.awt.CToolkit** on the property value line.
- Type **+** for more properties.
- Type **RmtAwtServer** as the property name for the second property.
- Type the TCP/IP address and port number of the Remote AWT workstation.
- Press the Enter key to start your application.

Figure 59 on page 83 shows the RUNJVA command prompt.

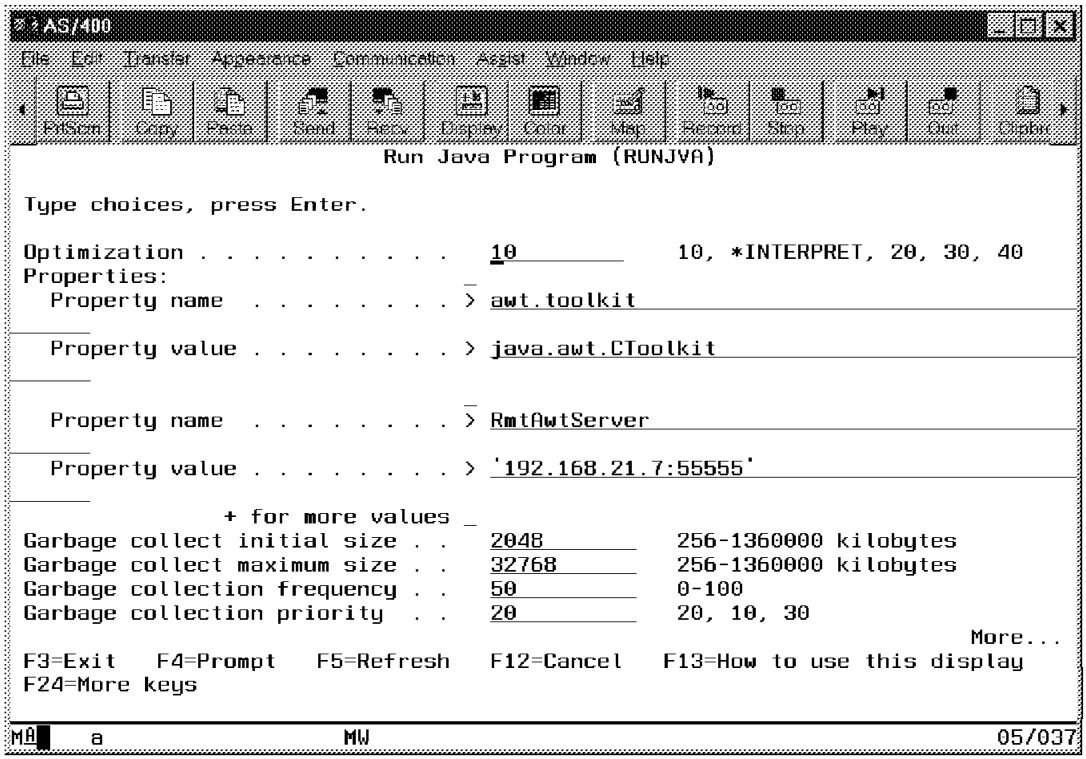


Figure 59. Using the Run Java Program (RUNJVA) Command to Run MyApp

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. This is a simple Java application that displays a simple windows panel on the workstation. The source is as follows:

```
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-orientation from an RPG programmer's perspective. If you are already familiar with OO, Java, or C++, you probably do 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. A complete introduction to Java may be obtained from any of the current books on the subject available at your local book shop.

This chapter contains information about the following subjects:

- Fields as variables
- Procedures/subroutines as methods
- Modules as classes
- Programs as packages
- The Java language

4.1 Object-Orientation and RPG

No, this is not an oxymoron! It is quite possible to write object-oriented 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 object-oriented principles in RPG as a bridge to the new terminology associated with object-orientation.

RPG IV has a closer fit to OO than the earlier versions of RPG so we shall confine our comparison to that dialect.

4.1.1.1 Variables

Variables equate to RPG fields. There are 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 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.

4.1.1.2 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.

4.1.1.3 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.

4.1.1.4 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.

4.1.1.5 Differences between Java and RPG

Java supports a number of modifiers when declaring classes and variables that do not have direct equivalents in RPG.

- **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.

The **public** and **private** modifiers can be implemented 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:

- The ability to apply object-oriented design principles in a native environment that directly supports those constructs.
- The ability to create Internet applications in a much easier manner than CGI programming.

Java also fits extremely well with the AS/400 system because they both share similar architectural principles as shown in Figure 60.

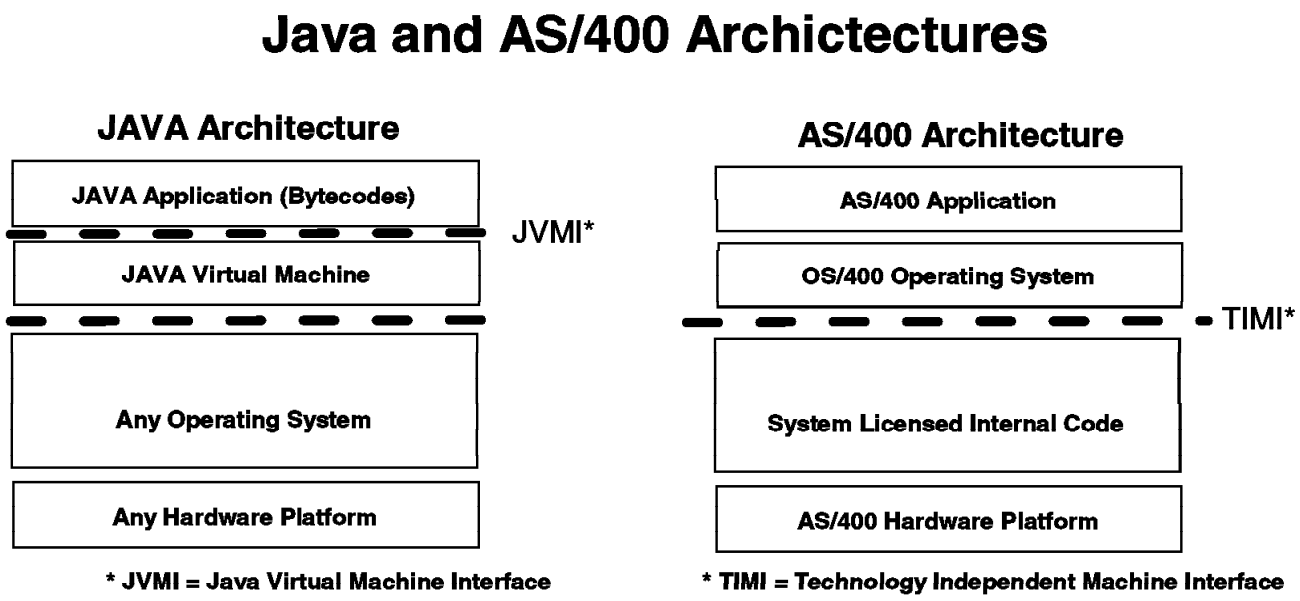


Figure 60. Java Architecture Compared to AS/400 Architecture

Figure 61 on page 88 shows a high level view of how Java is implemented on AS/400, taking advantage of the AS/400 architecture.

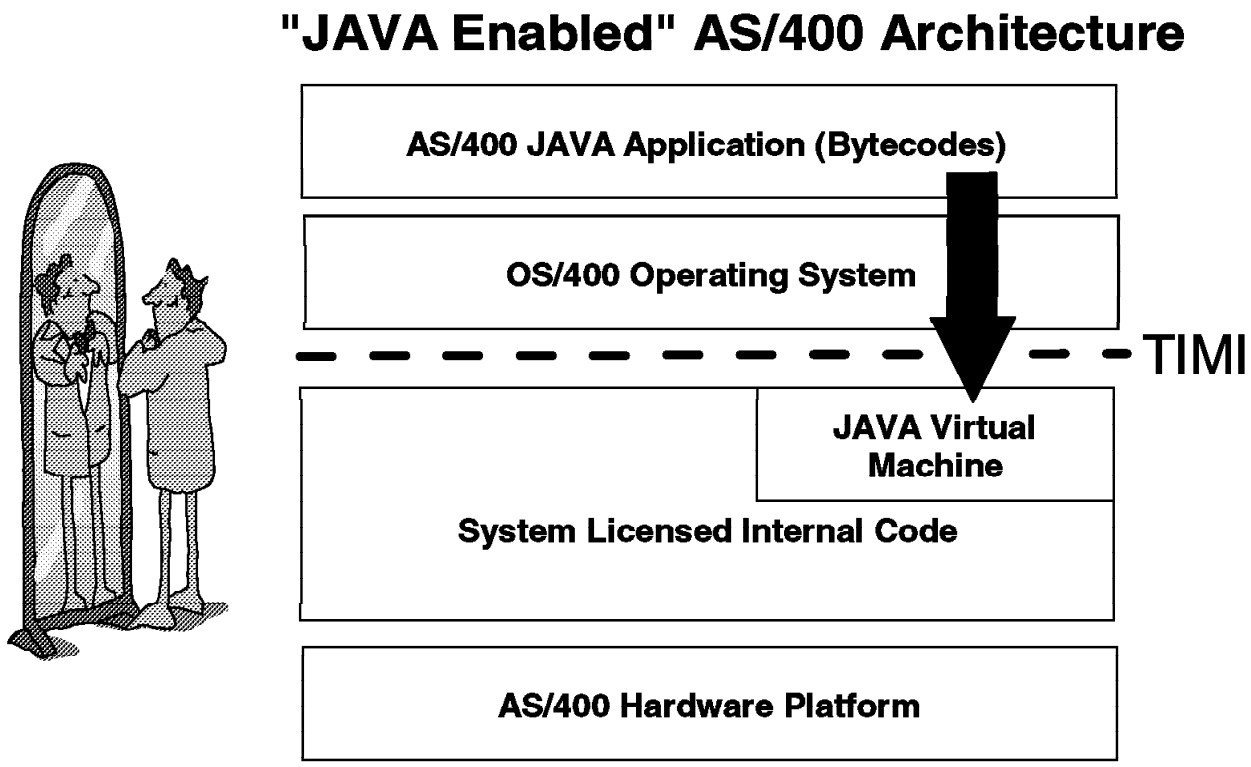


Figure 61. High Level Schematic of AS/400 Java Implementation

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 and realized that languages such as C and C++ were not suitable for this task because C and its derivatives require a compiler specific to the computer chip 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 run time.

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, 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, client/server applications can be easily written.

Java is intended to be used in a graphical environment. Graphical user interfaces 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 provided 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 be run on any hardware that implements a JVM. Byte-codes 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 object-oriented 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. 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 ending 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 incremented 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. These methods may be invoked 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 instance, 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 Java Virtual Machine 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 the code in the parent class (super class) and can extend the parent class by providing its own routines to do things 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 provided by its super class. For example, a class hierarchy representing 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**'s 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 (IFS). They may be 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 Java Virtual Machines (JVM). 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.

Compiling Java on the AS/400 system can be done in a number of ways:

- The **javac** command can be used from the QSH shell on the AS/400 system or within the SDK on a PC to compile the Java source into bytecodes and the resulting **.class** file stored in the IFS.
- The Create Java Program (CRTJVAPGM) command can be run from an AS/400 command line to create a direct execution version of the **.class** file.
- The **java** or Run Java (RUNJVA) command can be run, which automatically creates a direct execution version of the Java program.

For a more detailed coverage of this topic, see *Java for RPG Programmers*, by Phillip Coulthard and George Farr, IBM Press (1998).

Chapter 5. Overview of the Order Entry Application

In this chapter, we cover the example RPG order entry application. This application is representative of a commercial application although it does not include all 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 109, 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 141, we convert the server-based RPG application to Java so both sides of the application are written in Java.

5.1 Overview of the RPG 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 (30 000 total customers for the company). The warehouse maintains stock for the 100 000 items sold by the Company.

The following diagram illustrates the company structure (warehouse, district, and customer).

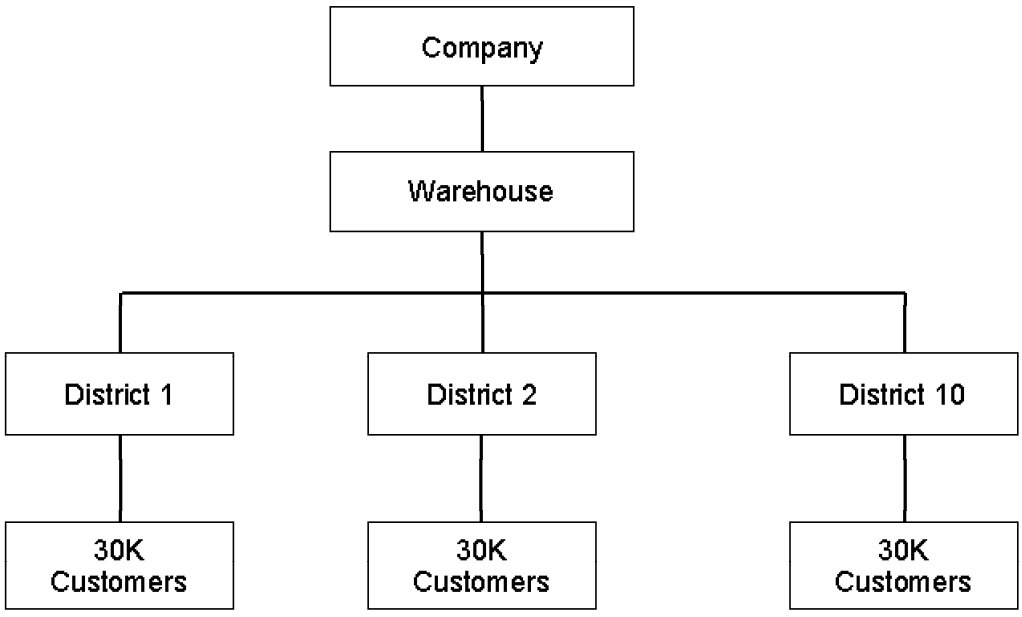


Figure 62. Company Structure

5.1.1.1 The ABC Company Database

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

The database includes tables with the following data:

- District information (next available order number, tax rate, and so on).
- Customer information (name, address, telephone number, and so on).
- Order information (date, time, shipper, and so on).
- Order line information (quantity, delivery date, and so on).
- Item information (name, price, item ID, and so on).
- Stock information (quantity in stock, warehouse ID, and so on.)

5.1.1.2 A Customer Transaction

1. Customers telephone one of the 10 district centers to place an order.
2. The district customer service representative answers the telephone, gets 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

   The customer service representative may prompt for a list of customers or a list of parts.
3. The application then:
   - Reads the customer last name, customer discount rate, and customer credit status from the Customer Table (CSTMR).
   - Reads the District Table for the next available district order number. The next available district order number is incremented by one and updated.
   - Reads the item names, item prices, and item data for each item ordered by the customer from the Item Table (ITEM).
   - Checks if the quantity of ordered items is in stock by reading the quantity in the Stock Table (STOCK).
4. When the order is accepted:
   - Inserts a new row into the Order Table to reflect the creation of the new order (ORDERS).
   - A new row is inserted into the Order Line Table to reflect each item in the order.
   - The quantity is reduced by the quantity ordered.
   - A message is written to a data queue to initiate order printing.

5.1.1.3 Application Flow

The RPG Order Entry Application consists of the following components:

Program Listings

To download the example code used in this redbook, please refer to Appendix A, “Example Programs” on page 203 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

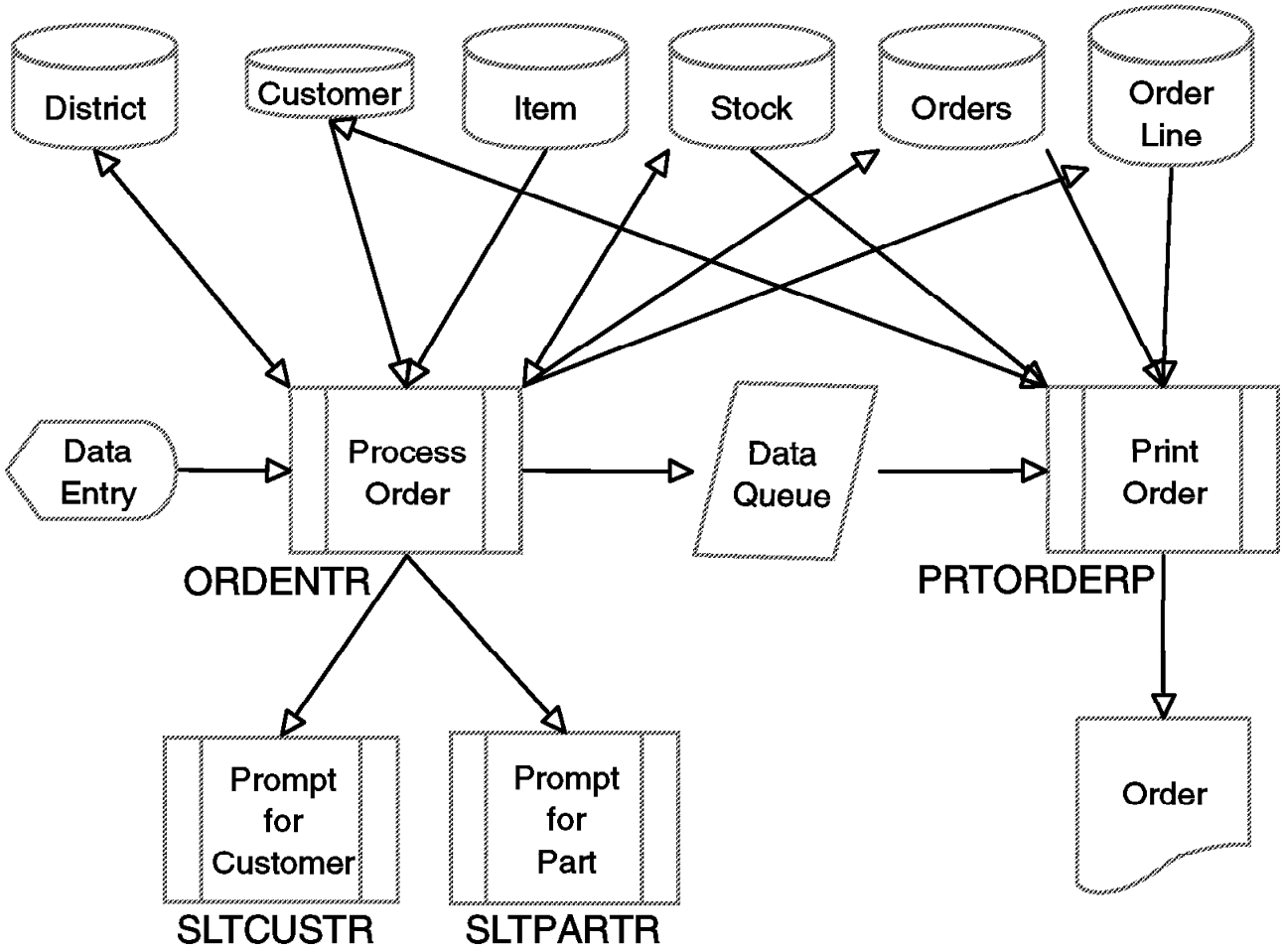


Figure 63. Application Flow

ORDENTR is the main RPG program. It is responsible for the main line processing. It calls two supporting RPG programs that are use 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.

5.1.1.4 Customer Transaction Flow

The following scenario steps through a customer transaction showing the application flow. Understanding the flow of the AS/400 application can assist in understanding the changes made to this application to support a graphical client.

5.1.1.5 Starting the Application

The application is started by calling the main program from an AS/400 command line.

CALL ORDENTR

When the order entry application is started, the following display is shown.

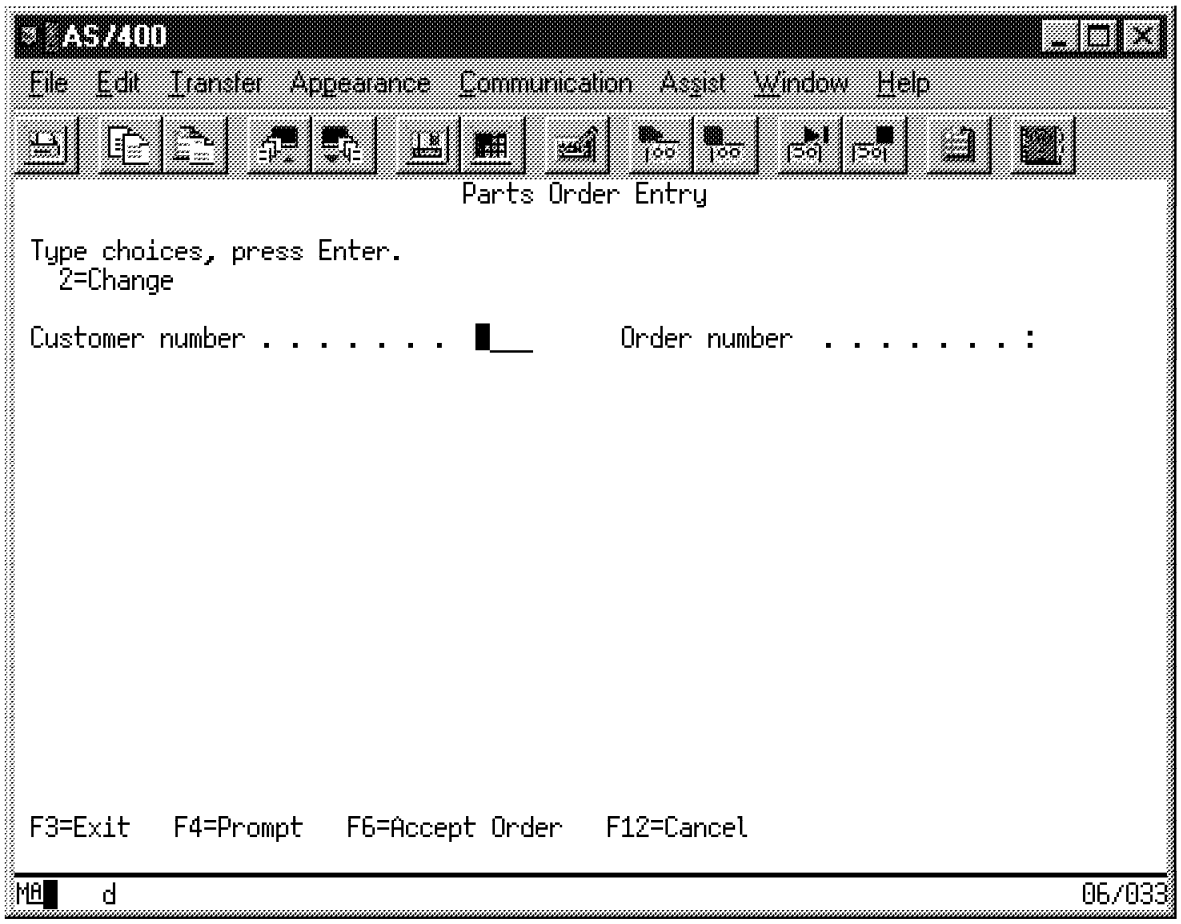


Figure 64. Parts Order Entry

The user is expected to type in a customer number and press the Enter key, but they may choose to end the program by pressing either F3 or F12.

If the user does not know the customer number, F4 may be pressed to show a window containing a list of available customers.

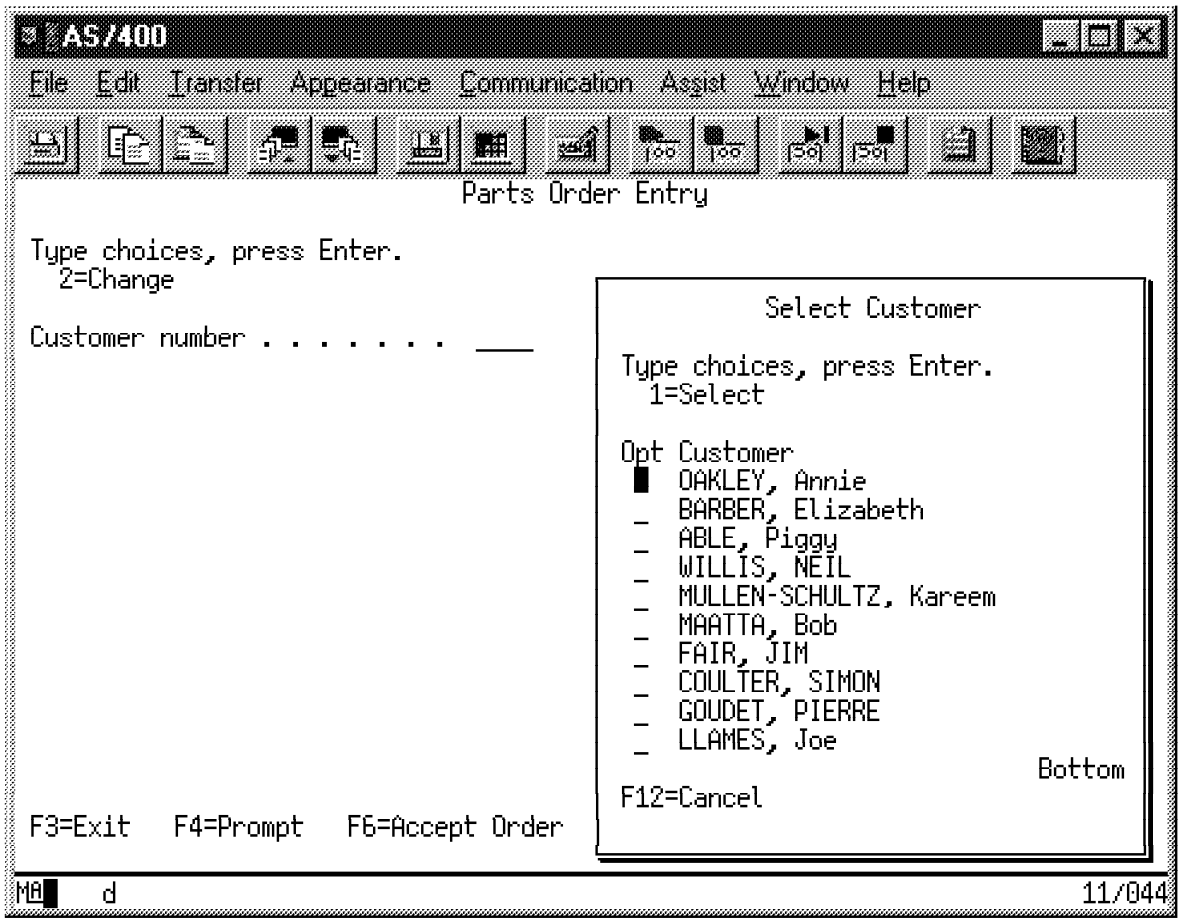


Figure 65. Select Customer

The user may press F12 to remove the window and return to the initial display. They may roll 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, they indicate their choice and the selected customer is returned to the initial display.

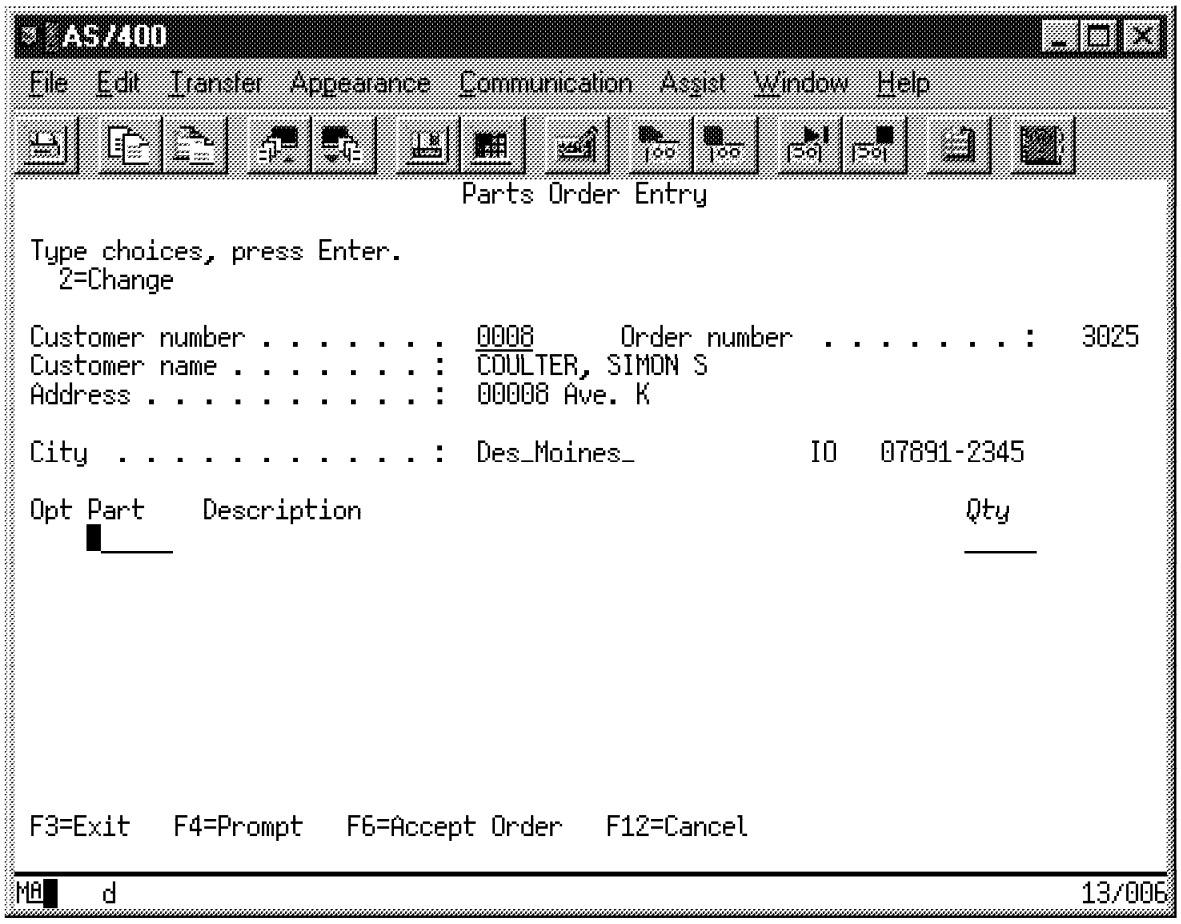


Figure 66. Parts Order Entry

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, F4 may be pressed to show a window containing a list of available parts.

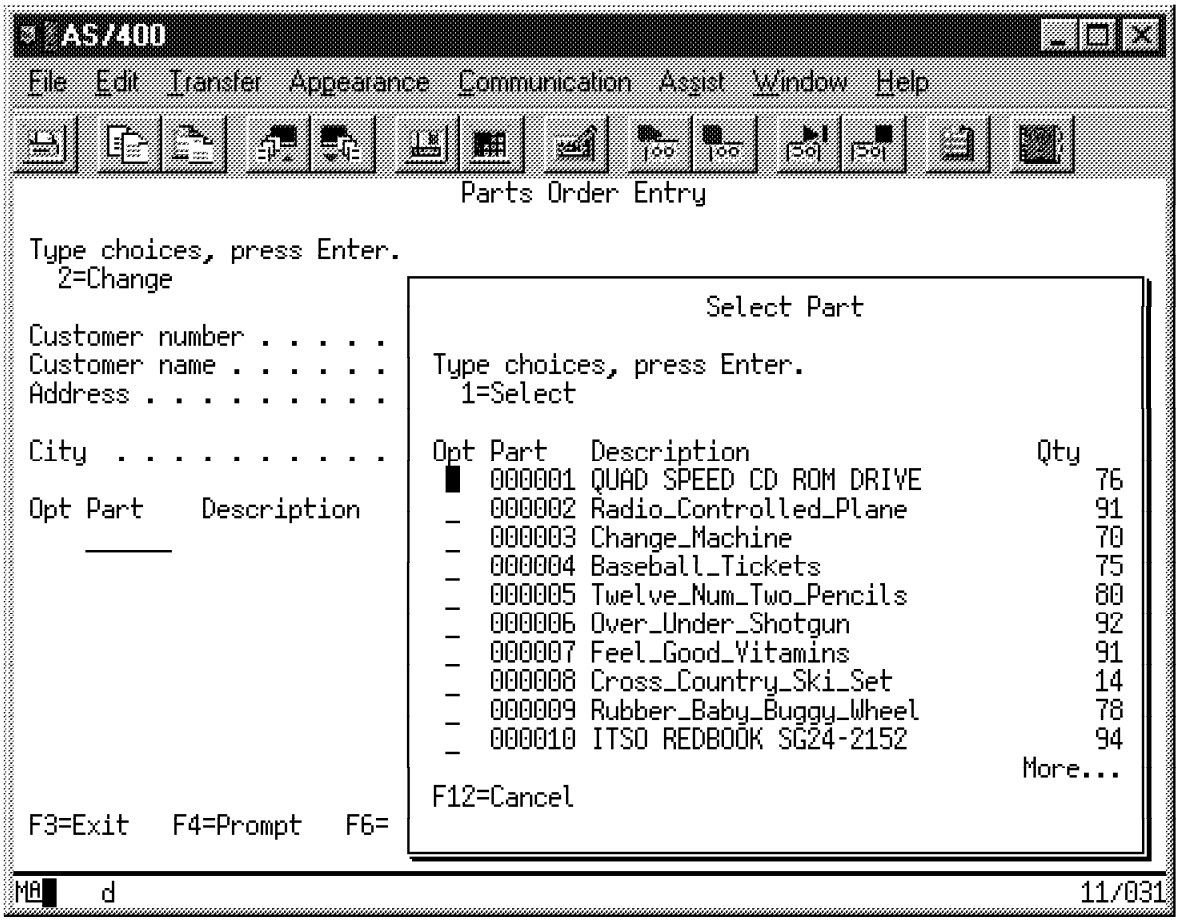


Figure 67. Select Part

The user may press F12 to remove the window and return to the initial display. They may roll 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 and the selected part is returned to the initial display.

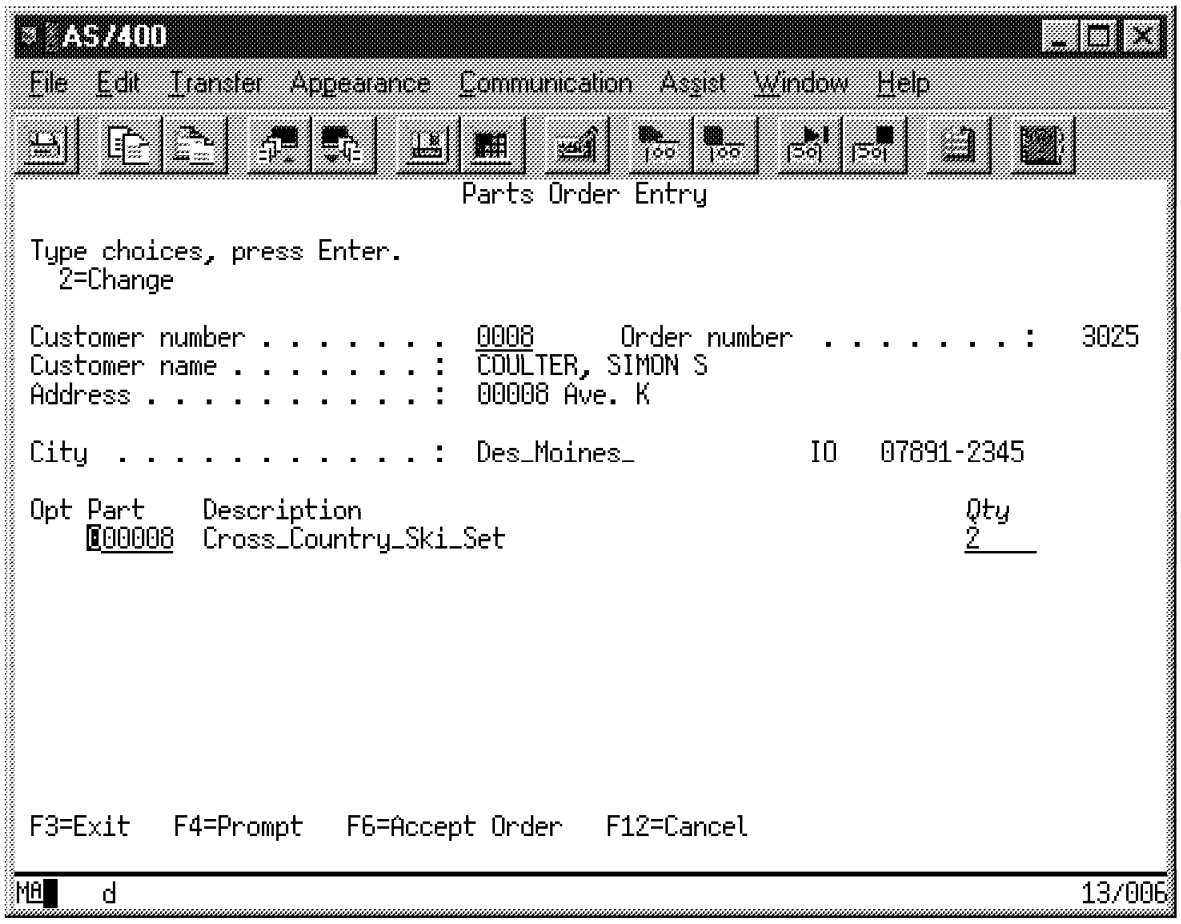


Figure 68. Parts Order Entry

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 a list section below the part entry fields.

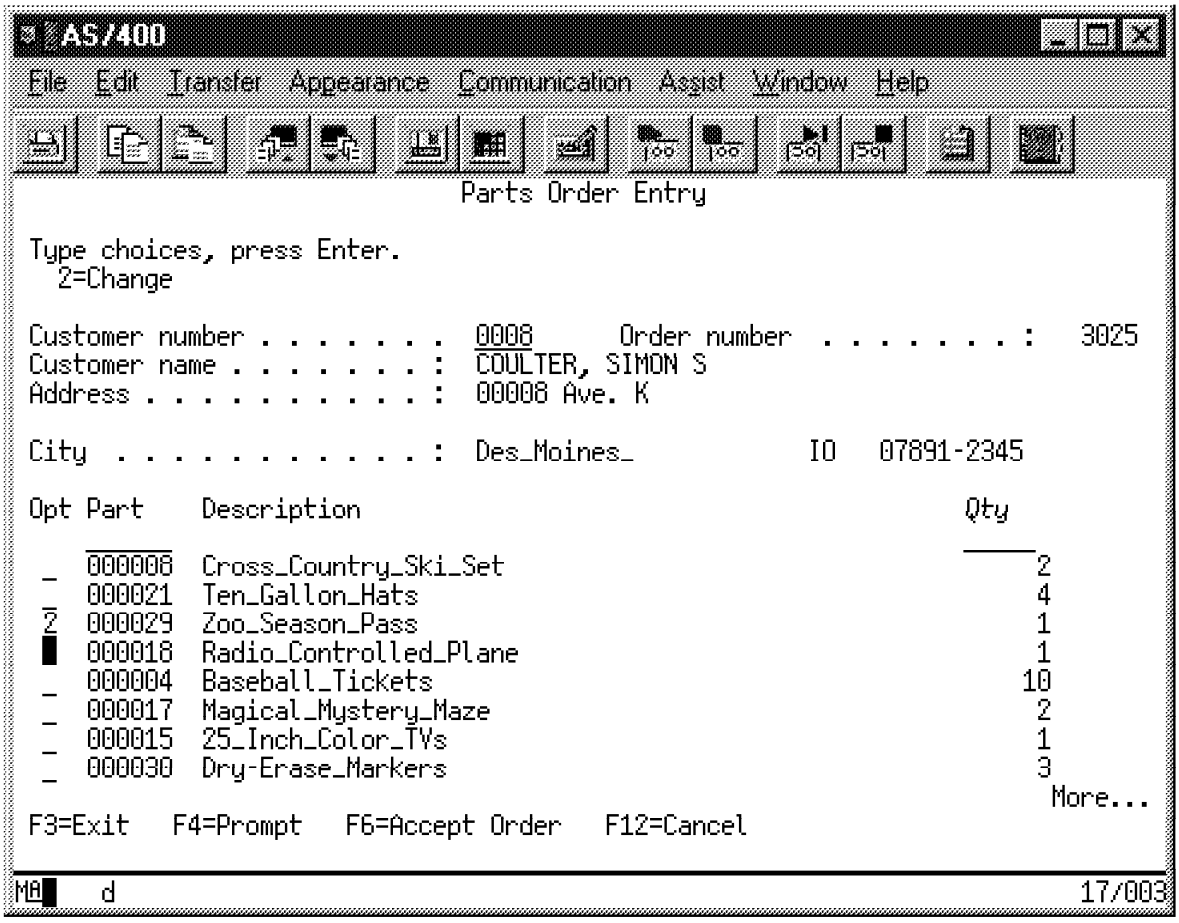


Figure 69. Parts Order Entry

The user may type a **2** beside an entry in the list to change the order. Pressing the Enter key causes a window to appear allowing the order line to be changed.

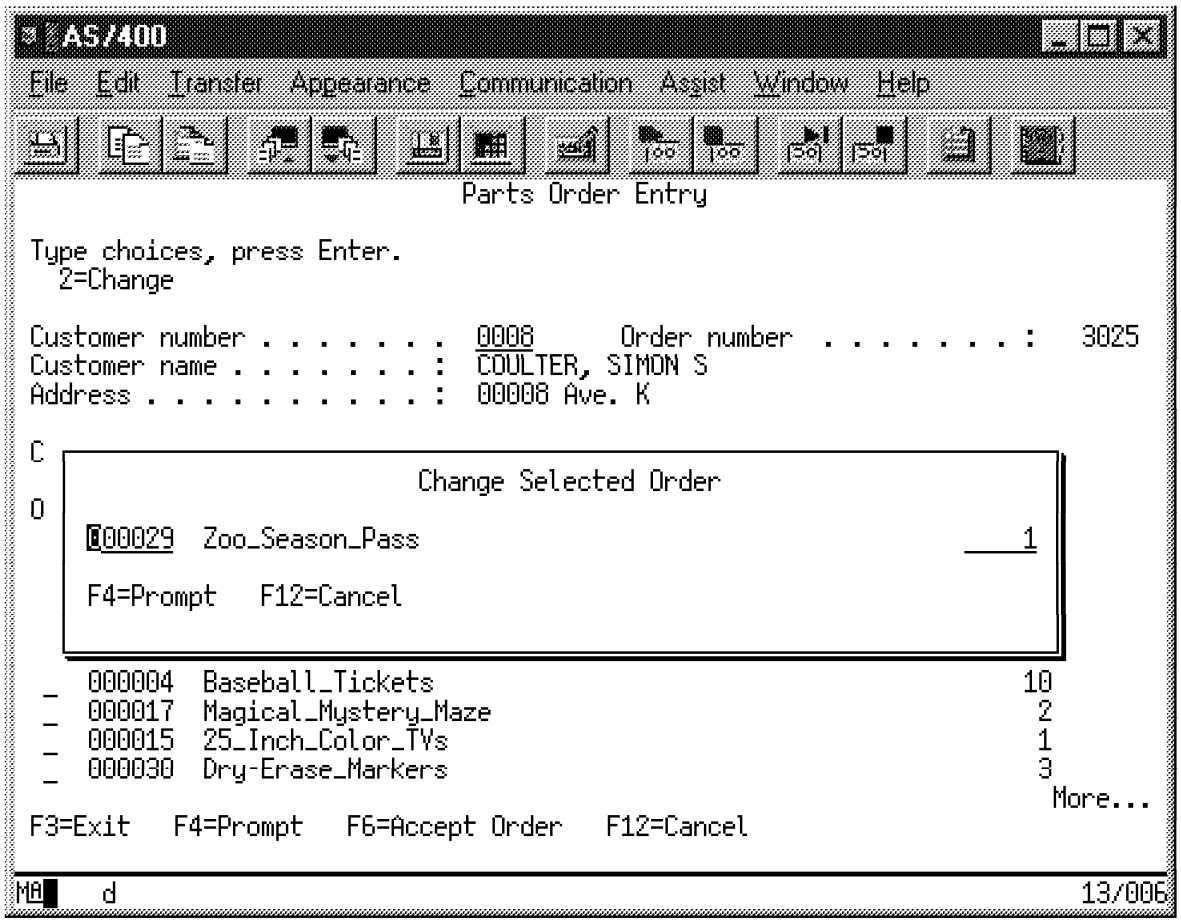


Figure 70. Change Selected Order

The user may press F12 to cancel the change, press F4 to list parts, or type a new part identifier or different quantity. Pressing the Enter key causes the part identifier and quantity to be validated and if valid, the order line is changed in the list and the window is closed.

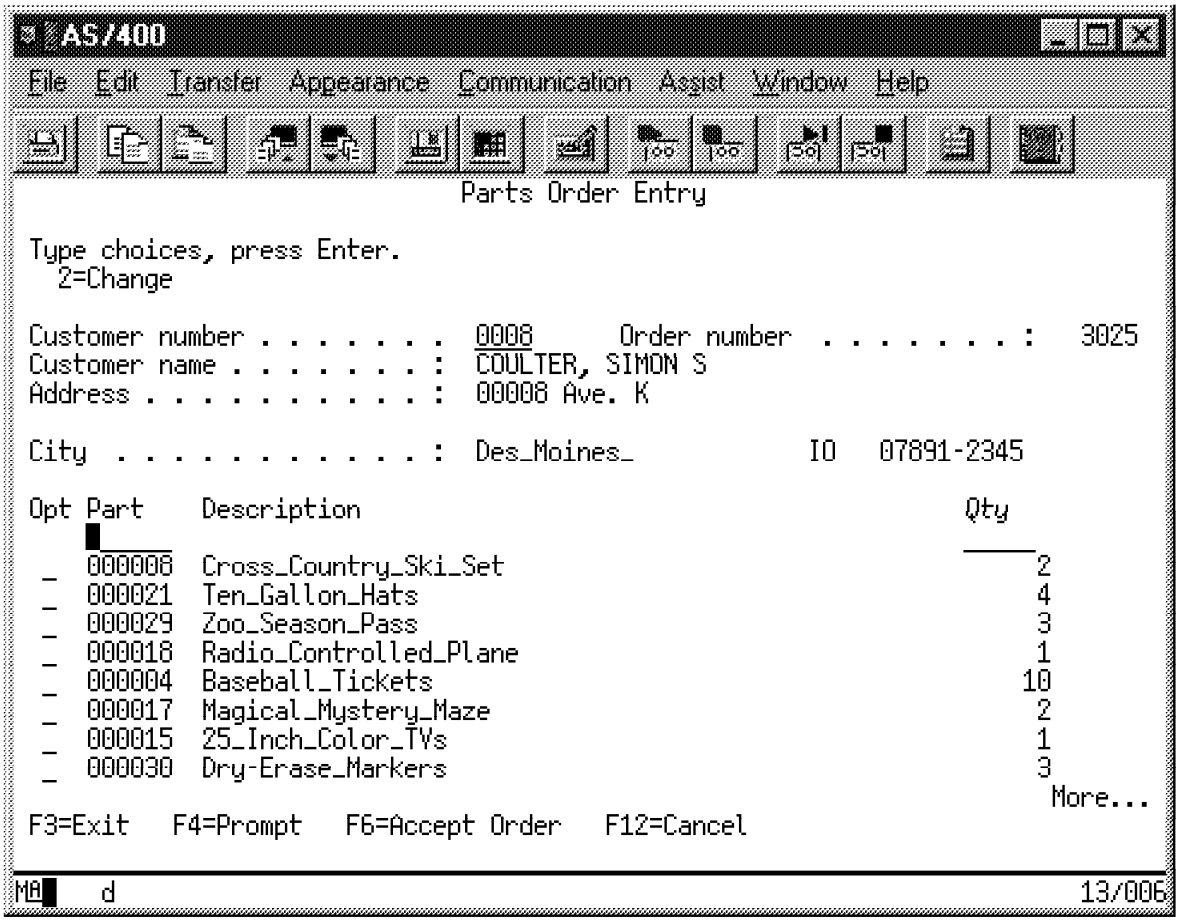


Figure 71. Parts Order Entry

Here you can see the quantity for Zoo_Season_Pass has been changed to 3. When the order is complete, the user may press F6 causing the database to be updated and an order to be printed.

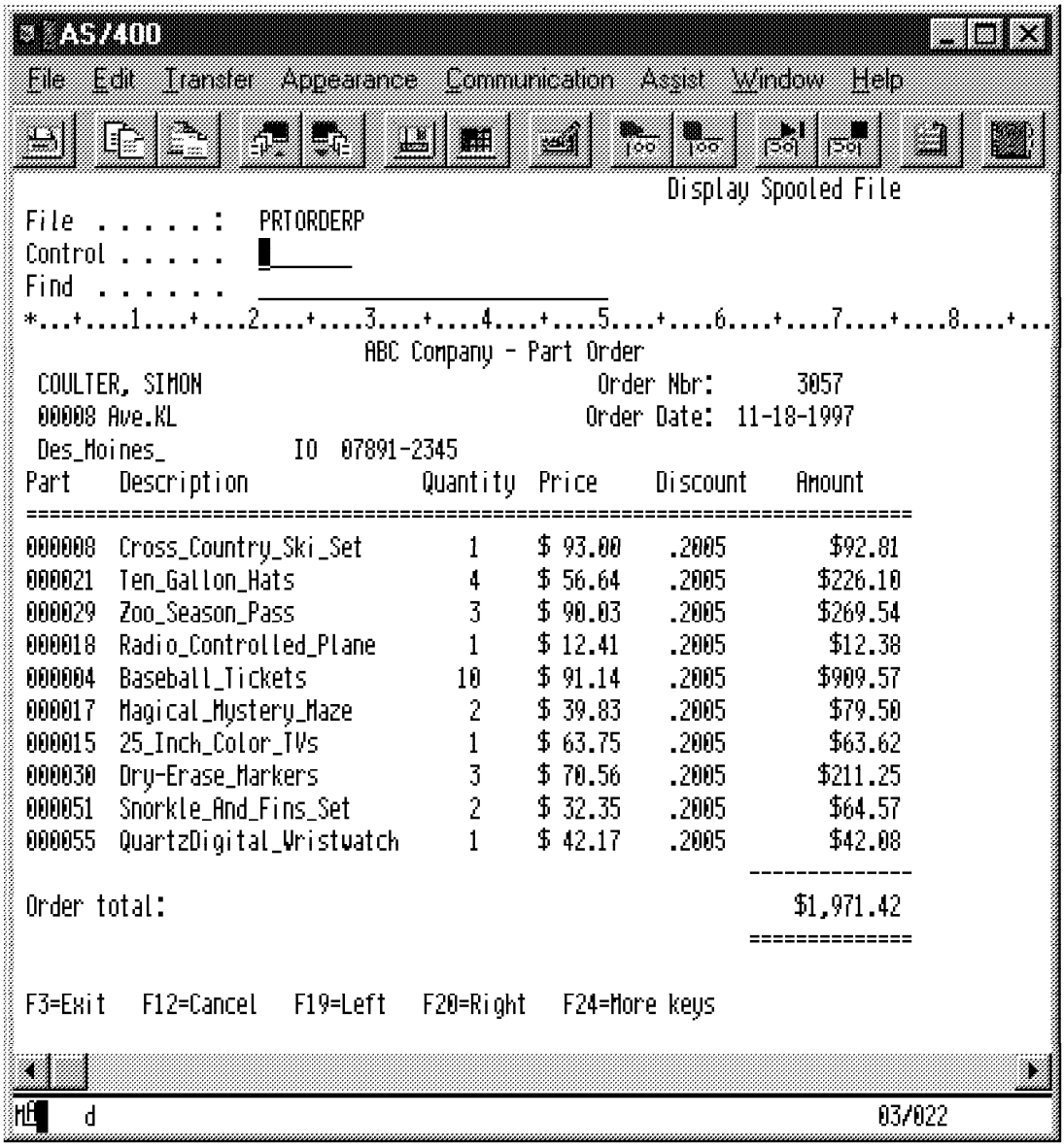


Figure 72. Printed Order

The printed order is created by a batch process. It shows the customer details and the items, quantities, and cost of the order.

5.1.1.6 Database Table Structure

The ABC Company database has eight tables:

- District
- Customer
- Order
- Order line
- Item
- Stock
- Warehouse (not used)
- History (not used)

The relationships among these tables are shown in the following diagram:

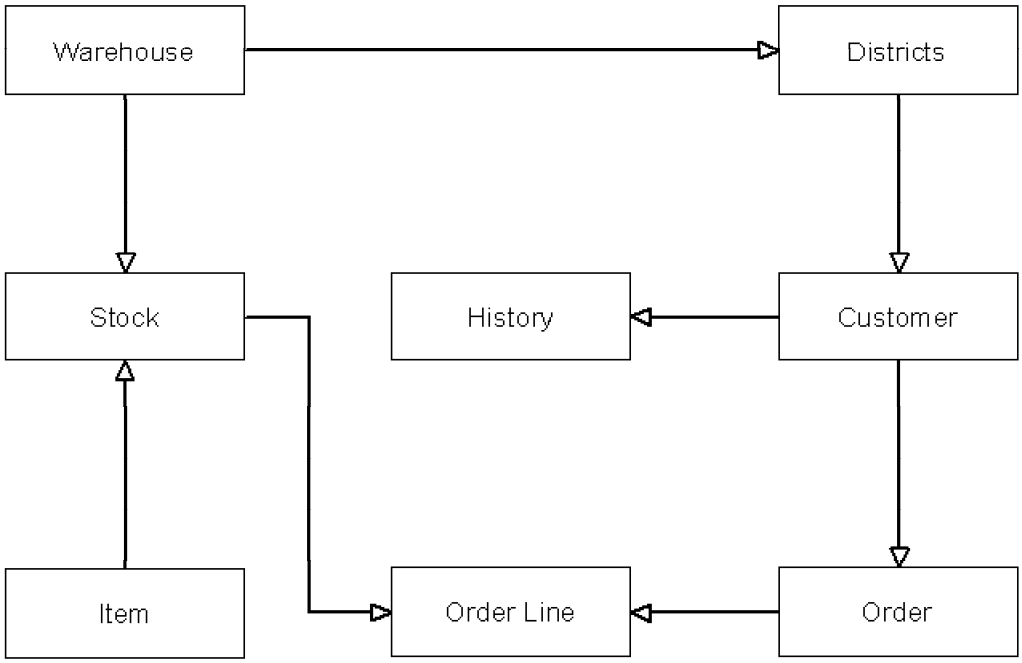


Figure 73. ABC Company Database Table Relationships

5.2 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 tables.

5.2.1 District

Table 1 (Page 1 of 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

Table 1 (Page 2 of 2). District Table Layout (DSTRCT)			
Field Name	**Real Name**	**Type**	**Length**
DTAX	Tax	Decimal	5
DYTD	Year to Date Balance	Decimal	13
DNXTOR	Next Order Number	Decimal	9
Note: Primary Key: DID, DWID			

5.2.2 Customer

Table 2. Customer Table Layout (CSTMR)			
Field Name	**Real Name**	**Type**	**Length**
CID	Customer ID	Character	4
CDID	District ID	Decimal	3
CWID	Warehouse ID	Character	4
CFIRST	First Name	Character	16
CINIT	Middle Initials	Character	2
CLAST	Last Name	Character	16
CLDATE	Date of Last Order	Numeric	8
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
CPAYCNT	Quantity	Decimal	5
CDELCNT	Quantity	Decimal	5
CLTIME	Time of Last Order	Numeric	6
CDATA	Customer Information	Character	500
Note: Primary Key: CID, CDID, CWID			

5.2.3 Order

Table 3 (Page 1 of 2). Orders Table Layout (ORDERS)			
Field Name	**Real Name**	**Type**	**Length**
OWID	Warehouse ID	Character	4
ODID	District ID	Decimal	3
OCID	Customer ID	Character	4

Table 3 (Page 2 of 2). Orders Table Layout (ORDERS)			
Field Name	**Real Name**	**Type**	**Length**
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
Note: Primary Key: OWID, ODID, OID			

5.2.4 Order Line

Table 4. Order Line Table Layout (ORDLIN)			
Field Name	**Real Name**	**Type**	**Length**
OLOID	Order ID	Decimal	9
OLDID	District ID	Decimal	3
OLWID	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	8
OLDLVT	Delivery Time	Numeric	6
OLDSTI	District Information	Character	24
Note: Primary Key: OLWID, OLDID, OLOID, OLNBR			

5.2.5 Item (Catalog)

Table 5. Item Table Layout (ITEM)			
Field Name	**Real Name**	**Type**	**Length**
IID	Item ID	Character	6
INAME	Item Name	Character	24
IPRICE	Price	Decimal	5
IDATA	Item Information	Character	50
Note: Primary Key: IID			

5.2.6 Stock

Table 6 (Page 1 of 2). Stock Table Layout (STOCK)			
Field Name	**Real Name**	**Type**	**Length**
STWID	Warehouse ID	Character	4
STIID	Item ID	Character	6

Table 6 (Page 2 of 2). Stock Table Layout (STOCK)			
Field Name	**Real Name**	**Type**	**Length**
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
STDI010	District Information	Character	24
STYTD	Year To Date	Decimal	9
STORDRS	Quantity	Decimal	5
STREMORD	Quantity	Decimal	5
STDATA	Item Information	Character	50
Note: Primary Key: STWID, STIID			

5.3 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 databases; in other cases, we use native database access.

The terminology used for the database access is different in both cases. In Table 7, you find the correspondence between the different terms.

Table 7. 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 necessary to create a Java Graphical User Interface that interacts with the **Order Entry** application discussed in Chapter 5, “Overview of the Order Entry Application” on page 93. The user interface is designed so that minimal changes are needed in the RPG code. Furthermore, the host application is changed so that it can handle both an invocation from a Java client as well as a native invocation involving no Java interface. The following figure shows the original design of the Order Entry application.

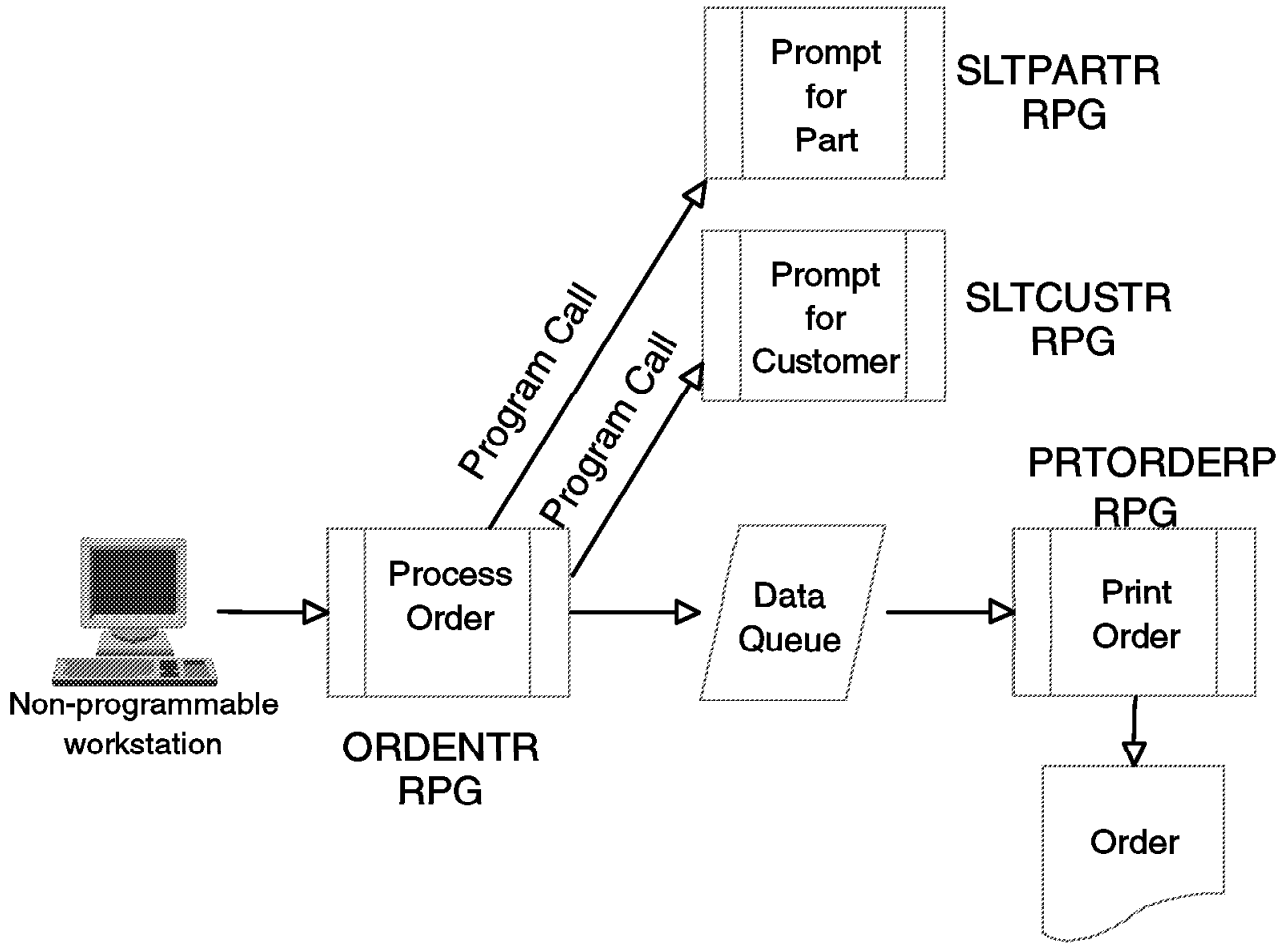


Figure 74. Original Order Entry Application

We migrate the application to a Java Client that provides a graphical user interface. 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. The following figure shows the new design.

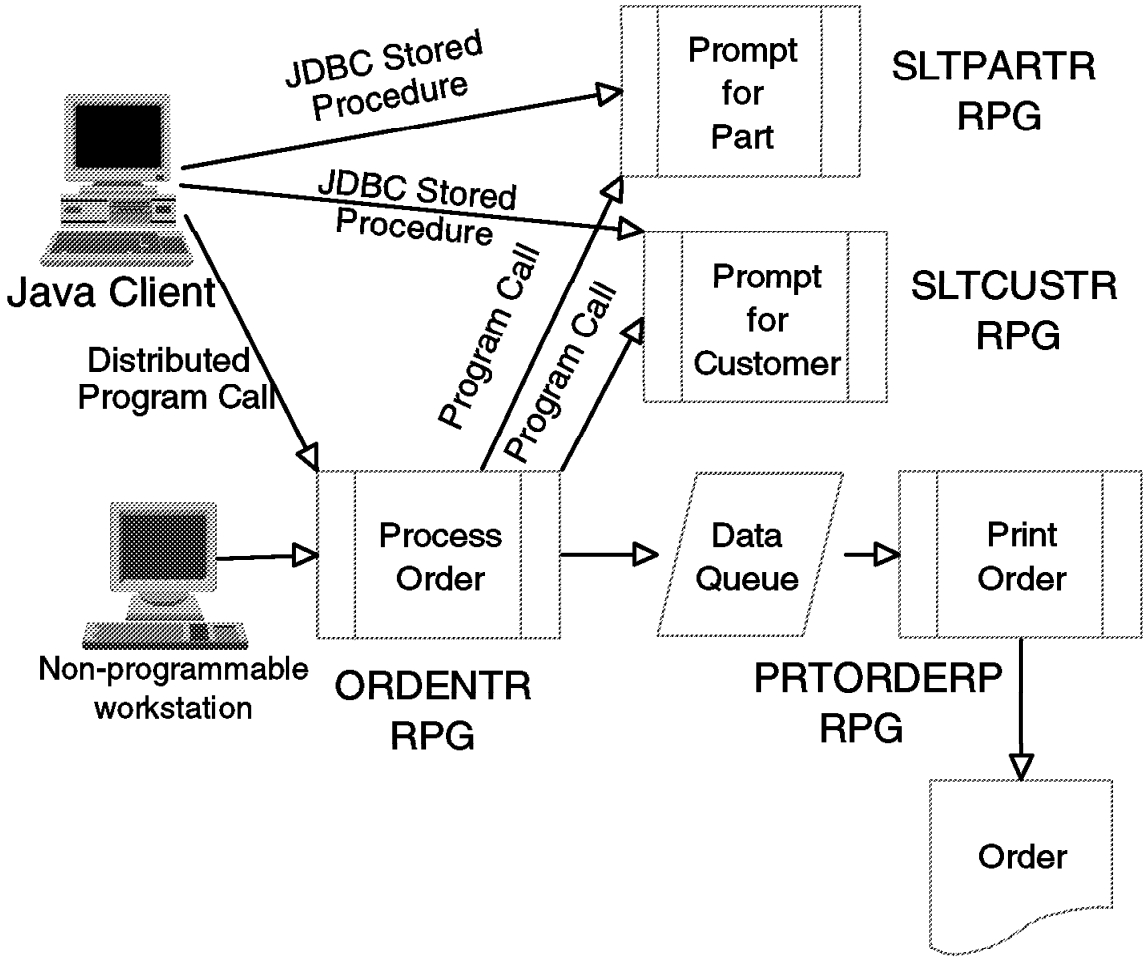


Figure 75. Java Client Order Entry Application

6.1 Creating the Java Client Graphical User Interface

First, we analyze the steps and code involved in creating the Java Graphical User Interface. It is built using **VisualAge for Java**. The **AS/400 Toolbox for Java** is used to access data and programs on the AS/400 system. The following Toolbox topics are covered:

- Stored procedures using the **AS/400 Toolbox for Java** JDBC driver
- DDM Record Level Access
- Distributed Program Call

Familiarity with **IBM's VisualAge for Java** as well as a basic understanding of the **AS/400 Toolbox for Java** is assumed. For more information pertaining to these topics, see the redbook *Accessing the AS/400 System with Java*, SG24-2152-00.

The changes made in the host RPG application are also discussed later in this chapter.

We begin by focusing on the "look" of the window that is designed to interact with the **Order Entry** application. The components contained in the window are discussed. Subsequent sections explore the issues relating to the functionality associated with the individual components in the window.

6.2 Overview of the Parts Order Entry Window

The following figure shows the main order entry window that is designed.

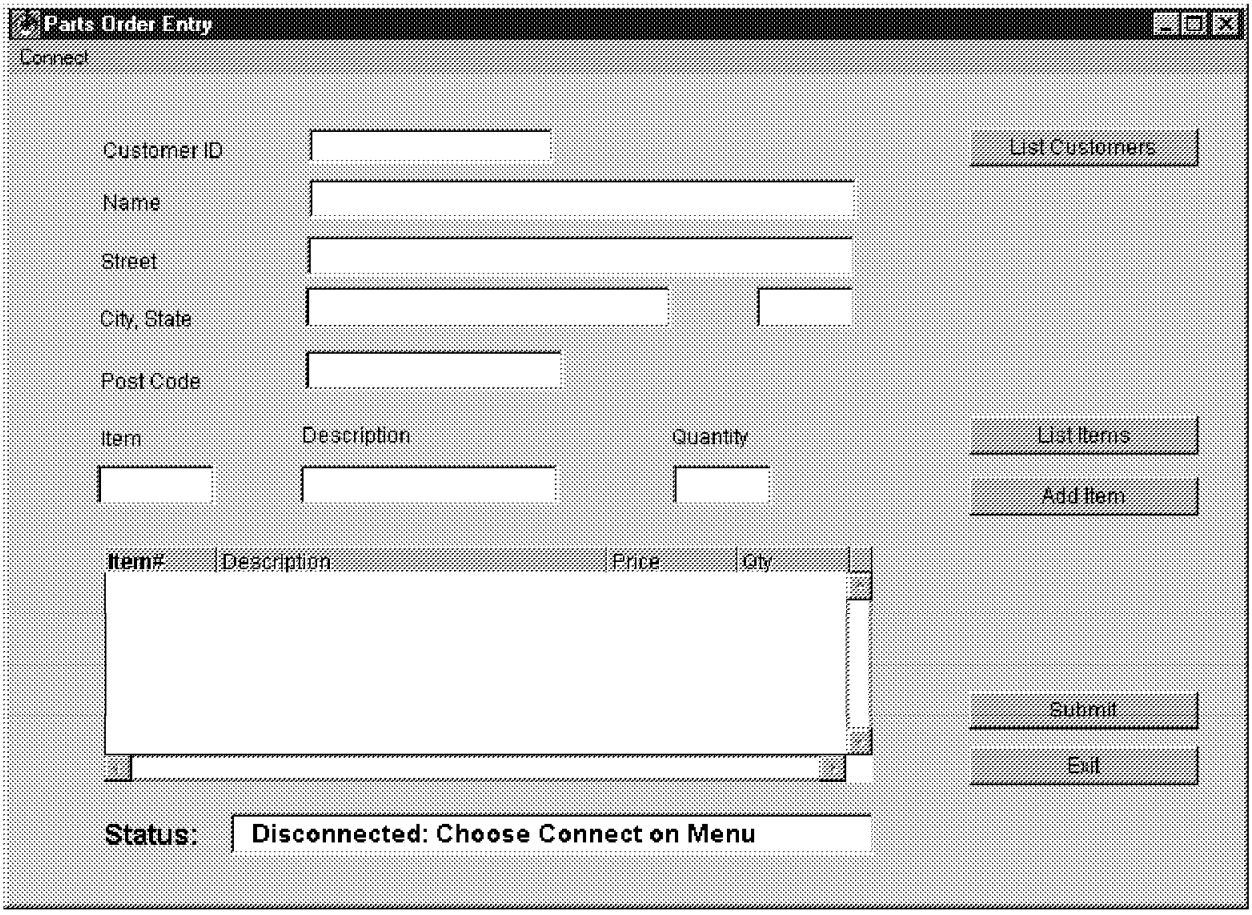


Figure 76. Parts Order Entry - Initial Panel

This window is built using **VisualAge for Java**. The name of the class that defines this window is **OrderEntryWdw**. It is the controlling class (or entry point) of the client application.

The following list contains the primary components contained in the window:

- 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 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).

The following Java code is a partial listing of the class definition for **OrderEntryWdw**:

Program Listings

For complete listings of all the code examples shown throughout this book, please refer to Appendix A, “Example Programs” on page 203 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 OrderEntryWdw 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 that is included with the **java.sql package**. Finally, we declare two objects from classes provided by the **AS/400 Toolbox for Java** - an **AS400** object, and a **KeyedFile** object. We now examine how these objects (in conjunction with GUI controls and other objects) are used to 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 the following 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.

Each of these tasks is now examined.

6.3.1 Connecting to Database

Once the **Parts Order Entry Window** has initially displayed, the **Connect** menu option is chosen to connect to the AS/400 system.

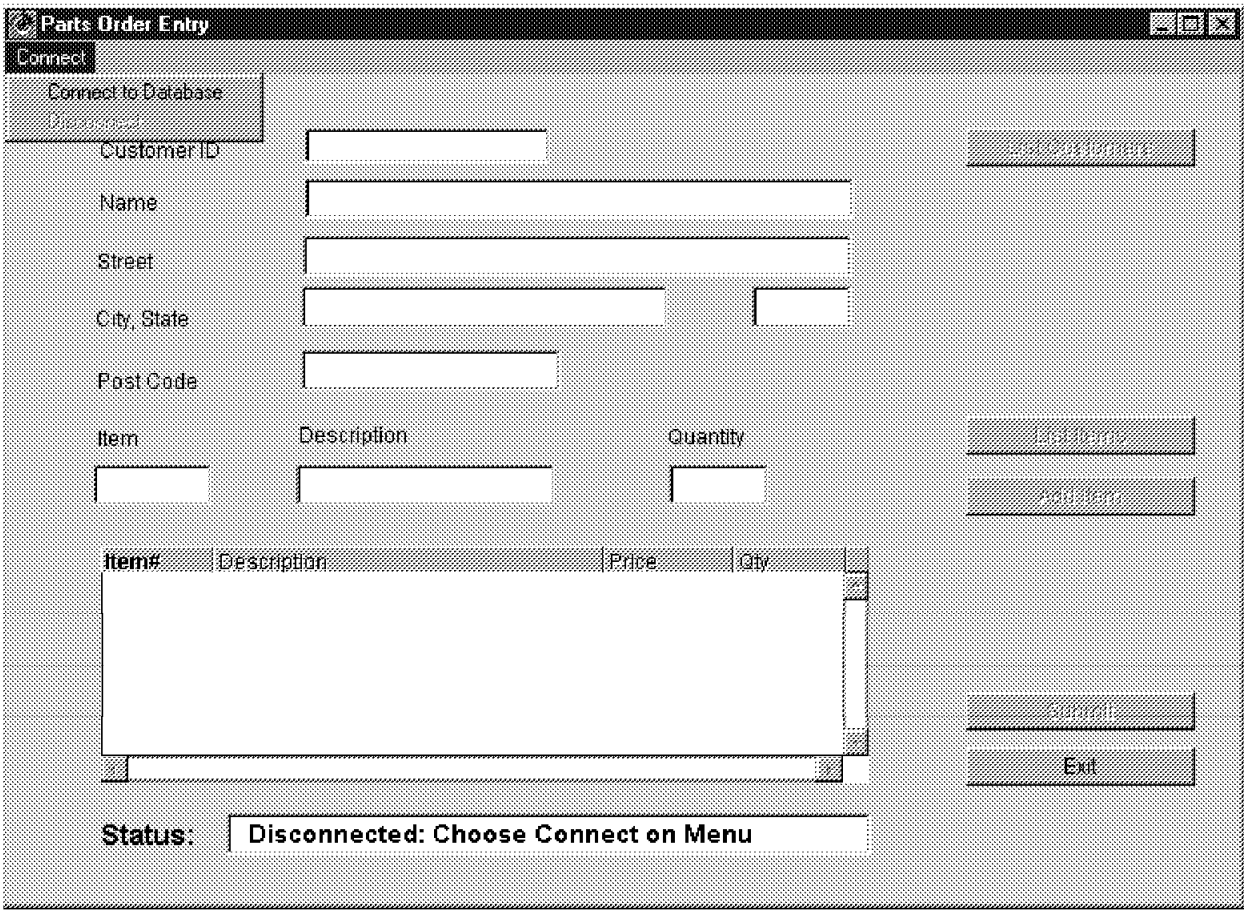


Figure 77. Parts Order Entry - Database Connect

A dialog is shown that requests sign-on information. A machine name, userid, and password must be entered:

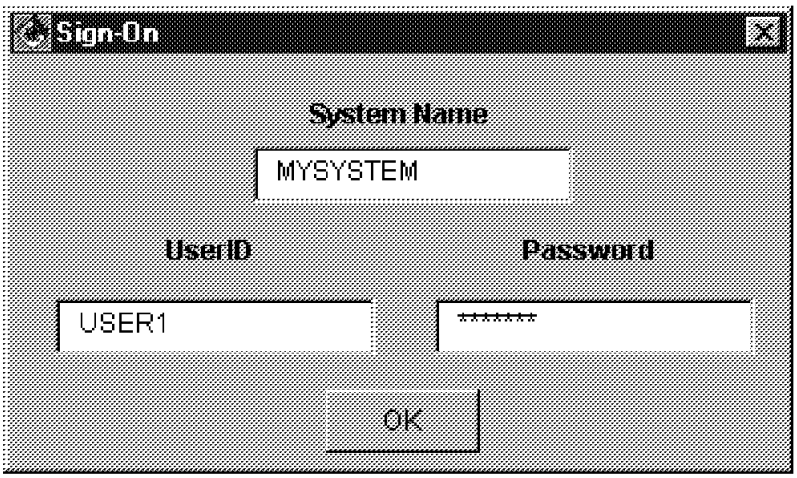


Figure 78. Sign On to System

The action of the **OK** button being pressed is connected to the **connectToDb( )** method of the **OrderEntryWdw** class. The code for this method follows:

```
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: invoking the **openItemFile( )** method, and establishing 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**. It is discussed in greater detail later.

After invoking the **openItemFile( )** method, the AS/400 JDBC driver is loaded with the following statement:

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

Next, the SQL connection is established by invoking 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 mentioned previously, 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 intializes the **as400** data member and establishes the **RECORDACCESS** connection for this object. The **itemFile** object is initialized so that 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, commitment control is not used. A blocking factor of one is used 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 relating to 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

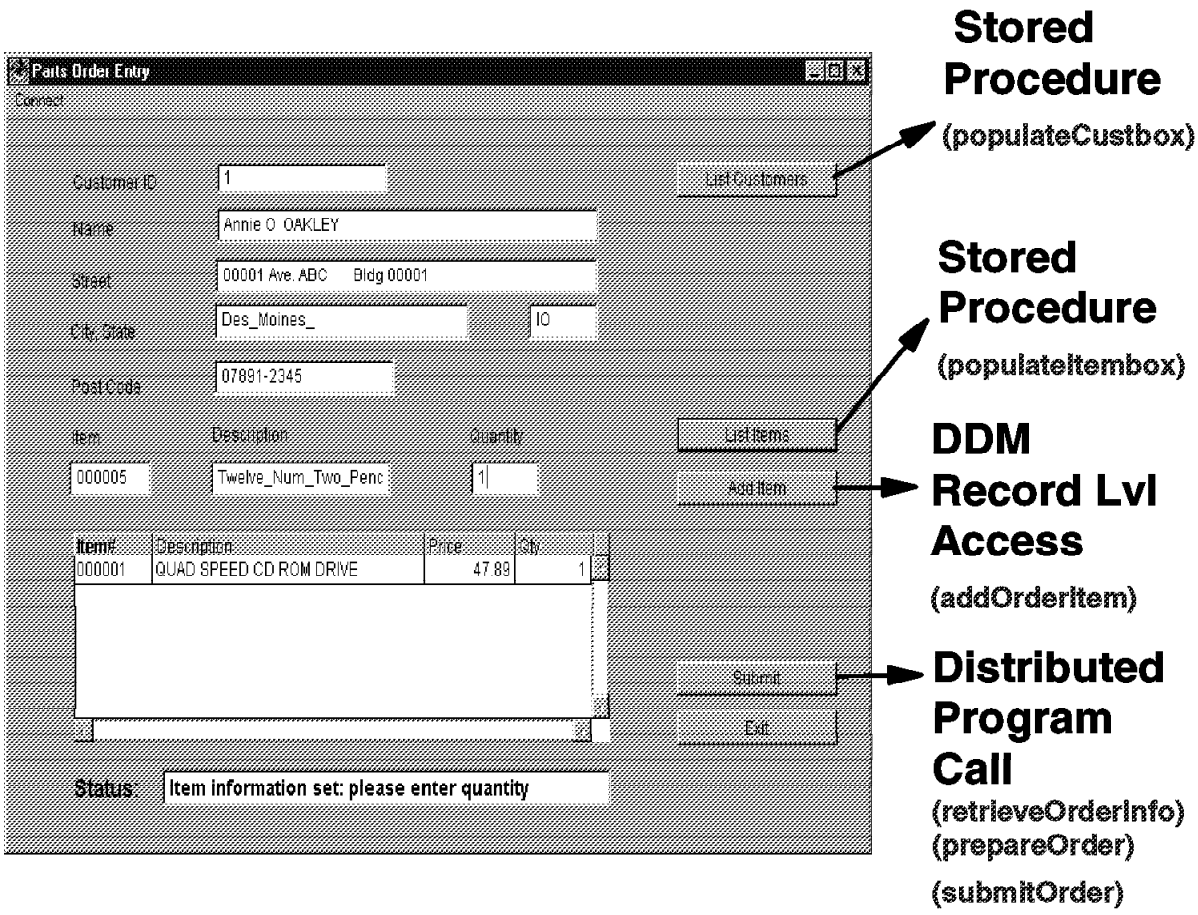


Figure 79. Java Client Programming Interfaces

The Java client program uses a number of different interfaces to access the AS/400 system. All of these interfaces are provided by the AS/400 Toolbox for Java. ODBC stored procedures are used to populate the customer listbox and the item listbox. DDM record level access is used to verify items being ordered and the Distributed Program Call interface is used to submit an order. In the following sections, we examine the coding implementation of these interfaces in more detail.

6.3.3 Retrieving the Customer List

In this section, we examine the code that enables us to retrieve a list of valid customers from the AS/400 system. The following figure shows the window that is displayed once the list of customers has been retrieved.

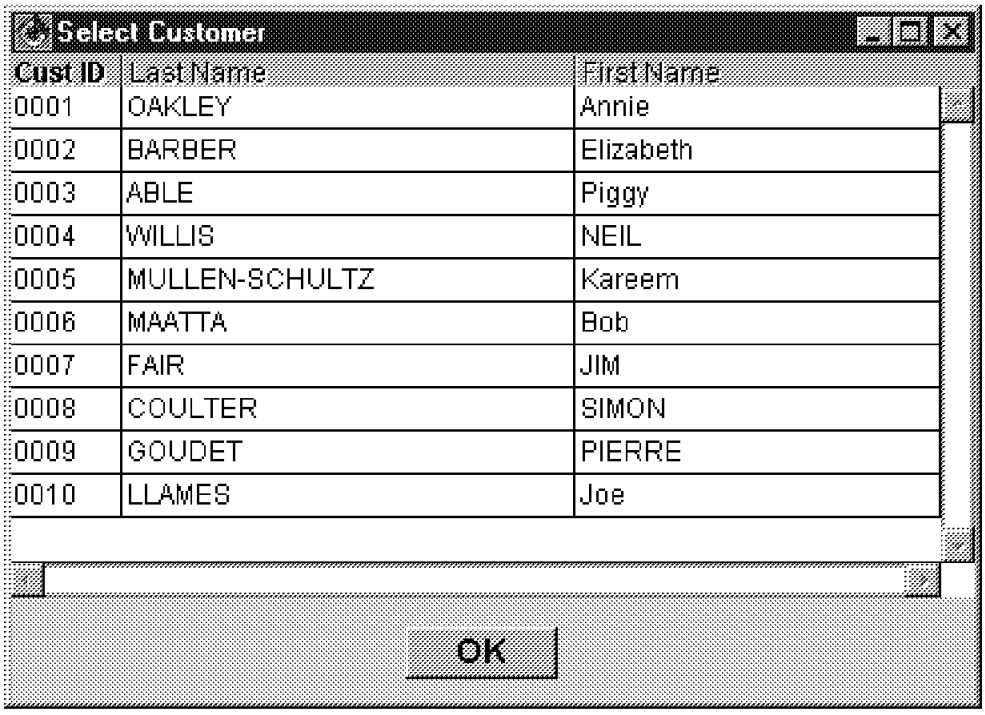


Figure 80. Select a Customer

SltCustWdw is the class that defines this window. It contains a Taligent multi-column list box and one button. JDBC is used to access 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 is used to populate the list box.

Note: The Taligent classes used in these examples are available from www.Taligent.com.

The following 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.taligent.widget.MultiColumnListboxivjCustMLB = null;
  private java.awt.Button ivjSltCustBTN = null;
  private OrderEntryWdw orderWindow = null;
}
```

When the **List Customers** button on the order entry window is clicked on, the constructor for **SltCustWdw** is invoked. The constructor receives a reference to the order entry window as well as a reference to the SQL connection object. The code for the constructor follows:

```
public SltCustWdw ( OrderEntryWdw 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. This 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())
     {
        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;
} // end method
```

As previously shown, **populateCustbox( )** receives a result set through an invocation of a remote stored procedure. This is done by invoking the **prepareCall( )** method, which returns a **CallableStatement** object. The **executeQuery( )** method of this object executes 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 is defined to call an RPG program 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 cover 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. The code for this method follows:

```
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 OrderEntryWdw
  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. We now 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 not discussed here.

As noted previously, the selected **Customer** is passed as a parameter to the **setSelectedCust( )** method of the **OrderEntryWdw** object. This method follows:

```
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**. The following figure shows the current state of the main window:

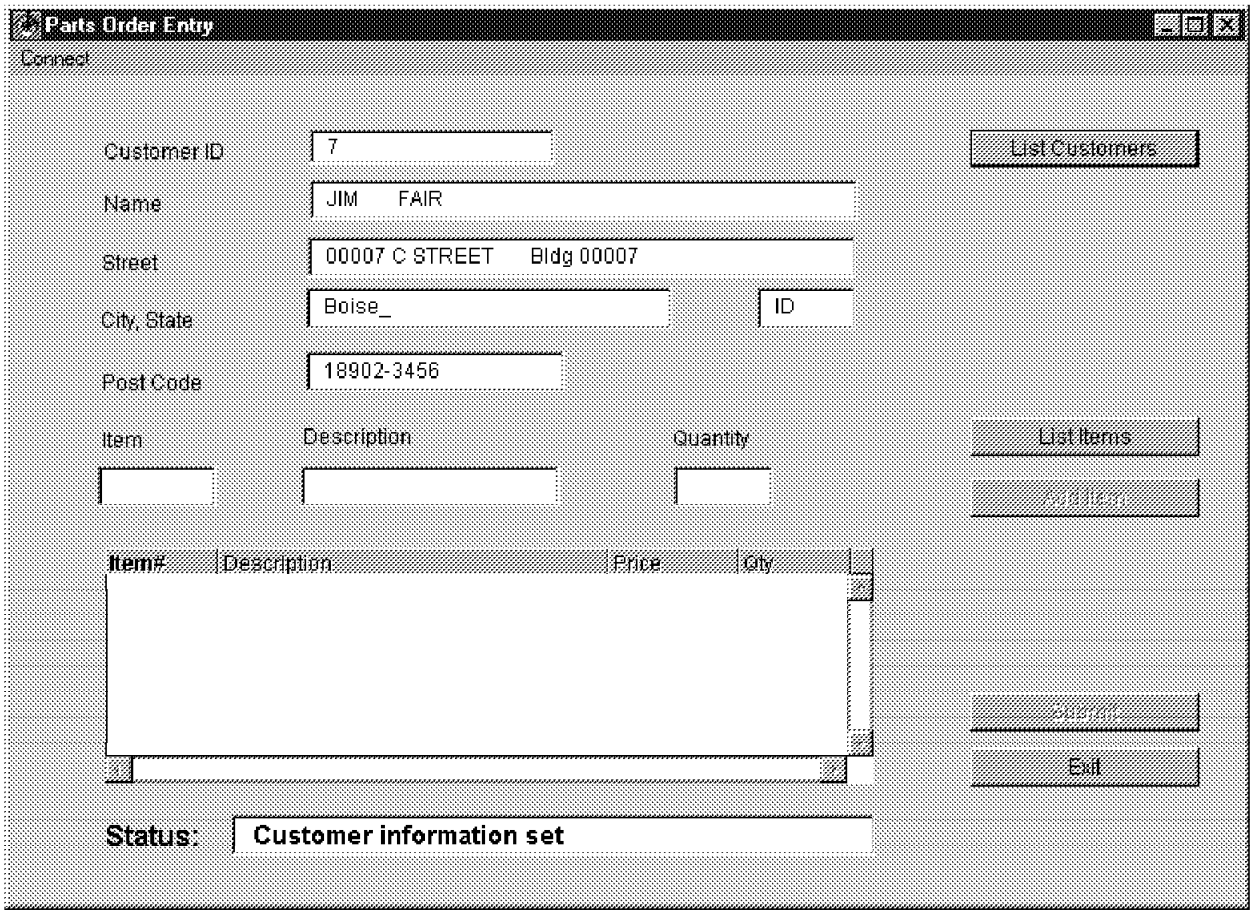


Figure 81. Order Entry Window with Customer Data

Once the information for the selected customer has been 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 enables us to retrieve a list of valid items from the AS/400 system. The process is similar to retrieving the customer list. The following figure shows the window that is displayed once the list of items has been retrieved.

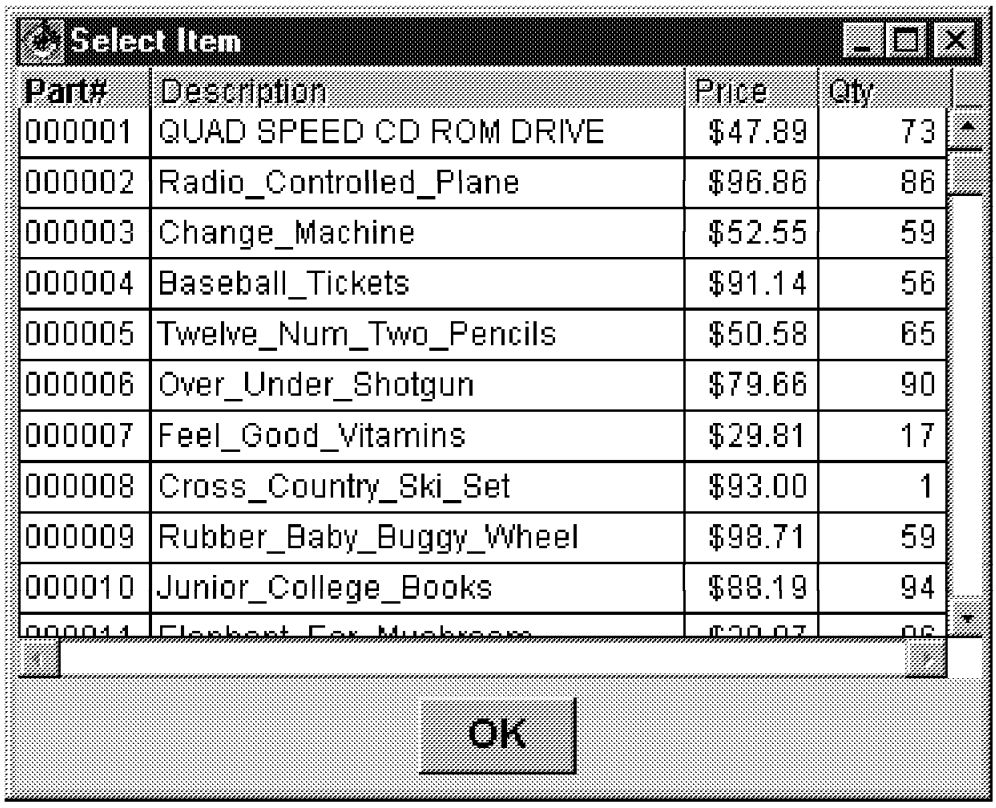


Figure 82. Selected Items

SltItemWdw is the class that defines this window. It contains a Taligent multi-column list box and one button. JDBC is used to access 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 is used to populate the list box.

The following 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.taligent.widget.MultiColumnListbox
                                ivjItemMLB = null;
  private OrderEntryWdw orderWindow = null;
}
```

When the **List Items** button on the order entry window is clicked on, the constructor for **SltItemWdw** is invoked. The constructor receives a reference to the order entry window as well as a reference to the SQL connection object. The code for the constructor follows:

```
public SltItemWdw (OrderEntryWdw 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. This 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())
    {
       array[0] = rs.getString("IID");
       array[1] = rs.getString("INAME");
       array[2] = "$"+rs.getBigDecimal("IPRICE",2).toString();
       array[3] = Integer.toString(rs.getInt("STQTY"));
       getItemMLB().addRow(array);
    }   // 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. This is done by invoking 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 then clicks on the **OK** button. This action is connected to the invocation of the **itemSelected( )** method. The code for this method follows:

```
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. The **ITEM** class is now examined.

```
public class Item
{
  private String id;
  private String name;
  private java.math.BigDecimal price;
  private int quantity;
}
```

The **ITEM** class is an object-oriented representation of a record in the **ITEM** file. The constructor takes a **String** array as a parameter and then sets the data members accordingly:

```
public Item ( String[] itemInf)
{
   id = itemInf[0];
   name = itemInf[1];
   // remember to trim the '$' symbol before
```

```
   // instantiating a BigDecimal
   price =  new java.math.BigDecimal(itemInf[2].substring(1));
   quantity = new Integer(itemInf[3]).intValue();
}
```

The **setSelectedItem( )** method in the **OrderEntryWdw** class puts the **ITEM id** and the **ITEM name** in the window. It positions the cursor to the quantity field, where a number must be entered before the entry is added to the order list:

```
public void setSelectedItem(Item selectedItem)
{
  getItemTF().setText(selectedItem.getId());
  getDscTF().setText(selectedItem.getName());
  getQtyTF().requestFocus();

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

  return;
}
```

6.3.5 Verifying the Item and Adding It 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 provided by the **AS/400 system Toolbox for Java** 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 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 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
         Object[] orderRow = new Object[4];
         orderRow[0] = data.getField("IID");
         orderRow[1] = data.getField("INAME");
         orderRow[2] = data.getField("IPRICE");
         orderRow[3] = getQtyTF().getText();
             getOrderMLB().addRow(orderRow);
         getOrderMLB().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 has been added to the order list, the **Submit** button is enabled. More items may be added to the list, or the order may be submitted. The following diagram shows the state of the window at this point.

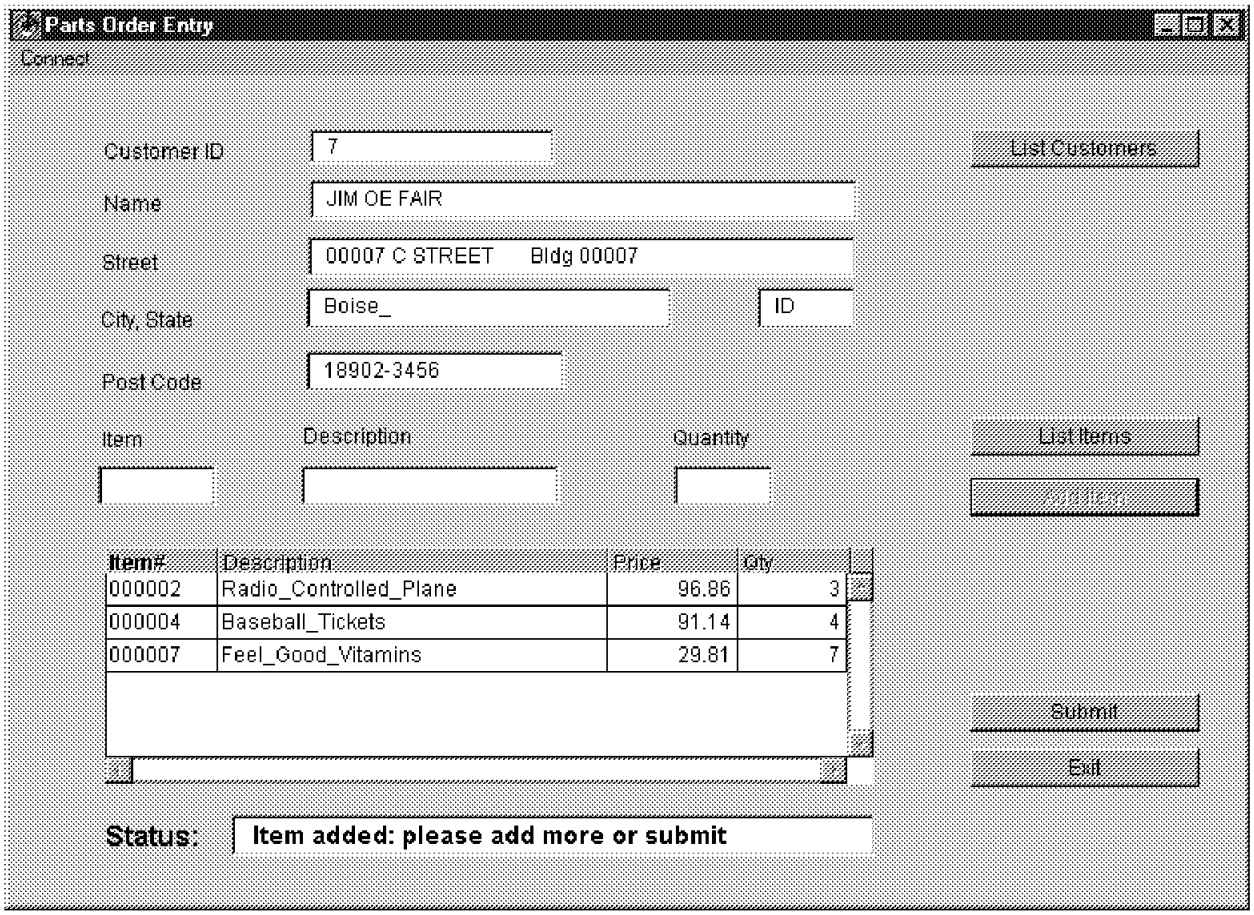


Figure 83. Parts Order Ready to Submit

6.3.6 Submitting the Order

When the **Submit** button is clicked on, 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. A **Vector** can also be used, which avoids having to determine a size in advance (vectors allow dynamic allocation, whereas 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 that 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 consisting 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:

The complete listing of the class is available in the example code. Please see Appendix A, “Example Programs” on page 203 for information on how to download the code. Other methods such as **getFirstEntry( )** and **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, care must be taken 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**. This 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 designated as starting points of certain data members. The breakdown of the **orderDetail** buffer is as follows:

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 created an **Order** and added **OrderDetail** objects to it. Next, the **retrieveOrderInf( )** method invokes the **prepareOrder( )** method and passes the **Order** object as a parameter. The **prepareOrder( )** method builds the constructs that are used as parameters when invoking the AS/400 system RPG program that processes orders **(ORDENTR**). Two parameters are required by this program.

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. The last 5 bytes are moved to a zoned numeric field. These 9 bytes of data are the first 9 bytes in the string returned by **Order.toString( ).**

The second parameter is a block of character data that represents all 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( )** invokes the **submitOrder( )** method. **SubmitOrder( )** calls the **ORDENTR** program on the AS/400 system using the **Distributed Program Call** class 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;
}
```

Let's analyze the method. First, we start the **AS400.COMMAND** service for the **as400** object. It should be noted that the **Toolbox** implicitly starts this if it needs to be. It is shown here for clarification.

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

Then we declare a **ProgramCall** object called **ordEntrPgm**. We then 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, the parameters for the program must be set. The scenario for setting up parameters is as follows.

First, we must declare an array of **ProgramParameter** objects. We declare an array of two since the program requires two parameters:

```
ProgramParameter[] parmList = new ProgramParameter[2];
```

We must now construct the individual ProgramParameter elements to fill the array. The **ProgramParameter** constructor must be passed an array of bytes. So before instantiating a ProgramParameter object, we must first generate the appropriate array of bytes.

The process of creating the array of bytes has two steps. First, declare an appropriate AS/400 system data type object. In this case, we are passing string data, so we declare an **AS400Text** object. Then we invoke the **toBytes( )** method on this **AS400Text** object and pass the string that is being converted to bytes. Here are the two steps:

```
AS400Text text1 = new AS400Text(9);
byte[] headerBytes = text1.toBytes(header);
```

Now we can invoke the constructor for the **ProgramParameter** class.

```
parmList[0] = new ProgramParameter(headerBytes);
```

The same scenario is followed for each additional parameter needed. Once the parameters have been instantiated, the **ProgramCall** object must be initialized with the actual name of the AS/400 program being called. The parameter list must also be specified. This is done by invoking the **setProgram()** method:

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

We are now ready to execute the program. The **run( )** method invokes the program. In this sample, it is done inside an "if" construct so that 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 executes successfully, the **submitOrder( )** method updates the status text field accordingly.

This concludes the application flow from the Java client perspective. We now 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 that 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 invoking 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:

- Execute **OpenCust** subroutine (declares and opens cursor for **CSTMR** file).
- Execute **BldSfl** subroutine (populates and displays the sub-file).
- Execute **Process** subroutine (detects the chosen customer and displays).

The new logic flow can be summarized as:

- Execute **CloseCust** subroutine (effectively resets cursor for multiple client requests).
- Execute **OpenCust** subroutine (same behavior as original version).
- If more than one parm, execute **ResultSet** subroutine (makes result set available to the caller of the stored procedure).
- If only one parm, continue as original version did: execute **BldSfl**, then execute **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 is executed and control is returned. The sub-file processing is bypassed:

```
C                   IF        NbrParms > 1
C                   EXSR      ResultSet
C                   RETURN
C                   ENDIF
```

The file specifications for the display file are changed so that user controlled open is specified (the keyword **USROPN** is added). There is no reason to open the file if the program is invoked 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 **OrderEntryWdw.setSelectedCust( )** methods.

6.4.2 Providing an Item List

We saw earlier that the **List Items** button invokes 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**:

- Execute **OpenPart** subroutine (declares and opens cursor for **ITEM** and **STOCK** file).
- Execute **BldSfl** subroutine (populates and displays the sub-file).
- Execute **Process** subroutine (detects the chosen part/item and displays it).

The new logic flow can be summarized as:

- Execute **ClosePart** subroutine (effectively resets cursor for multiple client requests).
- Execute **OpenPart** subroutine (same behavior as original version).
- If more than one parm, execute **ResultSet** subroutine (makes result set available to the caller of the stored procedure).

- If only one parm, continue as original version did: execute **BldSfl**, then execute **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 offered by the **AS/400 Toolbox for Java**. 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 that it recognizes the fact that it is being invoked 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 invoked 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 representing 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 invoked 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** is invoked 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 sub-file 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 that 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** is invoked as a distributed program, all display file processing is bypassed. The information normally received from the display files and sub-files 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 141.

Chapter 7. Moving the Server Application to Java

So far we have shown 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.

We next migrate the application to AS/400 Server Java. We use the Java Remote Method Invocation (RMI) interface to allow the client Java program to interface with the server Java code. The following figure shows the new design.

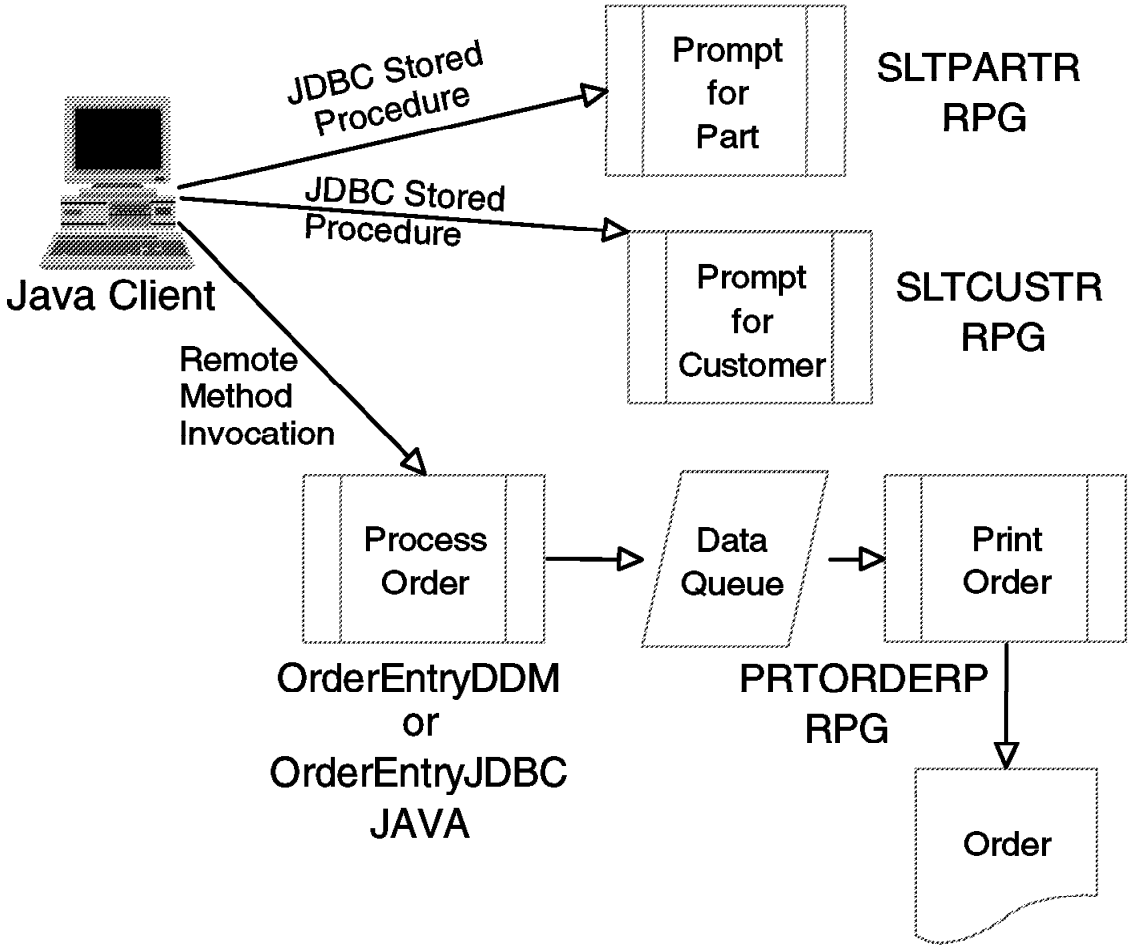


Figure 84. Java Client/Java Server Order Entry Application

This chapter discusses the Java code necessary to replace the RPG order entry application. We discuss two techniques: using Record Level access and using AS/400 JDBC. We also discuss the changes required for the client code.

There are many possible approaches to creating the AS/400 Java order entry application:

1. 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.
2. We can completely redesign the application to be fully object-oriented. This can involve creating classes to encapsulate all the files used, a class to hide the data queue implementation, and classes to describe an order and a customer.
3. 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, feeling that it is 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.

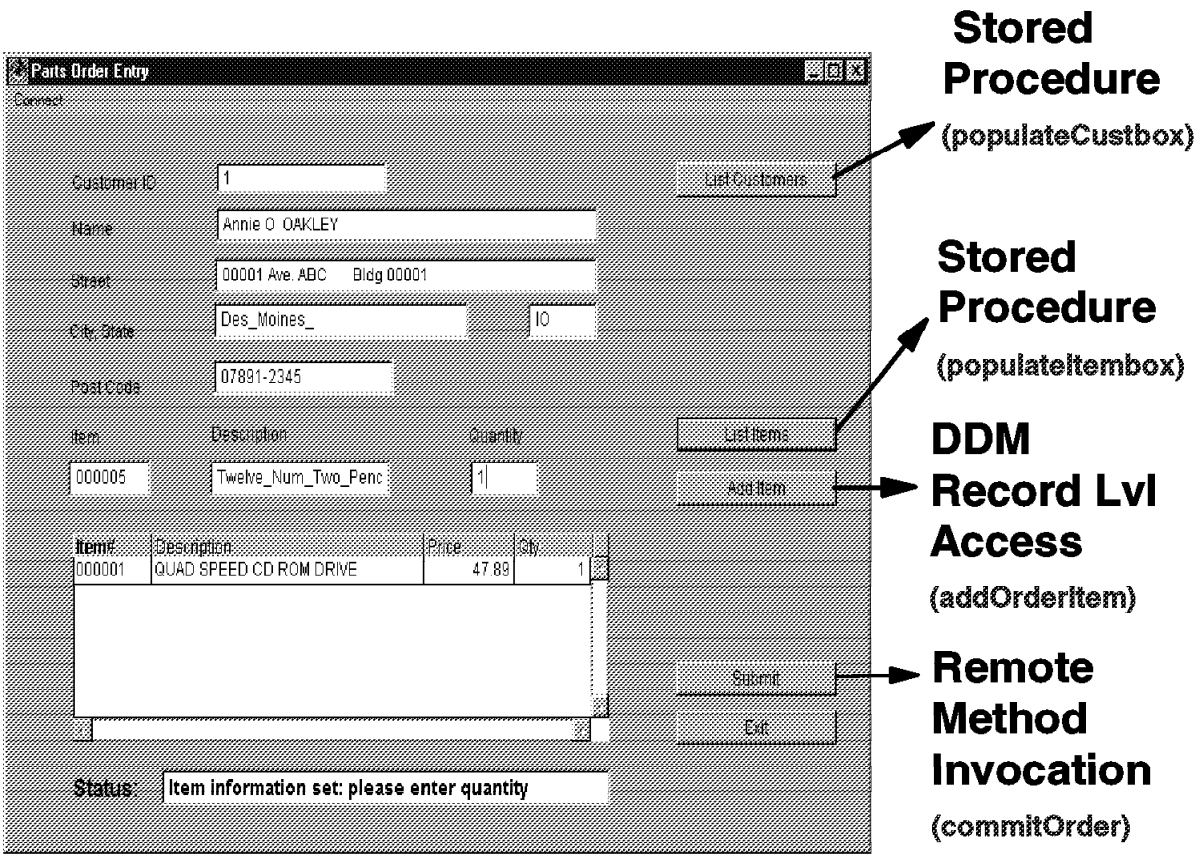


Figure 85. Java Client/Java Server Interfaces

The client Java program is comparable to the Java client program covered in Chapter 6, “Migrating the User Interface to Java Client” on page 109. The major difference is that now, the submit order processing is done through the Java remote

method invocation (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. There are four choices:

- JDBC-ODBC bridge
- AS/400 Toolbox JDBC running on the AS/400 system
- AS/400 Toolbox 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 they are both equivalent to the RPG application discussed in Chapter 5, “Overview of the Order Entry Application” on page 93. We create two classes. **OrderEntryDDM** is for the DDM record Level Access example and **OrderEntryJDBC** is for the JDBC example. Each class contains Java methods that map to the subroutines found in the RPG example. We initially implement Record Level Access because it is closer to the native I/O mechanisms used in the original RPG application and, therefore, 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 203. It also contains instructions to access the redbook Web site and 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.

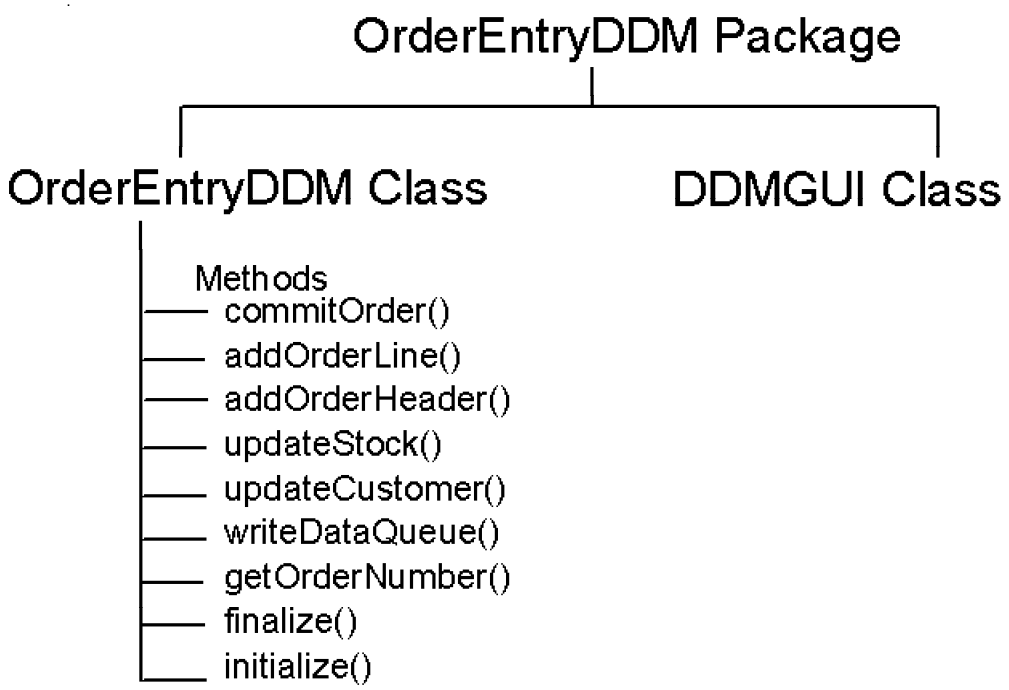


Figure 86. OrderEntryDDM Class

We first create a class called OrderEntryDDM in the OrderEntryDDM package. The basic class code follows with elaboration:

Here we specify that the class **OrderEntryDDM** is in the OrderEntry package.

```
package OrderEntryDDM;
```

Here we define the packages used by this class.

```
import com.ibm.as400.access.*; // for AS/400 Toolbox 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
{
```

Here we 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.

```
// 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";
```

Here we create some global instance variables. These 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;
```

Here we define the default constructor for the class. It 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()

We next add these methods to the class definition. The methods are inserted after the default constructor. Notice that 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.

7.1.1.1 initialize()

Here is the basic **initialize()** method. It is responsible for ensuring that the new object has the correct starting values.

```
private void initialize ()
{
  return;
}
```

7.1.1.2 commitOrder()

Here is the basic **commitOrder()** method. This 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 will return a string indicating whether the order was processed successfully or not.

```
public String commitOrder (Order anOrder)
{
  return("Order processed successfully.");
}
```

7.1.1.3 addOrderHeader()

Here 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;
}
```

7.1.1.4 addOrderLine()

Here is the basic **addOrderLine()** method. It is responsible for inserting a record in the **ORDLIN** file for each order line created by the order entry application. It needs to know the order identifier and the actual order. Notice that 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;
}
```

7.1.1.5 getCustomerDiscount()

Here 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;
}
```

7.1.1.6 getOrderNumber()

Here 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;
}
```

7.1.1.7 updateCustomer()

Here 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;
}
```

7.1.1.8 updateStock()

Here is the basic **updateStock()** method. It is responsible for ensuring 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;
}
```

7.1.1.9 writeDataQueue()

Here 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;
}
```

7.1.2 Method Logic

Next we add the program logic to each of the methods we have created. We also need to consider exception handling. Many of the classes we use to access the AS/400 system database send an error message if they encounter problems. The mechanism 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 Display Object Description (DSPOBJD). 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. A global MONMSG can be performed in Java by adding the **throws Exception** statement to the definition of a method.

7.1.2.1 initialize()

Here is the complete **initialize()** method. This method is invoked 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 is used to process 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 representing the files used by the OrderEntry class. We choose an object appropriate to the type of processing performed. The **ORDERS** and **ORDLIN** files are only written so they can be processed sequentially—we use instances of the **SequentialFile** class to represent these files. All the other files are processed randomly so support for keyed reads is required—we use instances of the **KeyedFile** to represent these files. An example follows:

```
ordersFile = new SequentialFile(as400,
   "/QSYS.LIB/"+DATA_LIBRARY+".LIB/ORDERS.FILE/%FILE%.MBR");
```

What this single line of Java code does is to set the **ordersFile** variable to reference a new **SequentialFile** using the **as400** connection object and the name of the database file. Notice that we must use the IFS naming convention.

Then we create objects to represent a file description. We need these objects to retrieve the record formats for each file. An example follows:

```
AS400 systemFileRecordDescription ordersFileD = new AS400FileRecordDescription(as400,
   "/QSYS.LIB/PRODDATA.LIB/ORDERS.FILE");
```

What this single line of Java code does is to declare a variable called **ordersFileD**, which is a type of **AS400FileRecordDescription** and initializes it to reference a new instance of an **AS400FileRecordDescription** representing a description of the file named **"/QSYS.LIB/PRODDATA.LIB/ORDERS.FILE")** on the system represented by the **as400** connection. Notice we are using the IFS 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 a record format is used in an AS/400 system high-level language. The record format is used to describe the data for read, update, and write operations. The record format is retrieved by an instance method of the file description objects just created. An example follows:

```
RecordFormat ordersFormat = ordersFileD.retrieveRecordFormat()[0];
```

What this line of Java code does is to define 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 generated by the methods 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;
}
```

7.1.2.2 commitOrder()

Here is the complete **commitOrder())** method. This method is the public interface to the OrderEntry class. It needs to be given an order to process. The **Order** class is described in Chapter 6, “Migrating the User Interface to Java Client” on page 109. It contains the following logic:

- Request the customer number and the number of order lines from the order object.
- Determine the next order number by invoking 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 (tries) 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.");
}
```

7.1.2.3 addOrderHeader()

Here 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.

Notice the data conversions performed on the date and time. The database fields are simply numeric data types representing 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 a simple matter to create a means of doing so. The **getRawDate**() and **getRawTime**() methods are discussed in detail later.

Any exceptions generated by the methods 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;
}
```

7.1.2.4 addOrderLine()

Here is the complete **addOrderLine()** method.

- Determine the amount of discount for this customer.
- Open the **ORDLIN** file and create an empty order line record.
- The first order line is extracted from the order object.
- 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.
- Each record is written to the **ORDLIN** file.
- The value of the order is accumulated and the stock quantity for each item is updated.
- When all order lines have been processed, the total order value is returned to the caller.

Any exceptions generated by the methods 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;
}
```

7.1.2.5 getCustomerDiscount()

Here 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, the district identifier, and the 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 generated by the methods 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;
    }
}
```

7.1.2.6 getOrderNumber()

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

Notice that the **orderNumber.add(new BigDecimal(1)** method does not affect the value of the **orderNumber** variable. That is because the **orderNumber.add(new BigDecimal(1)** method returns a new instance of **BigDecimal** containing the incremented value. This is, of course, a temporary value that is passed directly to the **districtRecord.setField()** method.

Any exceptions generated by the methods 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);
  }
}
```

7.1.2.7 updateCustomer()

Here is the complete **updateCustomer()** 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 **CSTMR** file (the file may already be open from the **getCustomerDiscount()** method, which is why we test for it already being 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 generated by the methods 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;
}
```

7.1.2.8 updateStock()

Here 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 is decremented and updated in the file.

Any exceptions generated by the methods 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;
}
```

7.1.2.9 writeDataQueue()

Here is the complete **writeDataQueue()** method :

- Create a description of the data queue layout.

  It first creates objects representing the different data types used in the layout. Then it creates an object representing the AS/400 system data queue object. Next it defines the layout of each record in the data queue using the field definitions created earlier.

- Populate the record with the values for the data queue.
- Send the record to the data queue to initiate printing the order.

Any exceptions generated by the methods 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;
}
```

7.1.2.10 getRawDate()

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));
}
```

7.1.2.11 getRawTime()

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));
}
```

7.1.3 Cleaning Up

After an order has been processed, we need to perform some housekeeping 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 run time 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 should be protected, must return void, must not accept arguments, and must throw the **Throwable** exception because it calls the **finalize**() method of its super class.

We first 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. We then close the connection to the AS/400 system and invoke 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 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; it is equivalent to the OrderEntryDDM class discussed previously in this chapter.

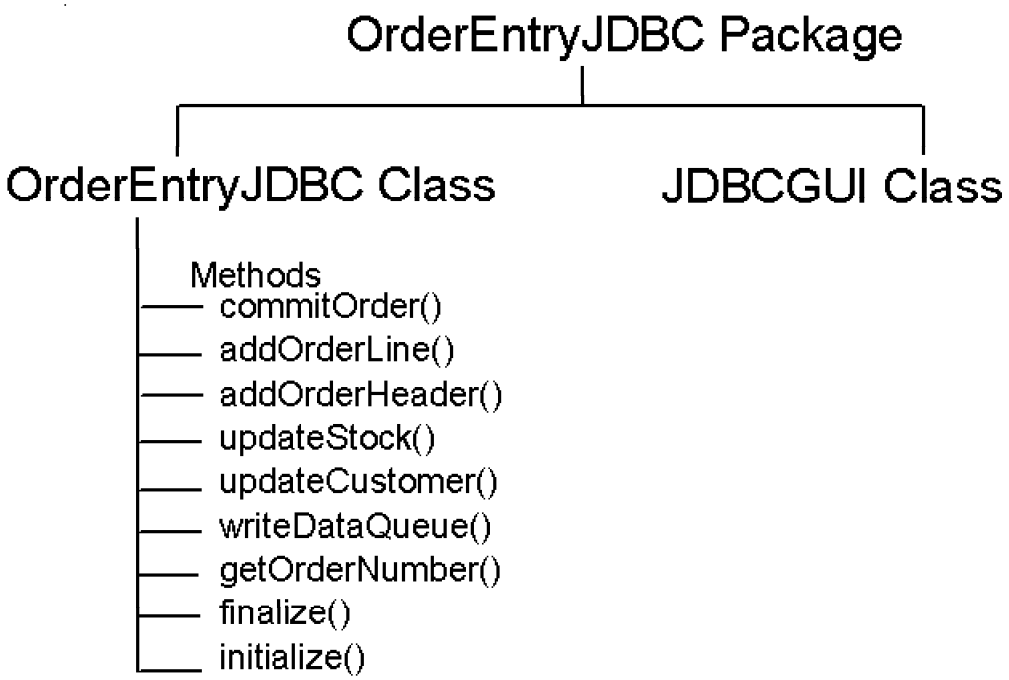


Figure 87. OrderEntryJDBC Class

OrderEntryJDBC is a platform-independent version of the **OrderEntry** class. 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 system Toolbox
- JDBC through the Developer kit

The option we use is determined by which JDBC driver we choose to load. Loading a driver requires a URL providing the connection information. Each of the three drivers requires a different URL.

- JDBC-ODBC - "jdbc:odbc:"
- Toolbox JDBC - "jdbc:as400://"
- Developer kit JDBC - "jdbc:db2:"

JDBC drivers can be loaded explicitly by registering a driver with the driver manager and then connecting to the URL, or loaded implicitly by invoking 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 be used to create

code that can run 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 system toolbox 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 system Toolbox 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.

7.2.1.1 initialize()

Here is the complete JDBC **initialize()** method. This method is invoked 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 is used to describe 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 quite 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.

We then use the class method **Class.forName()** to determine the proper JDBC driver and automatically register and load it. Then we create a connection to the AS/400 system using our properties object.

The next block of code creates a prepared statement for the only SQL statement that is run repeatedly. Using 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;
}
```

7.2.1.2 commitOrder()

The method of committing changed data is altered. A commit request is placed through the connection object rather than 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.");
}
```

7.2.1.3 addOrderHeader()

The only change necessary to this method is replacing the field population and write statement with an SQL INSERT statement. Notice that 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;
}
```

7.2.1.4 addOrderLine()

The only change necessary to this method is replacing 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 resulting 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;
}
```

7.2.1.5 getCustomerDiscount()

This method is similar to the DDM version. We create an SQL statement object and build an SQL query that returns the values 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 retrieved by the query. There may be many rows returned, in which case a loop is used to process 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 appropriate to each database data type. Here we use the **getBigDecimal()** method because the CDCT field is a packed decimal filed.

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 do need to process many rows from the result set, the Java code looks similar to:

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

7.2.1.6 getOrderNumber()

This method has the same structure as the DDM version; however, we need to do some additional processing to allow us to update a row. Updating records can be done 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.

We first 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 containing the next order number. We run the query, fetch the row from the result set, and extract the field value. Then we build an SQL UPDATE statement to increment the order number value and update the database.

And last, 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;
}
```

7.2.1.7 updateCustomer()

Here is another method that uses the positioned update technique 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, the current balance, and the 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;
}
```

7.2.1.8 updateStock()

Again, we use the positioned update technique 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;
}
```

7.2.1.9 writeDataQueue()

No changes are required to this method.

7.2.1.10 getRawDate()

No changes are required to this method.

7.2.1.11 getRawTime()

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. The reader is invited to 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 intended to be invoked from a client front-end and we cannot yet do that because there is no linkage to the client.

Remote Method Invocation (RMI) is the mechanism Java uses to support invoking 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.

Changing a class to support RMI is similar to an alchemist's incantation. You must specify certain things and magic happens!

The rules for making RMI work are:

1. The class must be a subclass of the **UnicastRemoteObject** class.
2. The class must implement an interface that describes the public methods.
3. The interface must be a subclass of the **Remote** class.
4. The interface must describe each public method.
5. The interface methods must throw RemoteException.
6. An RMI registry must be running on the server.
7. An instance of the class must register with the registry.
8. The client and 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

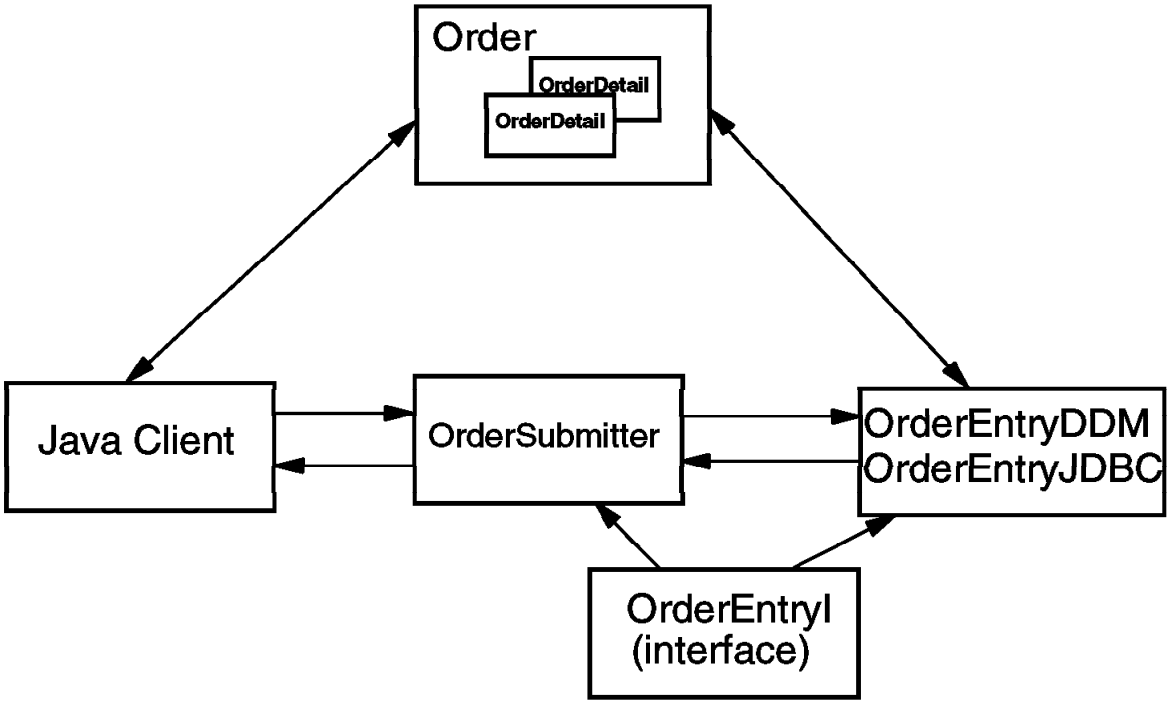


Figure 88. RMI Application Design

We use the Java Remote Method Invocation (RMI) interface to allow the client Java program to interface with the server Java code. The Java client program will interface to the AS/400 program (OrderEntryDDM or OrderEntryJDBC) through a class named OrderSubmitter. An Order object is passed from the client program to the server program.

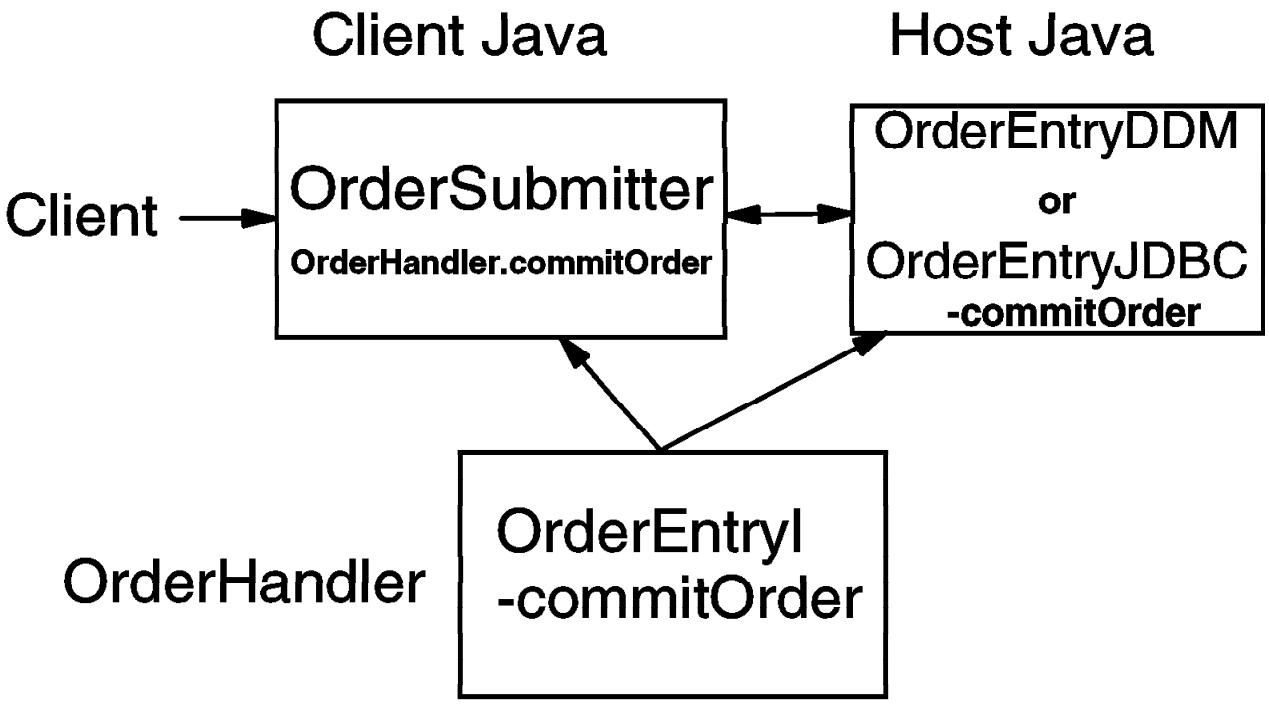


Figure 89. RMI Interface

The public method that we use is called **commitOrder**. It is described in the interface that we implement named **OrderEntryI**. The host Java program implements a method named **commitOrder**. The client programs calls this method through the OrderSubmitter class.

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.

```
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;
}
```

What this block of code does is:

- Create a new security manager.
- Create a new instance of our own class.
- Bind 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. These are created automatically by the **rmic** command. VisualAge for Java can also be used to create these classes.

Once the proxies have been created, we must start the RMI registry. This must be done from the shell. There is a shell command called **rmiregistry** to start the registry.

The RMI registry can be directed 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. This can be accomplished by adding the port number to the **Naming.bind()** or **Naming.rebind()** methods. For example:

```
Naming.rebind("//"+SYSTEM+":55555/OrderEntryJDBC", oeJDBC);
```

that specifies that port 55555 is to be used by the registry.

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 109, we analyzed how the Java client submitted an order. It created buffers of string data that were used as parameters in a distributed program call. The **ProgramCall** class, as well as other classes in the **AS/400 system 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 object, objects cannot be passed as parameters to legacy routines. The objects must be "flattened" or streamed out as byte data.

However, as we saw in Chapter 6, “Migrating the User Interface to Java Client” on page 109, this introduces more complexity to the program. The **Order** and **OrderDetail** classes provided **toString()** methods that helped deal with this situation. Once the server code is implemented in Java, the client's 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's **RMI** architecture. In the previous sections, we saw how to convert some of the legacy RPG code to Java classes. These classes are designed so that they implement the **RMI** interfaces. The client can now invoke 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 as well as 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 invoking the **Naming.lookup()** method. Once these two steps are completed, the application can invoke 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 109.

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 remote method invocation 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 invokes 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 remote method invocation, it must be changed so that it implements **Serializable**. Java's **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 necessary to enable the client's use of **RMI.**

7.3.5 Conclusion

To allow the Java client to interface with the new AS/400 server Java classes, we use remote method invocation (RMI). In this section, we discussed the changes 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. Performance

Java on the AS/400 system results in a different environment from what most AS/400 developers have previously used. First, Java is a truly object-oriented language compared to RPG and COBOL, which are considered to be procedural languages. The development life cycle is then also different.

From a work management viewpoint, there are several new concepts with which the application developer 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.

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 these 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 due to its portability across different platforms, but where portability is less of a concern and higher performance is required, DDM and DPC are widely used to access DB2/400. These techniques will be covered in a later redbook.

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 are being invested to improve Java run-time 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 2 of OS/400.

8.1 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 not only through standardized language constructs, where the Java source can be moved from one system to another, but also at a lower level through bytecode.

Bytecode is machine-independent psuedo-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 bytecode is created. This .class file containing the bytecode can be executed on original system's JVM and can also be transported to another platform and executed on a different JVM.

Executing the bytecode on a JVM involves interpreting the bytecode 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. The bytecode, on the other hand, is completely portable and is isolated from the underlying hardware implementation.

Consider the process that was just described. The program source was "COMPILED" into bytecode, then the bytecode was "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 is first loaded. Because the program object consists of 64-bit enabled RISC instructions, run-time performance is improved significantly without affecting the Java portability qualities of the corresponding bytecode. The traditional ILE environment and the new Java environment are graphically compared in Figure 90.

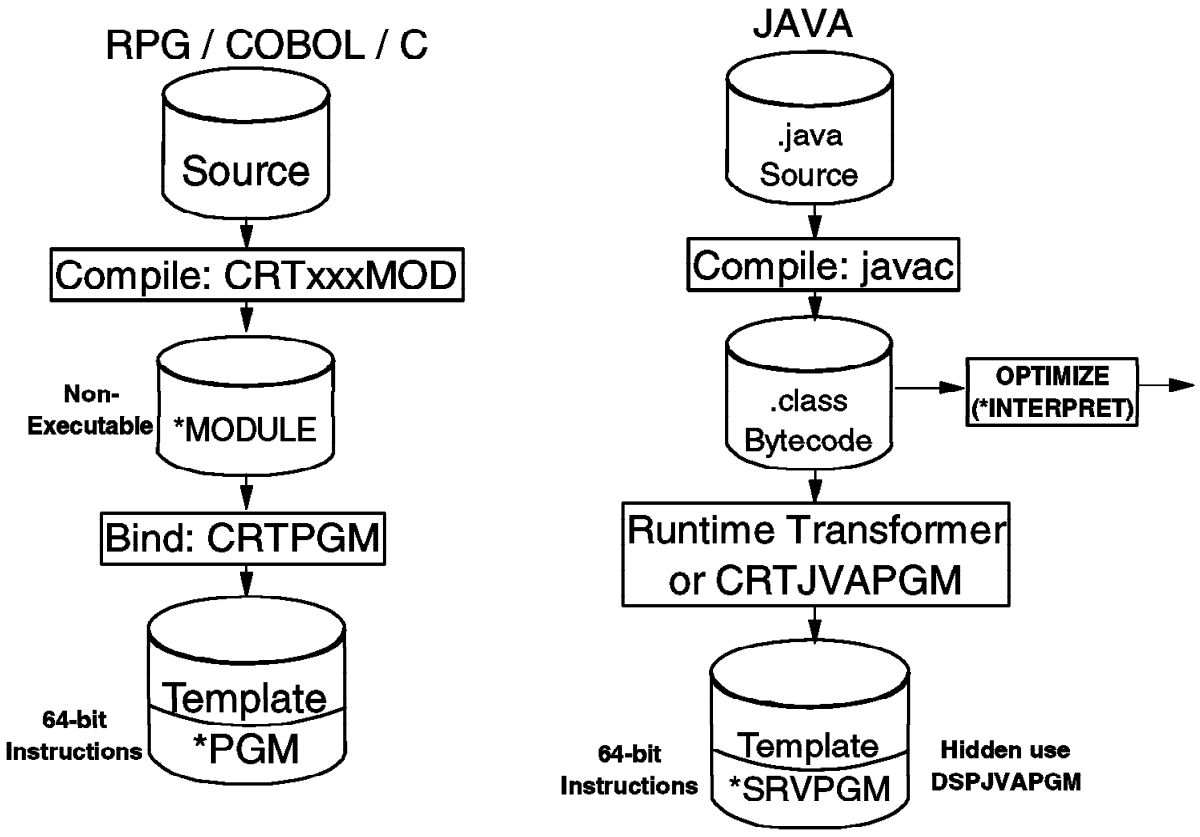


Figure 90. AS/400 Application Development Comparison

8.2 Performance Red Flags

This section discusses methods and techniques, which, when used, may adversely affect performance. Often there is a trade-off between performance and may 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.

8.2.1 Portablility and Interpreted Code

From a performance standpoint, there are two aspects of the Java programming model that 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 bytecode. 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 bytecode is needed for Java to be portable. During the creation of the Java .class file that contains the bytecode, the destination system may not be known. Therefore, the compile can only optimize the bytecode down to the lowest common denominator; that is, it cannot contain any features that are platform specific. The portable bytecode then may not be able to take advantage of a hardware platform's performance strengths.

As far as the interpreted nature is concerned, bytecode level interpretation occurs at a much lower level than the source level interpreters that many programmers are familiar with. In addition, the Java 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).

There are some large efforts being undertaken within the IT industry to optimize the performance of the portable Java bytecode by reducing the inherent slowness of interpretation. The first is the use of Just-in-Time compilers, or JITs. The second is direct compilation, where the Java source is compiled directly into executable programs, bypassing the bytecode level altogether and relying on source code for portability. A third approach is the AS/400 system's Java Transformer, which retains the bytecode portability (allowing them to be served to other platforms) and makes use of *JVAPGM program to perform the actual execution. This is described in detail later in this chapter.

8.3 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 supposedly due to the amount of messages that need to be passed from one class to another. In a typical OO implementation, when a method or procedure from another class needs to be invoked, a message containing the method name is passed to the class.

Note that Java is a full OO language and is capable of implementing OO concepts. To fully utilize 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 instance, instantiating an object is relatively expensive. Using a method that resides in another class (for example, the superclass) results in call

indirection, 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, use the javac -O option when creating the Java class from the Java source. One tradeoff is that the sizes of the class files (as well as the sizes of the transformed Java programs that are eventually created) increase. However, the resultant DASD storage penalty is well worth the improvement in run-time performance.

In addition, before discounting OO Technology and Java, be aware that the trend in computer science is that software development and maintenance costs are increasing rapidly while hardware price/performance continues to improve.

8.3.1 First Pass after Technology Preview

In comparison with other programming languages that are available on the AS/400 system, Java is fairly new not only to the platform but to the entire industry as well. Other languages such as RPG and COBOL have had many years head start over Java in terms of performance optimization. OS/400 V4R2 contains the first full production version of Java for the AS/400 system. Ideally, it should perform as well as the other languages but in reality, that may not be the case.

The industry's performance objective is to have Java perform similar to C++. The probability of this happening in an interpreted environment is small. 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 could eventually access information stored in DB2/400. If this happens, it fits the profile of a commercial environment, albeit with an electronic commerce flavor.

Currently, there is no widely accepted commercial benchmark for Java. The benchmarks that are used (for example, CaffeinMark 3.0) are not commercial benchmarks and deal with program instruction execution rather than system services such as DB access. On the AS/400 system, one of the internal benchmarks used is called the Business Object Benchmark (BOB), which runs a workload similar to Commercial Processing Workload (CPW) but has no persistence; that is, everything is done in memory. Another benchmark is the Commercial Java Workload (CJW). This is also based on CPW and actually implements persistence by accessing DB2/400 using the JDBC driver.

Traditionally, the AS/400 system has been an excellent performer within the commercial environment. Typical RPG, COBOL, or a C commercial application have a performance profile where fewer resources are spent processing in high level language program instructions than in system services such as database processing (this is represented graphically in Figure 91 on page 181).

Figure 91. Traditional Commercial Application

A commercial Java application is expected to have a similar profile, but in this first version, the HLL portion is noticeably higher, implying that more time is spent executing machine instructions compared to performing system activities such as database I/O's.

Maximum performance payback is then achieved by implementing similar coding techniques that have worked so well in traditional languages (for example, keeping files open, logical blocking, and so on).

8.4 AS/400 Java Execution Steps

The following major events occur when a Java program 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
- Loading and verifying the class files
- Actual program interpretation or instruction execution
- Accessing system services

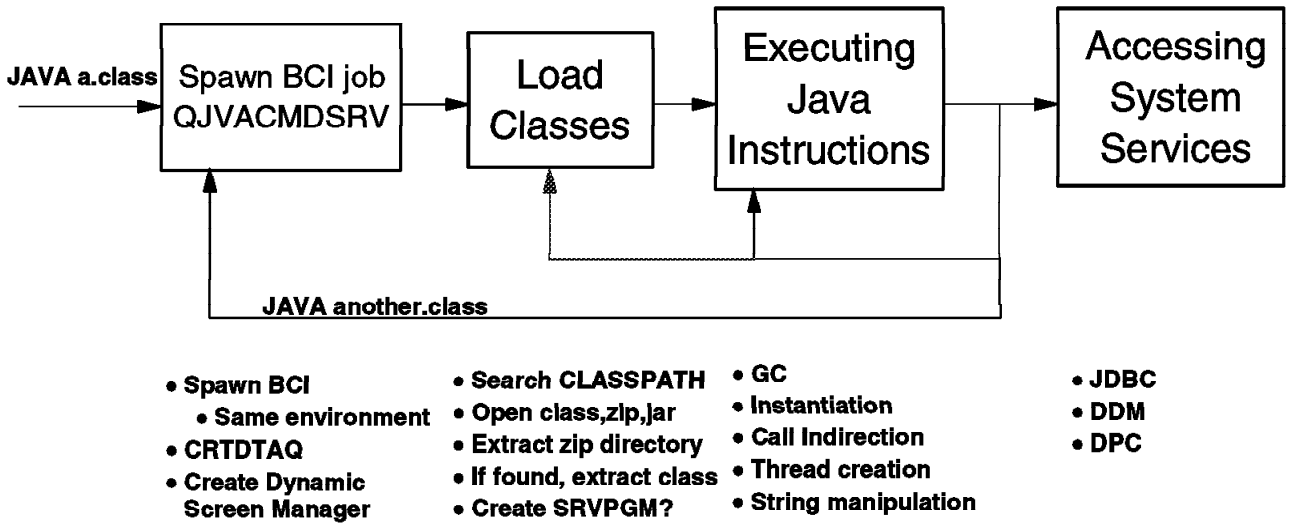


Figure 92. AS/400 Java Execution Steps

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, as well as loading classes where the JVM searches through the CLASSPATH to load the required classes.

Running many short Java programs is not recommended because it may take more time and resources to start up 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.

8.5 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's 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.

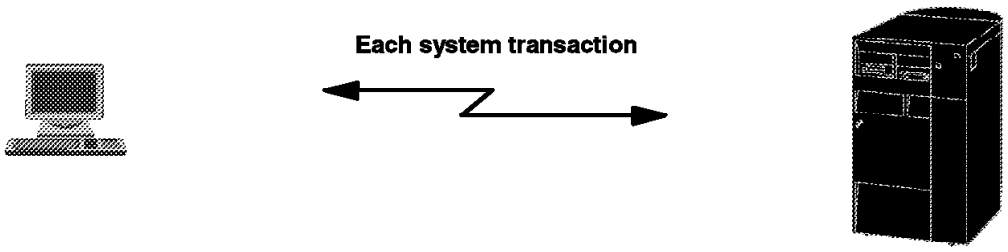


Figure 93. Main Frame Interactive Response

For every transaction, there is usually one conversation between the terminal and the AS/400 system unless certain 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 94 on page 183, we see the response time divided into its individual components

Note: The actual contribution of each response time component may vary depending on the system environment.

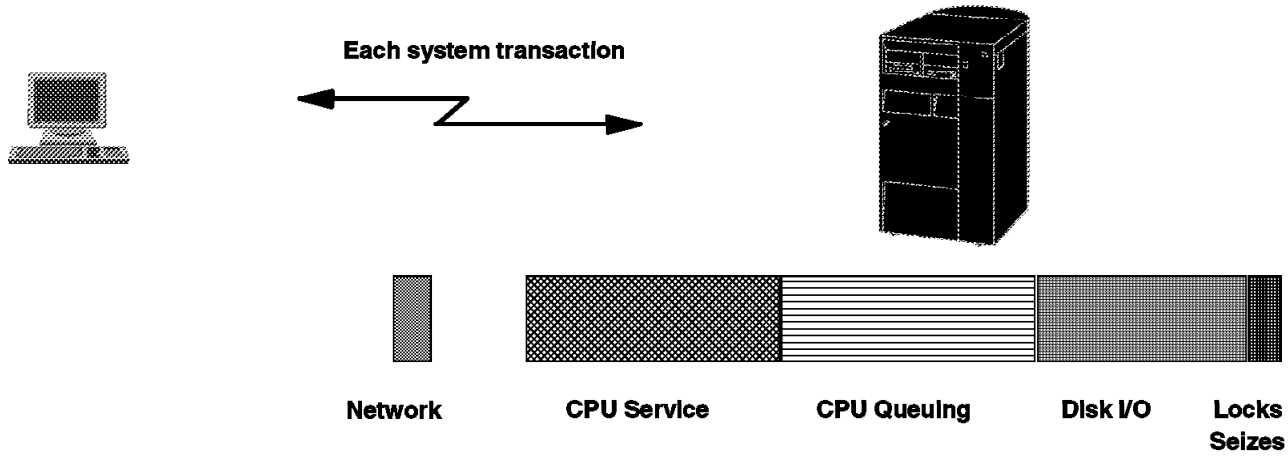


Figure 94. Main Frame Interactive Response Time Components

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 gets busier (higher CPU utilization). The effect is exponential and the queueing factor is approximated by:

```
Q(f)= 1 / (1 - (u**n))
            where u the the utilization of the system between 0 and 1
            and   n is the number of CPU's
```

Since queuing is a function of CPU utilization, reducing the utilization by improving the efficiency of high volume transactions will reduce 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 but generally, it is the synchronous ones that 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 functions such as expert cache, Set Object Access (SETOBJACC), pool sizes, and so on can also affect this number of physical I/O operations. Generally, 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 disks 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*, SC41-5306-01.

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 occuring and if they do, for short times. Excessive seizes or locks usually indicates poor application design. AS/400 Performance Tools, 5769-PT1, can be used 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 95 shows a typical client/server transaction.

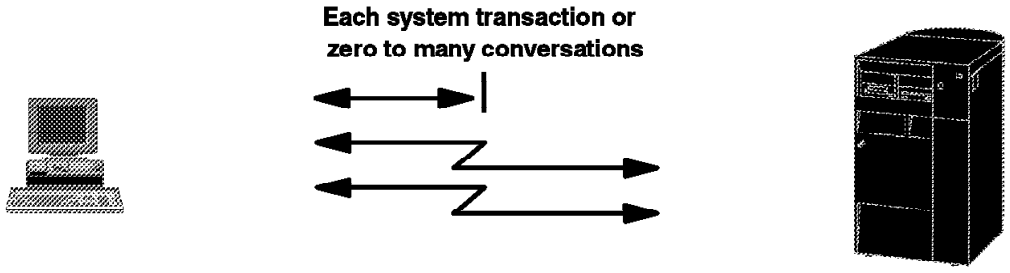


Figure 95. Client/Server Response

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:

- The client
- The server (AS/400 system)
- The network

The response time components of each of these areas are shown in Figure 96 on page 185.

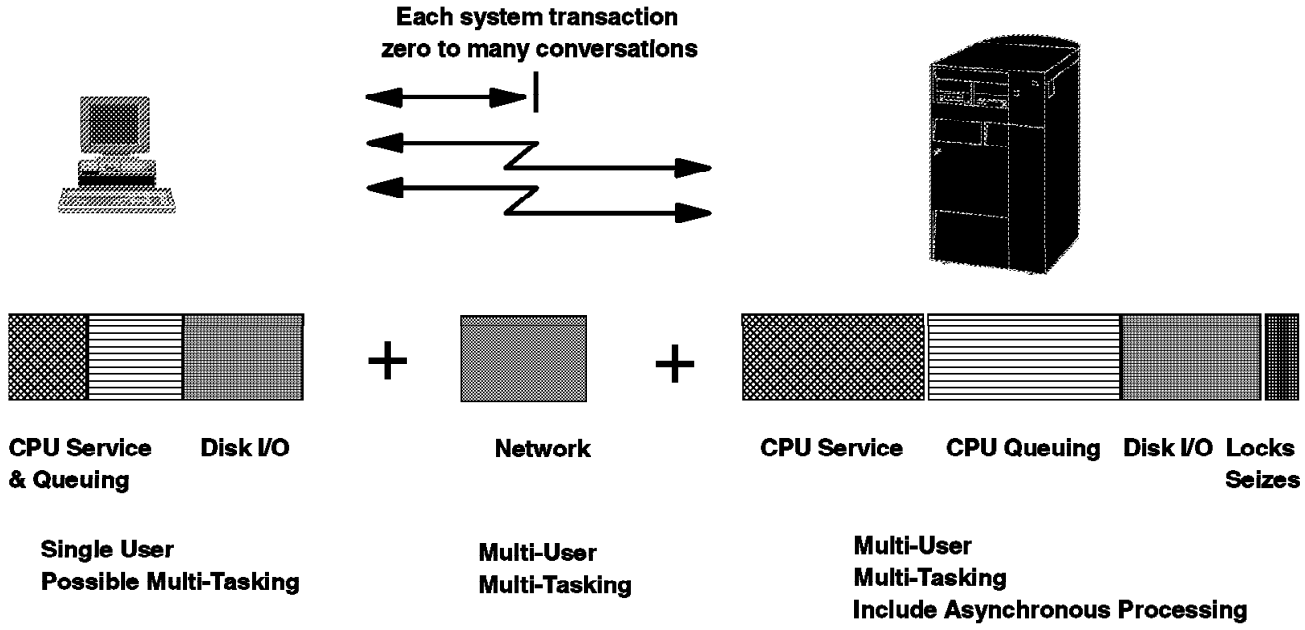


Figure 96. Client/Server Response Components

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 97 on page 186 shows some that are more commonly used (with some AS/400 access methods overlayed).

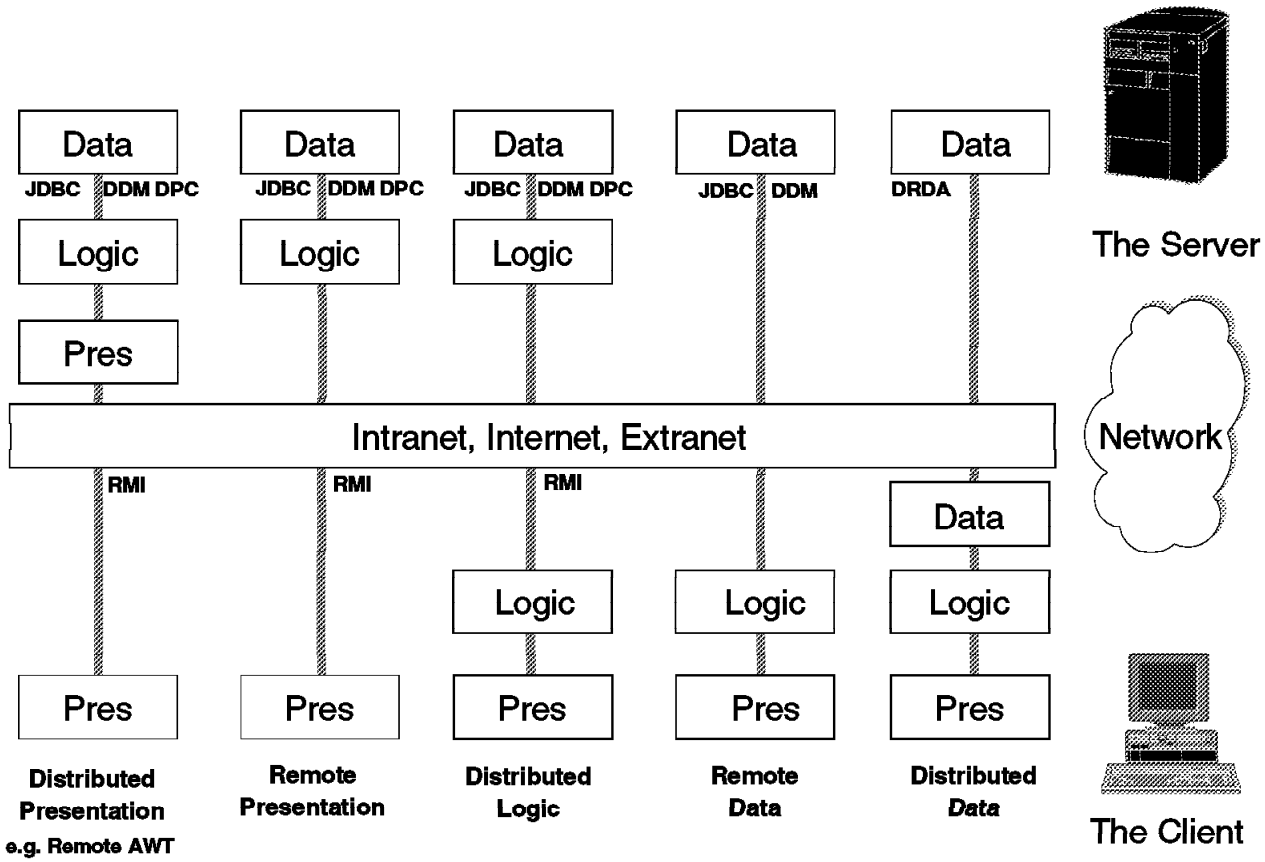


Figure 97. AS/400 Java and the Client/Server Model

In all five, 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 Window Toolkit (AWT) classes.

Logic: Involves business logic such as processing and enforcement of business rules.

Data: Access to the database.

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 leftmost scenarios, 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 *Accessing the AS/400 System with Java*, SG24-2152. The Distributed Data scenario is covered by many books but a good reference is *Distributed Database Programming*, SC41-5702-01.

In these three client/server Java scenarios, user interaction is through a graphical user interface (GUI) and is not based on the 5250 data stream. All of these Java scenarios are excellent candidates for AS/400e servers, which provide the best price/performance for non-5250 applications.

Under Distributed Presentation, all three logical tiers of the Java application run on the server. However, because the Abstract Window Toolkit (AWT) 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 is built over Java's Remote Method Invocation (RMI) standard. RMI provides distributed objects support within Java.

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, 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 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 is expected to be the facility for invoking remote methods.

8.6 Addressing Performance

All performance situations can be addressed in these four major ways (see Figure 98 on page 188). Any of these can provide a significant performance improvement. Conversely, if any of these are not properly implemented, performance can be negatively affected. For instance, the best hardware can be installed, but if the system is not tuned properly, there is a good chance that performance will suffer.

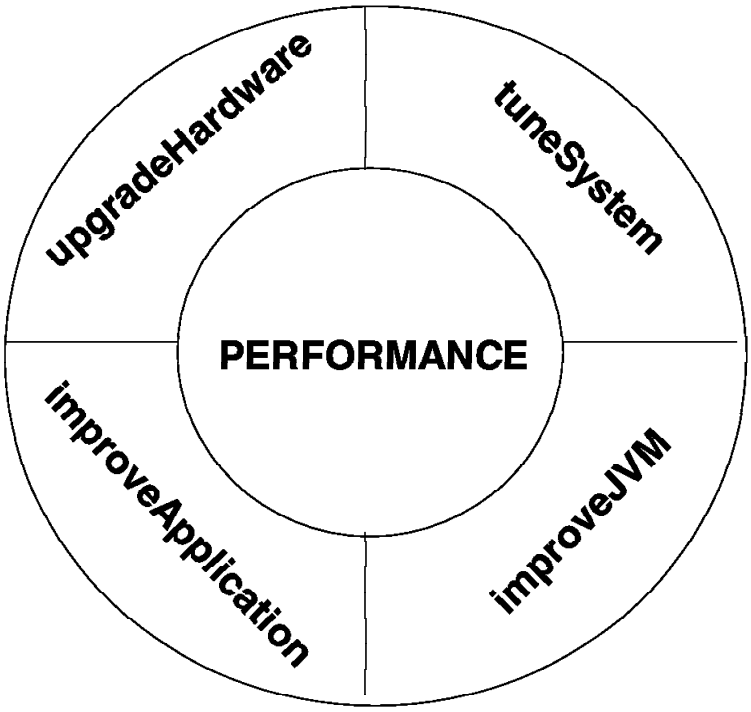


Figure 98. Performance Factors (Donut)

The criteria generally used to determine the most appropriate approach are:

- Whether the option is available or not
- The cost of implementation versus expected improvement
- How long 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 (and memory) upgrade should provide improvements. Similarly, if disk I/O service time is excessive, adding more disk arms or faster disk arms may reduce 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 environment), 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 run-time support can be expected in future updates as this is the first production version of the AS/400 JDK.

8.7 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.

There are three ways of running Java programs on the AS/400 system:

1. Through the JAVA or RUNJVA command, which makes use of the BCI job QJVACMDSRV.
2. Through the QSHELL interpreter, wherein 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.
3. Use the Unity interface through Client Access/400 by double-clicking on a class file within Operations Navigator (this method is not discussed here and is mentioned only for clarity).

Each Java program that is run results in four major execution steps:

- Initiate the Batch Immediate job.
- Search and load classes.
- Java instruction execution.
- Access system services.

8.7.1 Initiating the Batch Immediate Job

Each time a Java program is run using the JAVA or RUNJVA commands, a Batch Immediate (BCI) job is spawned by the job that ran the command. Job initiation such as that encountered in a SBMJOB or signing on to an interactive session incurs considerable overhead.

Spawning a BCI job is not as costly as a full job initiation because it uses the same environment as the initiating job (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.

However, frequent initiations should still be minimized for reasons that will become clear 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 instance, do not run a Java class from an RPG program to perform a date conversion routine.

This does not limit the use of Java to a batch environment, however. It fits very well into the interactive client/server scenarios previously described. With RMI, the server classes are started and remain active in a "Never-ending" type mode.

8.7.2 JAVA or RUNJVA Commands = QJVACMDSRV BCI Job

Whenever a Java program is invoked with the RUNJVA or JAVA commands, the following steps occur:

- A BCI job named QJVACMDSRV is spawned under the same work management environment; that is, 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 99.

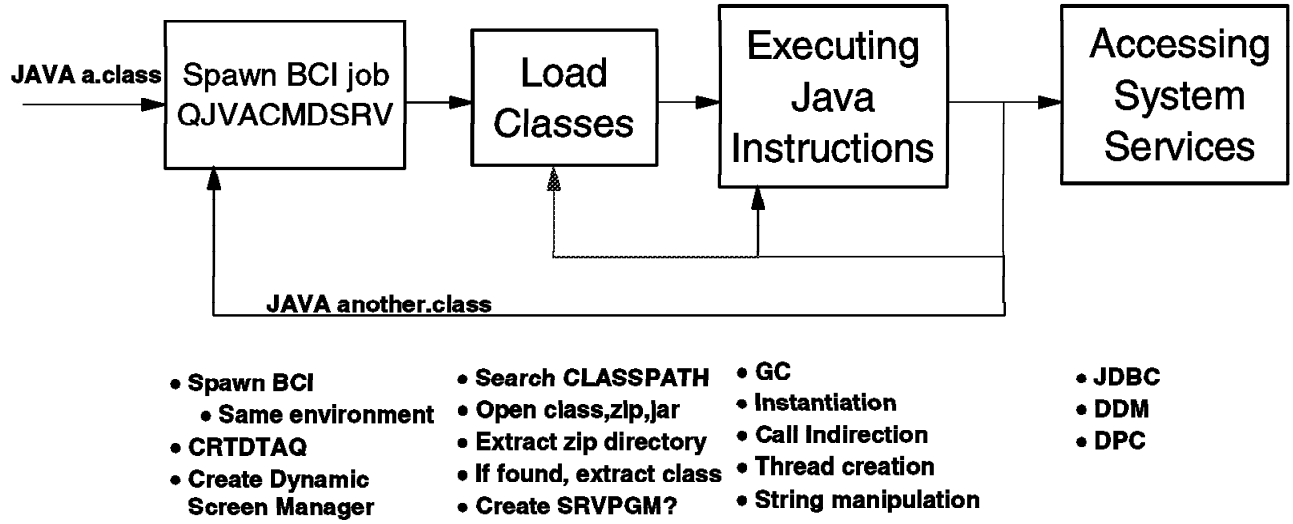


Figure 99. Java Run Time - Java Major Steps

Figure 100 on page 191 shows a **Work with Active Jobs** display where interactive job QPADEV0007 ran a Java program. Note its entry under the Function column, **CMD-JAVA**. This program was initiated using the following **JAVA** command:

```
JAVA CLASS(a970107e.RMIExample.HelloImpl) CLASSPATH('.')
```

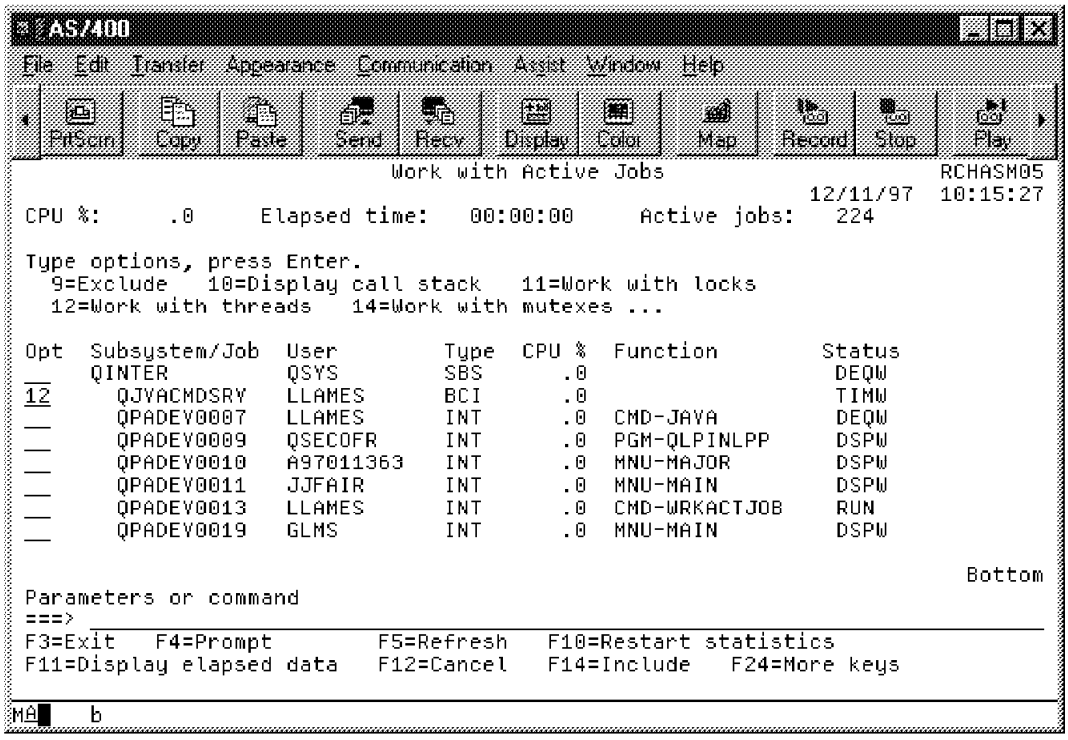


Figure 100. WRKACTJOB - INT and BCI Jobs

This 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 034040/LLAMES/QJVACMDSRV started on 12/11/97 at 10:14:47
in subsystem QINTER in QSYS. Job entered system on 12/1/97 at 10:14:47."
```

To 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), Java classes should only be called or used when they are relatively long running.

8.7.3 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 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.

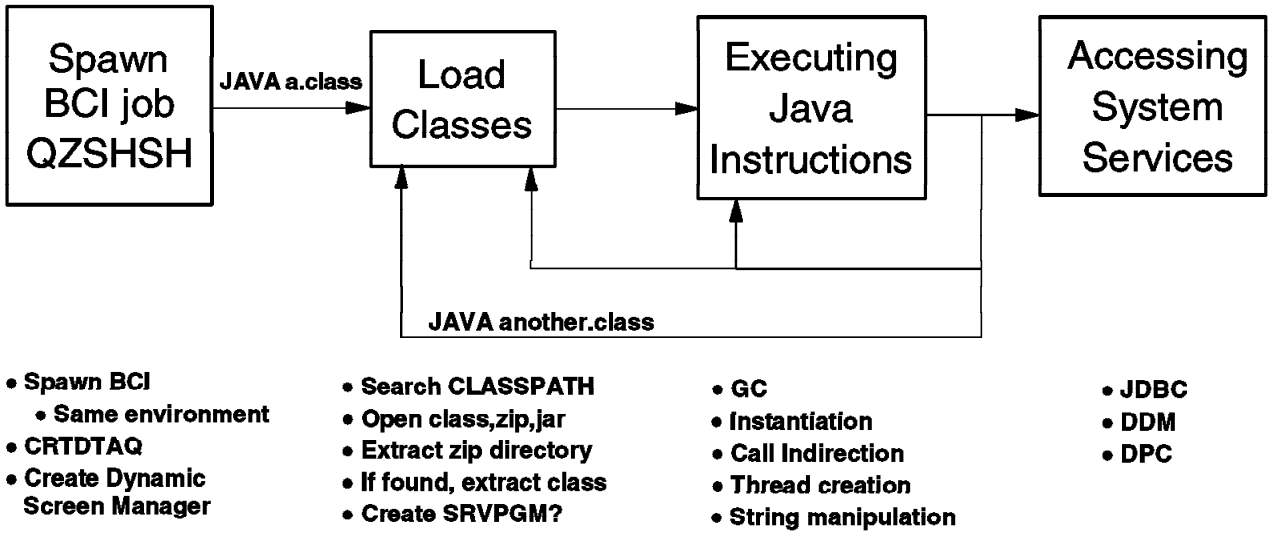


Figure 101. Running Java from QShell

The spawning of a BCI job QZSHSH is similar to that of QJVACMDSRV. In Figure 102, you can see the BCI job that was spawned from the interactive job running CMD-QSH.

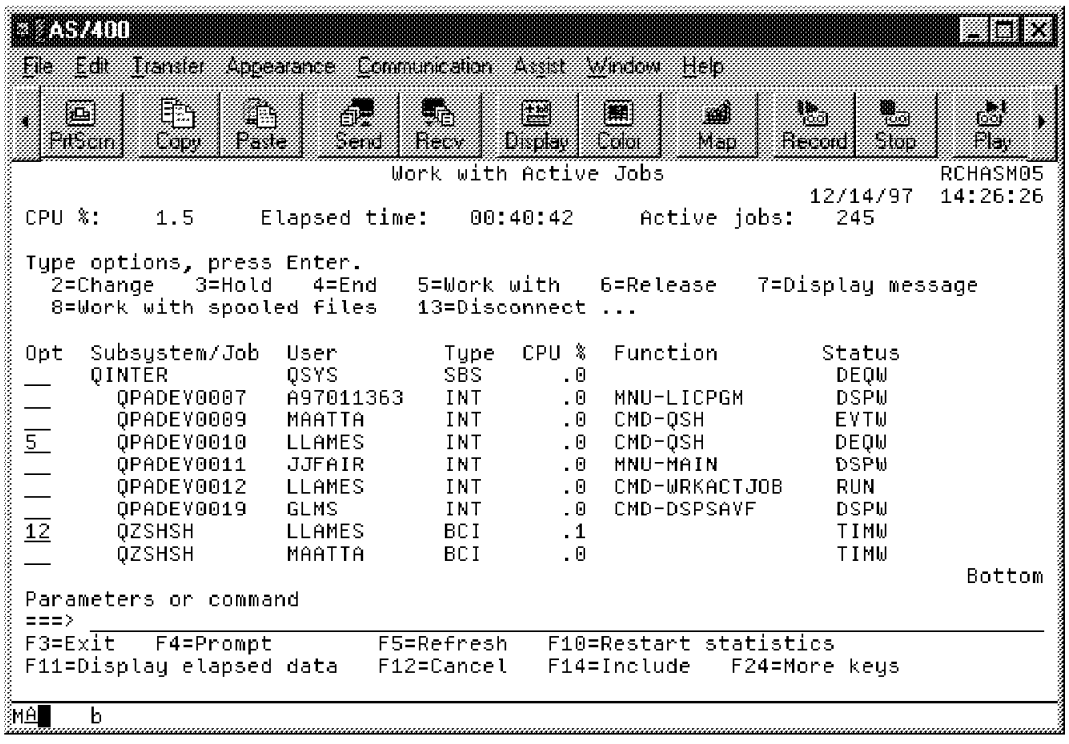


Figure 102. WRKACTJOB - Java QShell

When running under a QSHELL script, it is also possible to pre-start jobs to improve execution. The pre-start job entries should be added to the subsystem from which the QSH or STRQSH command is run. The preceding example uses subsystem QINTER. An example is:

ADDPJE SBSD(QSYS/QINTER) PGM(QSYS/QP0ZSPWP) INLJOBS(20) THRESHOLD(5) ADLJOBS(5) JOBD(QGPL/QDFTJOBD) MAXUSE(1) CLS(QGPL/QINTER)

To run a QSHELL session with a QSHELL script, use the QSH or STRQSH command with the parameter CMD(textfile_in_IFS_syntax).

For more information about QSHELL, refer to the *QSHELL Interpreter Reference*, which can be found at *http://as400bks.rochester.ibm.com*.

8.7.4 Searching and Loading Classes

When running Java under OPTION(*VERBOSE), many messages that relate to class loading are generated:

```
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** is loaded and starts executing. After this, other required classes are loaded in a "lazy load" mode or as needed.

One main difference between CRTJVAPGM . . . OPTIMIZE(30) and OPTIMIZE(40) in the Direct Execution environment is that the latter is meant for fastest execution and therefore, more classes are pre-loaded. If the JVM cannot determine if a particular class is executed, it is loaded to ensure that execution is not delayed if it is indeed used.

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.

Classes are searched for in the order at which directories are specified in the class path.

There are several things that can be implemented to improve the class search and load times:

- **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). A 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 (that is, they should have the "hidden" service 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 follows:

```
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 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 environment, the Create Java Program (CRTJVAPGM) command should be used to create the service program. Using the OPTIMIZE(40) parameter enhances execution performance but reduces debug capabilities such as changing the program variables.

Jar or zip files without service programs have long class search and load times. When a class has to be loaded from the zip file, the zip file is opened, its directory searched for the class, and a non-persistent service program has to be created for the class to be executed in OPTIMIZE(10) mode. Thus, not only is the class loading slow, but so is the class execution because of the low degree of optimization. Because the service program is non-persistent, this process is performed every time the class is run.

A class that is not stored in a zip or jar file also needs its service program to execute. If it has not been put through CRTJVAPGM, a persistent service program is created internally the first time the class is run. It then runs subsequently under the OPTIMIZE(10) level by default.

- **Minimize the directory search:**

  Searching through directories for a Java class can be likened to searching through a library list for a non-qualified program name. Similar recommendations then 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 result in many misses.

- **Store related Java classes in jar or zip with service program:**

  The best scenario for searching and loading classes is to have them in a jar or zip file that has a service 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 results in frequent changes to the zip or jar files. It is generally applied when the classes are relatively static.

In order of performance, with the fastest performer first, the following common environments are encountered:

- Zip or jar file with service program created through CRTJVAPGM

- Regular class directory search
- Zip or jar file without service program (WORST)

8.8 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, see the AS/400 manual *Developing Multi-threaded Applications*, SC41-5436.

Thread support actually provides a way of 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 through 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 103 shows the threads of a QJVACMDSRV BCI job:

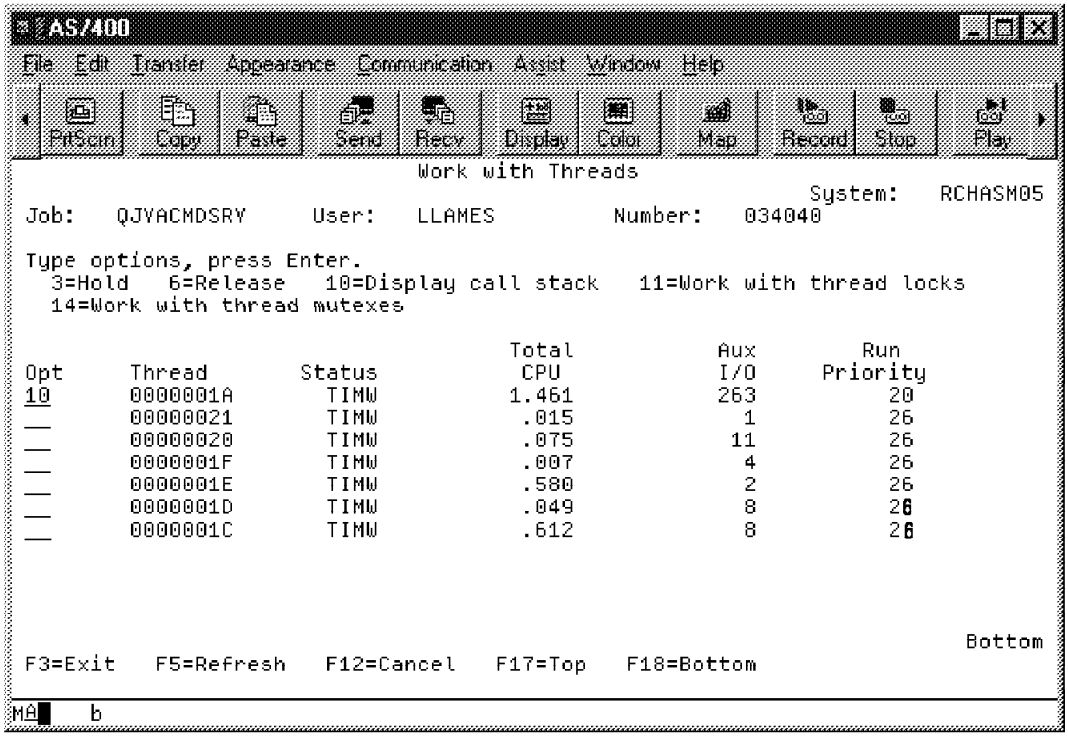


Figure 103. Work with Threads

The introduction of native thread support adds a new twist to system tuning.

8.8.1 Initial Thread

In the Java environment, an initial thread is created to perform the initial service functions such as the initial piping for STDIN and STDOUT. It also handles interrupt requests when the BCI job that it is running under is ended and then performs clean up functions. When this thread is doing nothing, it sits at TIMW status.

This thread has the lowest thread number assignment as it is the first one that is started.

8.8.2 Run Priorities

The initial thread 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. Note that in the preceding example, the first thread is the initial thread with a run priority of 20.

When the actual Java main method starts, the initial thread spawns a new thread whose run priority is lower by a relative amount, usually 5.

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, the higher the run priority. 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 equation as follows:

AS/400 Thread Priority = (20 + (11 - 5))

If the JAVA command is run in batch mode such that 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 the 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 *CLS. Its priority does not change if it has been sitting at a prolonged wait state or TIMW status. Hitting 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. For example, if the Java application is used for online transaction processing and there are interactive RPG 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). Recall that the ranges are as follows:

- 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, 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 should be the same as a Java NORM_PRIORITY, or 5. In this example, they should be running at an AS/400 run priority of 26.

With GCPTY(10), the garbage collection threads should be running at a Java priority of 1, which is the lowest priority. With the BCI job 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.

8.8.3 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 (recall 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.

8.8.4 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 is established with the CPUTIME parameter on the CRTCLS command or the CHGCLS command.

8.8.5 PURGE

If the job is multi-threaded (a BCI job is 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.

8.9 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 compile options, and from a coding standpoint, there are certain features that need to be minimized. Most of these are also applicable to other platforms.

8.9.1 Compile Options JAVAC

Java source is compiled into bytecode using the javac command. One way is to do the compilation on a client, then move the generated bytecode to the AS/400 Integrated File System. Another way is to run the command on the AS/400 inside QSHELL.

- **javac -O:**

  An object-oriented design is generally more expensive than a functionally equivalent procedural design because of call indirection (that is, when a method is requested, a search has to be done up the class hierarchy). Generally, the deeper the class hierarchy, the higher the overhead.

  The -O option on the AS/400 system does method inlining. Any method that is final (private, static, or protected) is a candidate. Run-time performance is improved because method call overhead is reduced.

  Because a final method cannot be sub-classed, it is possible to determine which method is used. A copy of the method's bytecode is then included in the caller (inlining) so that the method execution is much faster. The tradeoff is that the class size will increase.

- **javac -g:**

  This option is needed so that when debugging the class, variables can be changed.

8.9.2 Explicit Java Transformer CRTJVAPGM (Optional)

Using the Create Java Program (CRTJVAPGM) command on a class file results in a "hidden" service 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.

The program is created by TRANSFORMING the class file's bytecode into an optimized program object attached to that class. This program object contains fully compiled 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 but with maximum debug capability.

8.9.3 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 .class versus the .jar and .zip files.

- **.class files:**

  If the class file does not have a hidden 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:**

  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 8.7.4, “Searching and Loading Classes” on page 193).

  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 a JVM. It is then imperative, as mentioned in Section 8.7.4, “Searching and Loading Classes” on page 193, to run the CRTJVAPGM Transformer on .jar and .zip files.

8.9.4 Optimization Levels

Optimization is usually a tradeoff between performance versus size and debug capabilities. For maximum debugging on the AS/400 system, there are three requirements:

1. JAVAC -g so that debug information is included in the class.
2. The .java and the .class file should be in the same IFS directory.
3. The class should be running in *INTERPRET mode. This means that the CRTJVAPGM . . . OPTIMIZE(*INTERPRET) should first be run to ensure it will run under bytecode interpretation mode. Another way is to DLTJVAPGM, then use the JAVA or RUNJVA command with OPTIMIZE(*INTERPRET).

Otherwise, when the application is stable, it is best to run at the highest optimization level of 40 for best performance.

8.9.5 Automatic Garbage Collection

Unlike most other JVM implementations, the AS/400 automatic garbage collection is not Stop & Copy, where other threads stop as garbage collection runs. The AS/400 JVM usually performs garbage collection 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 is 2048KB. This specifies the initial size of the garbage collection heap before garbage collection will start. Setting too low a value causes garbage collection to start on small programs, which is why the default should be kept the minimum size.

- **Maximum size:**

  The keyword for this parameter is GCHMAX. The default is 32 768KB. This specifies the maximum size that the garbage collection heap can grow to prevent runaway programs from consuming all of the available storage.

  In a balanced application, this limit should not be hit. If it does, garbage collection, which normally runs asynchronously, stops all threads while garbage collection takes place. The effect of such an occurrence is noticeable. That is why the smallest value should be 32 768.

  The next parameters, frequency and priority, are used to ensure that garbage collection occurs frequently enough and gets enough cycles to perform cleanup so that this maximum heap size is not hit.

- **Frequency:**

  The parameter keyword is GCFRQ. The default is 50 and a lower integer value causes garbage collection to run less frequently. A value higher than 50 causes it to run more frequently.

- **Priority:**

  The parameter keyword is GCPTY. This is described in Section 8.8.2, “Run Priorities” on page 196.

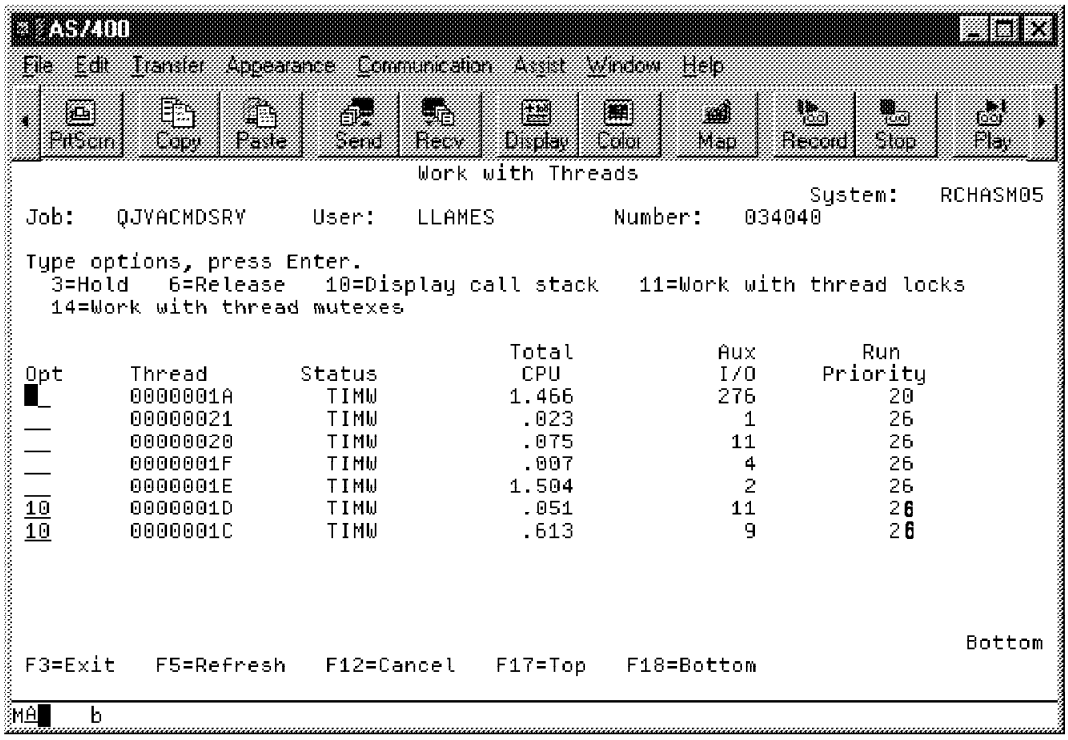


Figure 104. Garbage Collection Threads

There are two threads associated with garbage collection. They are the ones with the next higher thread numbers to the initial thread with a gap of one hexadecimal increment. They are marked with option 10 in Figure 104. Their functions are:

- Collector Garbage Collection Thread - locates objects no longer being referenced.
- Finalize Garbage Collection Thread - When an object collected by the garbage collection, this garbage collection thread invokes the FINALIZE method.

Appendix A. Example Programs

The Java client programs and the AS/400 programs and libraries used in this redbook are available to be downloaded through the Internet. These examples were developed using VisualAge for Java Enterprise edition. OS/400 V4R2 or later is required. The following components are available:

- AS/400 RPG code
- AS/400 databases
- AS/400 Java code
- Client Java code

Important Information

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 Web Site

To use these files, you must download them to your personal computer from the Internet Web site. A file named **README.TXT** is included. It contains instructions for restoring the AS/400 libraries, the VisualAge for Java examples, and run-time notes.

The URL to access is: **www.redbooks.ibm.com**

Click on Downloads and then select directory **SG242163**. In the SG242163 directory, click on download.htm.

Appendix B. Java/400 V4R2M0 PTF List

When installing Java/400, make sure that you have the most current PTF Cumulative Tape. Below is a list of the known fixes as of January 1, 1998. Please review this list of PTFs to make sure that they have been applied. Be aware that some may have been superseded.

B.1 SLIC PTFs Needed for Java 5769-999

```
MF17593 Fix invalid NumberFormatException. No IPL required.
MF17608 GC abnormally terminates with 4300/0100 vlog. No IPL requred.
MF17612 Handle JAR files correctly
MF17659
MF17622 Incorrect exception location reported from interpreter for
        invoke interface bytecode.
MF17661
MF17667 Pre-req for MF17671
MF17671 Fixes to assure that specfied interface is implemented and
        that final class isn't overridden
MF17674 OX problem with 4 byte signed SRL (Shift Right Logical)
        operation). IPL is requred.
MF17693 (superseded) Raise correct exception (VerifyError) in some
        cases during class loading
MF17722 Fixes for Threads
MF17735 Fix verifier bug for ldc2_w
MF17739
MF17741 Fix for freeMemory
MF17742 Fix for java_lang_SecurityManager
MF17743 (superseded) and MF17744 (co-req) Fixes for authority checks on
        invoke interface bytecode
MF17760 (superseded) The penultimate invoke interface fix
MF17774 Fix for JavaSerial
MF17781
MF17783 Pre-req for MF17784
MF17784 Fix verifier missing invalid bytecodes & empty methods
MF17789 Fix memory leak
MF17791 Object signature missing semicolon not diagnosed.
MF17793 Fix memory leak (requires MF17789).
MF17798 Evaluating uninitialized variables.
MF17810 Detect invalid bytecodes
MF17816 Verifier Does Not Reject Bad Initializer const
MF17833 Java Format Errors Should be Verify Errors
MF17852 Wrong Exception Signaled From Java Pgm
MF17861 Debug fixes
MF17882 Exception fixes
MF17907
MF17908 Exception fixes
MF17921 Exception fixes
MF17922 Thread fixes
MF17955 Java Check for ()V Signature on clinit
MF17997 Additional verify checks
MF17998 (superseded) Java Verify Error Thrown for Final Static
MF18001 Translate time failures
MF18021 Debug fixes
```

```
MF18022 OSP Back off Floating Point Change in Java Thread
MF18029 (Supersedes MF17852) Numeric overflow in converting some
        floating-point numbers
MF18031 Suppress verify errors on aastore
MF18032 Unpredictable Failures Running Java Direct Ex
MF18043 (superseded) Add Access Checking for Protected Members
MF18048 (Supersedes MF17774) NullPointerException in ObjectInputStream
MF18055 NullPointerException when using reflection invoke with a null parm
MF18067 (superseded) Fix problems with JNI interfaces RegisterNatives
        and UnregisterNatives
MF18083 Null Ptr Exception Not Thrown
MF18088 Assorted Verification Errors
MF18101 (Supersedes MF18043) Public Members of Non-Public classes
MF18106 (Supersedes MF18048) Object access fields returned incorrectly
MF18108 Array classes should inherit public access authority from base
        class
MF18119 Debug fixes
MF18135 ASCII/EBCDIC name clash causes binder failure
MF18152 Java Threads Need to Throw the StackOverFLowError
```

B.2 XPF PTFs Needed for Java 5769-SS1

```
SF45172 Generic Terminal support fix.
SF45235 Qshell version of Java utilities syntax fixed
SF45383 Qshell Interpreter fixes QSH CL command failed and QSH CMD(xxx)
        CL command inheriting current working dir
SF45407 database fix for a thread problem with the JDBC CLI -
        implementation that causes the database connect to fail
SF45576
SF45588 QSHELL fix for having random newlines inserted in your-
        system.out output to the QSHELL screen
SF46056 Debug fixes.
```

B.3 JV1 PTFs Needed for Java 5769-JV1

```
SF45202 ICONV thread safe use by JV1
SF45238 Shell utility syntax fixes
SF45298 Fix to translation in QjvaLibPrc
SF45300 Fix for file read errors in batch
SF45438 Fix QJVANET *SRVPGM to *OWNER
SF45439 Fix for error when class path name too large
SF45450 Retrieve txt msg for NLS, not hardcoded English
SF45485 298 Op Nav RUN function fails with -secure invalid option
SF45524 java code for javah, javap
SF45610 Fix for DSPJVAPGM
SF45700 Java in batch
SF45707 Hursley 1.1.4 JDK java.zip and sun.zip
SF45861 STDOUT/STDERR truncated for DBCS
SF46084 JDBC SQL0842 connection already exists
SF46135 RAWT i/o exception in OS/2
SF46199 Fix for canonPath native method
SF46200 Fix for not deleting shared memory segments
SF46210 Fix for null spool files in batch
SF46296 RAWT caused err in javac
SF46353 bindOwnVirtual and bindOwnSpecial used by opt 40 only
```

B.4 Miscellaneous Fixes

Also review fixes for System Threads and System Debugger as there may be PTFs that need to be applied to support these system functions.

Appendix C. Java Source Code Samples

This appendix shows samples of small Java programs. They are also found on the Internet and download instructions are given in Appendix A, “Example Programs” on page 203. The code samples are distributed on an "as is" basis, and as such IBM makes no guarantees nor warranties for correctness or quality.

C.1 checkToolbox Java Program

```
////////////////////////////////////////////////////////////////////////////
//
// Install/Update example.  This program uses the AS400ToolboxInstaller class
// to install and update the AS/400 Toolbox for Java package on the workstation.
//
// The program checks the target path for the AS/400 Toolbox for Java package.
// If the package is not found, it installs the package on the worksatation.
// If the package is foundc it checks the source path for updates.  If
// updates are found they are copied to the workstation.
//
// Command syntax:
//    checkToolbox source target
//
// Where
//    source = location of the source files.  This name is in URL format.
//    target = location of the target files.
//
//
//
////////////////////////////////////////////////////////////////////////////
//
// This source is an example of AS/400 Toolbox for Java "AS400ToolboxInstaller".
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantablility and fitness for a particular purpose are
// expressly disclaimed.
//
// AS/400 Toolbox for Java
// (C) Copyright IBM Corp. 1997
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////////

import java.io.*;
import java.util.*;
import java.net.*;
import AS400ToolboxInstaller;
```

Figure 105. checkToolbox Program (Page 1 of 4)

```
public class checkToolbox extends Object
{
   public static void main(String[] parameters)
   {
      System.out.println( " " );

      // Continue with the install/update only if both source and target
      // names were specified.

      if (parameters.length >= 2)
      {

      // The first parameter is the source for the files, the second is the target.

         String sourcePath = parameters[0];
         String targetPath = parameters[1];

         boolean installIt = false;
         boolean updateIt  = false;


         // Created a reader to get input from the user.

         BufferedReader inputStream = new BufferedReader(new InputStreamReader(System.in),1);

         try
         {
            // Point at the source package.  AS400ToolboxInstaller uses the URL
            // class to access the files.

            URL sourceURL  = new URL(sourcePath);



            // See if the package is installed on the client.  If not, ask the user
            // if install should be performed at this time.

            if (AS400ToolboxInstaller.isInstalled("ACCESS", targetPath) == false)
            {
               System.out.print("AS/400 Toolbox for Java is not installed.  Install now (Y/N):");

               String userInput = inputStream.readLine();

               if ((userInput.charAt(0) == 'y')   ¦¦
                   (userInput.charAt(0) == 'Y'))
                  installIt = true;
            }
```

Figure 106. checkToolbox Program (Page 2 of 4)

```
// The package is installed.  See if updates need to be copied from the
// server.  If the target is out of data ask the user if update should
// be performed at this time.

else
{
   if (AS400ToolboxInstaller.isUpdateNeeded("ACCESS", targetPath, sourceURL) == true)
   {
      System.out.print("AS/400 Toolbox for Java is out of date.  Install fixes (Y/N):");

      String userInput = inputStream.readLine();

      if ((userInput.charAt(0) == 'y')   ¦¦
          (userInput.charAt(0) == 'Y'))
         updateIt = true;
   }
   else
      System.out.println("Target directory is current, no update needed.");
}



// If the package needs to be installed or updated.

if (updateIt ¦¦ installIt)
{
   // Copy the files from the server to the target.

   AS400ToolboxInstaller.install("ACCESS", targetPath, sourceURL);



   // Report that the install/update was successful.

   System.out.println(" ");

   if (installIt)
      System.out.println("Install successful!");
   else
      System.out.println("Update Successful!");



   // Tell the user what must be added to the CLASSPATH environment
   // variable.

   Vector classpathAdditions = AS400ToolboxInstaller.getClasspathAdditions();

   if (classpathAdditions.size() > 0)
   {
      System.out.println("");
      System.out.println("Add the following to the CLASSPATH environment variable:");

      for (int i = 0; i < classpathAdditions.size(); i++)
      {
         System.out.print("  ");
         System.out.println((String)classpathAdditions.elementAt(i));
      }
   }
```

Figure 107. checkToolbox Program (Page 3 of 4)

```
            // Tell the user what can be revmoed from the CLASSPATH environment
            // variable.

            Vector classpathRemovals = AS400ToolboxInstaller.getClasspathRemovals();

            if (classpathRemovals.size() > 0)
            {
               System.out.println("");
               System.out.println("Remove the following from the CLASSPATH environment variable:");

               for (int i = 0; i < classpathRemovals.size(); i++)
               {
                  System.out.print("  ");
                  System.out.println((String)classpathRemovals.elementAt(i));
               }
            }
         }
      }

      catch (Exception e)
      {
         // If any of the above operations failed say the operation failed
         // and output the exception.

         System.out.println("Install/Update failed");
         System.out.println(e);
      }
   }



   // Display help text when parameters are incorrect.

   else
   {
      System.out.println("");
      System.out.println("");
      System.out.println("Parameters are not correct.  Command syntax is:");
      System.out.println("");
      System.out.println("  checkToolbox sourcePath targetPath");
      System.out.println("");
      System.out.println("Where");
      System.out.println("");
      System.out.println("   sourcePath = source for AS/400 Toolbox for Java files");
      System.out.println("   targetPath = target for AS/400 Toolbox for Java files");
      System.out.println("");
      System.out.println("For example:");
      System.out.println("");
      System.out.println("checkToolbox http://mySystem/QIBM/ProdData/HTTP/Public/jt400/ d:\\jt400");
      System.out.println("");
      System.out.println("");
   }

   System.exit(0);
  }
}
```

Figure 108. checkToolbox Program (Page 4 of 4)

Appendix D. Special Notices

This publication is intended to help anyone with a need to understand how to use Java to build AS/400 client/server 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 Number 5769-JV1) or the AS/400 Toolbox for Java (Program Number 5763-JC1). See the PUBLICATIONS section of the IBM Programming Announcement for the AS/400 Developer Kit for Java (Program Number 5769-JV1) or the AS/400 Toolbox for Java (Program Number 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 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.

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

AIX®
AIX/6000®
AIXwindows®
Applet.Author
Application System/Entry®
Application System/400®
AS/400®
Bean Machine
C/400®
Client Access

Client Access/400
Coffee Shop
Cryptolope
DATABASE 2 OS/400
DataHub®
DB2®
DProp
Electronic Marketplace®
IBM Global Network
IBMLink
Integrated Language Environment®
Operating System/2®
OS/400®
PowerPC AS
RPG/400®
SAA®
Software Mall®
SOM (NOT FOR USE IN JAPAN)
SOMobjects Application Class Library
SQL/400®
System/38
VisualAge®
WebConnection
WIN-OS/2®
COBOL/400®
Common User Access®
CUA®
DataGuide®
DataPropagator
Distributed Relational Database Architecture
DRDA®
Encryptolope
IBM®
IIN
IPDS
Operating System/400®
PowerPC Architecture
Print Services Facility
RS/6000
SNAP/SHOT®
Solutions for a small planet
SOMobjects
SOMobjects Compatible
System/36
ThinkPad®
VRPG CLIENT
WebExplorer®
Xstation Manager®

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

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

Appendix E. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

OS/400 Work Management, SC41-5306-01

Performance Tools/400, SC41-5340-00

Distributed Database Programming, SC41-5702-01

QShell Interpreter Reference (see http://www.as400bks.com/ the AS/400 internet book server. Note that this book is only available on the internet)

AS/400 Java Developer Kit for Java Reference (see http://www.as400bks.com/ the AS/400 internet book server. Note that this book is only available on the internet)

Developing Multi-threaded Applications, SC41-5436-00

JAVA 1.1 Interactive Course, Laura Lemay, 1997, The Waite Group (ISBN 1-57169-083-2).

E.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 217.

Accessing the AS/400 System with Java, SG24-2152

E.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
Application Development Redbooks Collection	SBOF-7290	SK2T-8037
Personal Systems Redbooks Collection	SBOF-7250	SK2T-8042

E.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

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com`.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in United States
- **GOPHER link to the Internet** - type `GOPHER.WTSCPOK.ITSO.IBM.COM`
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

  For a list of product area specialists in the ITSO: type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/redbooks`

- **IBM Direct Publications Catalog on the World Wide Web**

  `http://www.elink.ibmlink.ibm.com/pbl/pbl`

  IBM employees may obtain LIST3820s of redbooks from this page.

- **REDBOOKS category on INEWS**
- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to `announce@webster.ibmlink.ibm.com` with the keyword `subscribe` in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.htm`). 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.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** — send orders to:

	IBMMAIL	**Internet**
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** — send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1)001-408-256-5422 (Outside USA)** — ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to `softwareshop@vnet.ibm.com`
- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to `announce@webster.ibmlink.ibm.com` with the keyword `subscribe` in the body of the note (leave the subject line blank).

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.htm`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook 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
CPW	Commercial Processing Workload
EAB	Enterprise Access Builder
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
ITSO	International Technical Support Organization
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Development Toolkit
JIT	Just in Time Compiler
JVM	Java Virtual Machine
MI	Machine Interface
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OOP	Object Oriented Programming
PROFS	Professional Office System
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
WWW	World Wide Web

Index

Special Characters

Numerics

A

B

C

D

E

G

ITSO Redbook Evaluation

Building AS/400 Applications with Java
SG24-2163-00

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.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@vnet.ibm.com

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!)**

_____

_____

_____

_____

_____

Printed in U.S.A.