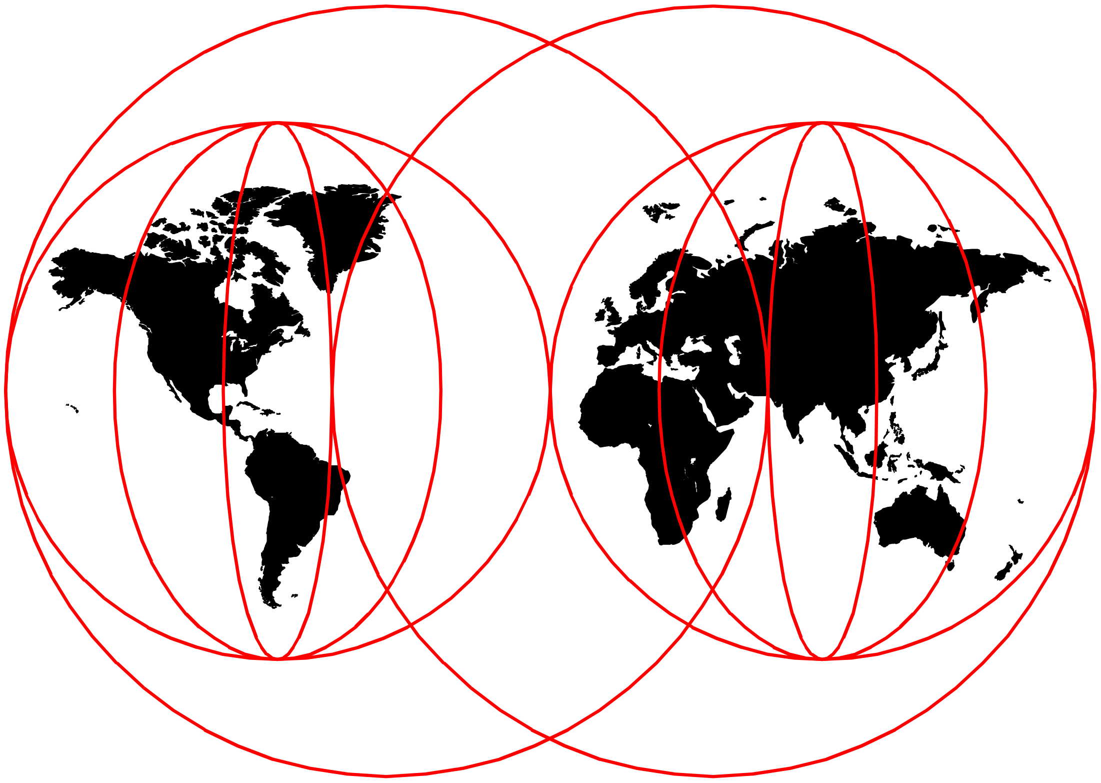
Building AS/400 Client/Server Applications with Java

Bob Maatta, Dan Murphy



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-2152-02

International Technical Support Organization

SG24-2152-02

Building AS/400 Client/Server Applications with Java

July 1999

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 413.

Third Edition (July 1999)

This edition applies to OS/400 Version 3, Release Number 2 and later.

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

Contents

Figures

Tables

Preface

Java's portability and its ability to produce Internet-enabled applications have made it the hot new programming language. If you want to design and build AS/400 client/server applications using Java, this redbook is the source for the information you need. This guide offers you a fast start for using Java and the AS/400 system.

This redbook focuses on two key products: VisualAge for Java Version 2.0 and the AS/400 Toolbox for Java. It provides many practical programming examples with detailed explanations of how they work. These examples are also available for downloading from the redbook Web site.

The Team That Wrote This Redbook

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

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

Dan Murphy works in Hursley, England, at the Solution Partnership Centre (SPC) assisting commercial developers to port to, and make the most of, the AS/400 system. He completed an honor's degree in computer science at Brighton University (England) in 1993, and joined IBM's AS/400 Support Group the same year. During his five year tenure in the support group, Dan has been the team leader for both the AS/400 Communications and Application Development support teams.

The authors of the previous editions of this book were:

Bob Maatta
ITSO Rochester

Markus Abegglen
DV Bern AG

Marshall Dunbar
Data Processing Services, Inc.

Craig Pelkie
Bits&Bytes Programming

Stuart Foster
IBM U.K.

Paul Holm
Cheryl Pflughoeft
Kevin Roberts
Partners In Development, IBM Rochester

Roger Wong
IBM Hong Kong

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

Clif Nock
Doug Petty
Schuman Shao
David Wall
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 427 to the fax number shown on the form.
- Use the online evaluation form found at: `http://www.redbooks.ibm.com/`
- Send your comments in an internet note to: `redbook@us.ibm.com`

Chapter 1. Object-Oriented Technology Overview

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

This chapter contains information about the following subjects:

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

1.1 Before Object-Oriented Technology

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

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

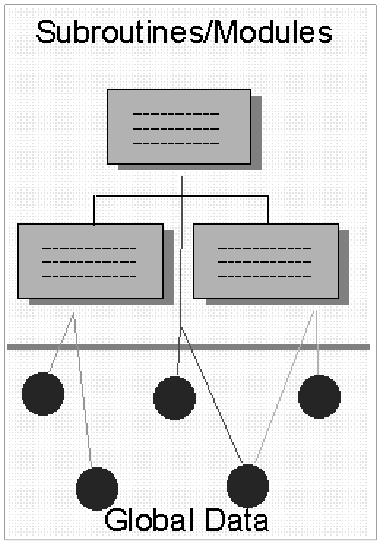


Figure 1. Traditional Application Development Scenario

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

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

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

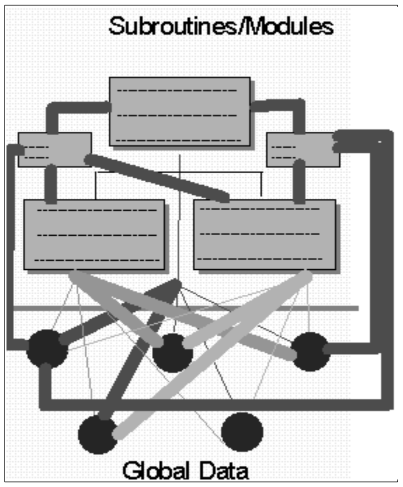


Figure 2. Updated Application Development Scenario

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

1.2 Objects

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

1.2.1 Encapsulation of Objects

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

Objects that share the same behavior are said to belong to the same class. A class is a generic specification for an arbitrary number of similar objects. Objects that behave in a manner specified by a class are called *instances* of that class. All objects are instances of some class. Once an instance of a class is created, it behaves the same as all other instances of its class. Upon receiving a message, it can perform any operation for which it has methods. It may also call on other instances, of the same or other classes, to perform still other operations on its behalf. A program can contain as many or as few instances of a particular class as required. See Figure 3.

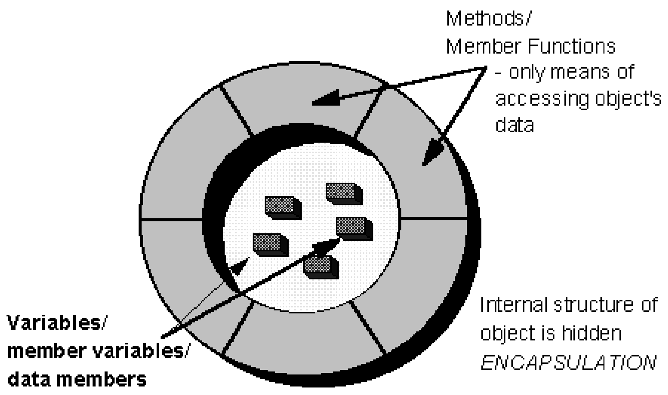

Methods/
Member Functions
- only means of accessing object's data
Variables/
member variables/
data members
Internal structure of object is hidden
ENCAPSULATION

Figure 3. Classes

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

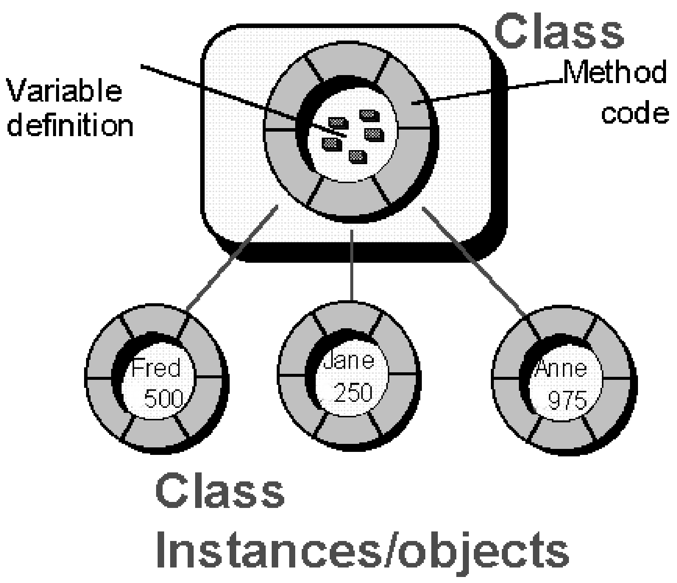


Figure 4. Instantiating Objects

To conclude, note these points:

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

1.3 Class Relationships

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

- Specialization
- Composition
- Collaboration

1.3.1 Specialization

By declaring one class to be a special case, or a subclass of another, the subclass inherits all the method and variable definitions of its superclass. In the class hierarchy shown in Figure 5 on page 5, *vehicle* is the superclass of all the other subclasses, and *car* is a subclass of vehicle (because it is a type of vehicle). *Car* is the superclass of its four subclasses (Hatchback, Station Wagon, Sedan, and Sports). These last four classes are subclasses of *car* because they are a type of car.

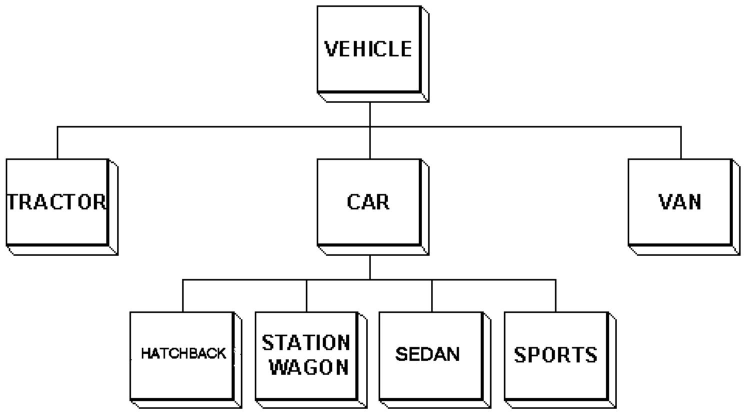


Figure 5. Class Hierarchy

It is all well and good arranging classes into a hierarchy, but what features does the hierarchy have and what benefits does this bring? The vehicle class abstracts as much data and procedures common to all vehicle types, and implements these data items (variables) and functions. In Figure 6, the vehicle class defines the regno variable (registration number or plate number) and all the functions that act on regno, for example, to set its value and to retrieve it (commonly called *setters* and *getters*). These are defined only once at the vehicle class level, but they are immediately inherited by the seven subclasses shown in this hierarchy. There is no copying and pasting code, no retyping. It all happens automatically. This has a dramatic effect on the amount of code needed to be written, on the quality of the code, and on the downstream maintenance effort (because you amend it in one place only, not in eight).

Therefore, in Figure 6, the sedan class has a variable and methods for trunk capacity (which it defines itself), for trim (which it inherits from car), and for regno (which it inherits from vehicle).

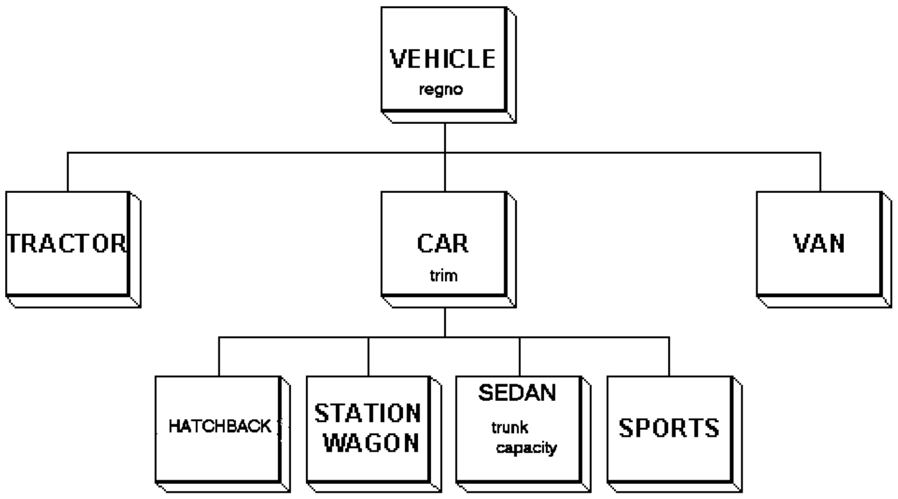


Figure 6. Inheritance

1.3.2 Composition

Classes can also be defined as components of one another. A laser printer may contain, among many other parts, a print engine, a roller, a cartridge, a paper tray, and so on. Composition provides a convenient means of capturing the fact that these parts all go together, and it allows them to be treated as a single collective entity. Composition is especially useful for defining high-level objects that hide the details of their inner workings. A division may consist of a specified set of departments, several divisions can be combined into a business unit, and a company may include any number of business units. It is important not to confuse specialization with composition. They have different properties and serve different functions. For example, the hierarchy defined by an organization is not an inheritance hierarchy. Departments do not inherit properties from divisions, and divisions do not inherit from business units. That is because they are components of one another, not special cases of each other.

1.3.3 Collaboration

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

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

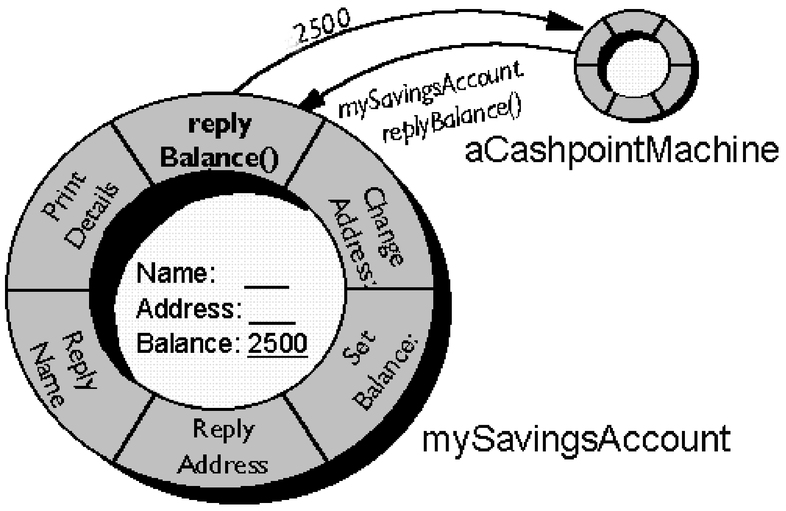


Figure 7. Collaboration

1.4 Polymorphism

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

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

1.5 Benefits of Object-Oriented Technology

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

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

Components (Figure 8 on page 8) are entities that can be used in a number of different programs. Items such as lists, arrays, and strings are components of many different programs. The primary goal when designing components is to make them general, so they can be components of as many different applications as possible. Application developers that use components do not need to understand the implementation of those components. They are reusable code in its simplest form. Components are typically discovered when programmers find themselves repeatedly writing similar pieces of code. Although each piece has been written to accomplish a specific task, the tasks themselves have enough in common that code written to accomplish them appears remarkably alike. When a programmer takes the time to abstract the common elements from the disparate pieces into one, and create a uniform, generally useful interface to it, a component is born. Ultimately, programmers can aim for abstracting out common functionality as they design a piece of software before they code similar pieces again and again.

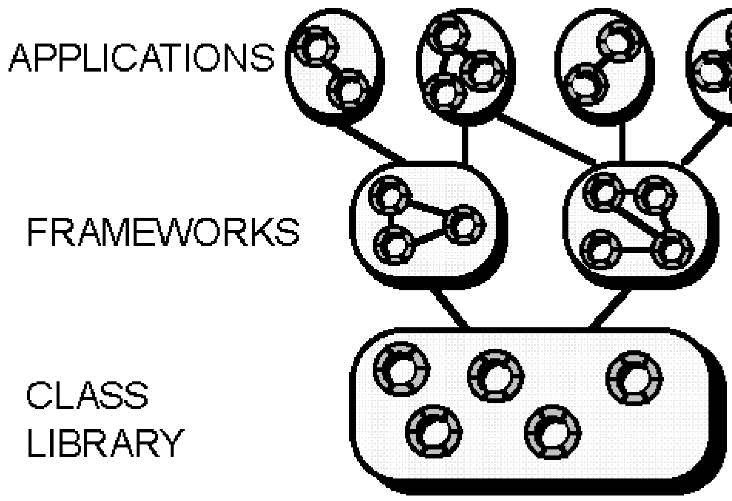


Figure 8. Object-Oriented Development Components

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

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

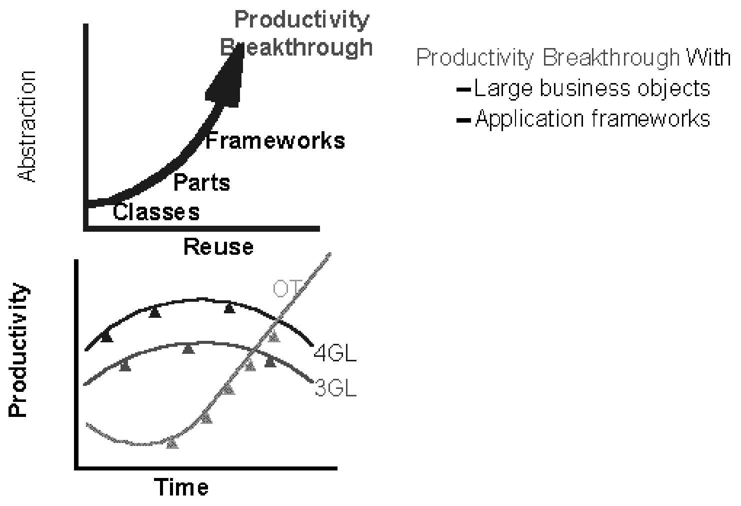


Figure 9. Object-Oriented Technology Benefits

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

In summary, object-oriented technology simulates the real world. Objects are software packages containing methods/functions (behavior) and variables (state). Object-oriented technology delivers the following benefits:

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

These benefits are accomplished by implementing the following concepts:

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

Chapter 2. Introduction to VisualAge for Java

This chapter explains VisualAge for Java with a specific emphasis on the components most likely to be used by an AS/400 development team. It discusses the following topics:

- The VisualAge family
- VisualAge Java overview
- Integrated Development Environment (IDE)
  - Java support
  - Navigating within VisualAge for Java
  - Visual Composition Editor (VCE)
  - Team development
  - Applet viewer
  - Editor/Debugger/SmartGuides
- The Enterprise Access Builders (EAB)
- System requirements and prerequisites
- Migration from VisualAge for Java Version 1.0 to Version 2.0
- Upgrading VisualAge for Java 2.0

This chapter also discusses various processes and windows that you use in the development of applications using VisualAge for Java. All development for this redbook was performed using the Windows NT client of the Enterprise Edition of VisualAge for Java Version 2.0. If you are using a different client or the Professional Edition, there may be some slight differences in the processes and windows discussed and shown here.

The source code for any of the examples discussed in this chapter is available on the Internet. For download instructions, please refer to Section A.1, “Downloading the Files from the Internet” on page 396.

2.1 The VisualAge Family

VisualAge for Java is one of the newest members of the family of VisualAge products. These products cover the complete range of client/server application development topologies, clients, servers, and languages.

The VisualAge family supports the following programming environments:

- VisualAge for Java
- VisualAge for Java e-Business Edition
- VisualAge Generator (4GL)
- VisualAge for COBOL
- VisualAge for RPG
- VisualAge for C++
- VisualAge for Smalltalk
- VisualAge for Basic
- VisualAge for PacBase
- VisualAge Financial Foundation
- VisualAge 2000
- VisualAge WebRunner
- VisualAge Data Atlas
- VisualAge ISPF
- VisualAge Team Connection

- VisualAge Tivoli
- VisualAge PL/I

In addition, the VisualAge product set supports application development across the following client and server platforms.

Note: Not all VisualAge products support all the client and servers listed here.

- OS/2
- Windows 3.1 and 3.11
- Windows NT
- Windows 95/98
- AIX
- OS/390
- OS/400

VisualAge uses a construction-from-parts paradigm, which eases the migration to client/server, object-oriented, and Web-based technologies. With the Visual Composition Editor, which is available with VisualAge for Java, you can develop programs by visually arranging and connecting prefabricated parts. You can also create your own reusable parts.

For a complete description of each of the VisualAge family members and supported environments, visit the VisualAge Family Web page at: http://www.software.ibm.com/ad/

2.2 VisualAge for Java Overview

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

2.2.1 VisualAge for Java Versions

VisualAge for Java is available in four scalable packages:

- **Entry**

  A free, scaled-down version of the Professional product, aimed at people who want to see to believe. Try it, but it is limited to 500 classes. No documentation is shipped with the product (documentation can be viewed through the Support page on the Web). The user is licensed for "internal, non-commercial use." Note these points:

  – No support provided
  – Free
  – Five hundred class limit
  – Does not support the AS/400 Toolbox for Java

- **Professional Edition**

  Includes a robust editor, debugger, browser, and a powerful Visual Composition Editor that uses IBM's award-winning VisualAge programming paradigm. The Professional Edition is included with the VisualAge Developer Domain Subscription for Java. Note these advantages:

- – Supports JDK 1.1.7 and Swing1.0.3
- – Supports the AS/400 Toolbox for Java
- – Includes the IDE and VCE features
- – Includes JavaBeans for Easy Access to Data
- – Includes Integration with VisualAge TeamConnection, ClearCase, and PVCS
- – Includes Open Tool Integrator APIs

- **Professional Edition shipped with ADTS V4R4**

  The AS/400 V4R4 Application Development Toolset (ADTS) product includes a customized version of VisualAge for Java 2.0. The customized version contains all of the features of the normal Professional edition. It includes a copy of the AS/400 Toolbox for Java Modification 1. In addition, it contains two DH Andrews reports on Java, and a coupon for a one-year free membership to the VisualAge Developer Domain.

- **Enterprise Edition**

  The first enterprise-aware, incremental Java application development environment designed to connect Java clients to existing server data, transactions, and applications. It includes these features:

  - – Includes all Professional Edition support and features mentioned above
  - – Includes Java Team Programming Support
  - – Includes Enterprise Toolkits, High Performance Compiler, and a Remote Debugger
  - – Includes Enterprise Access Builders
  - – Includes Servlet Builder
  - – Includes Automated Object to Relational Mapping
  - – Supports SanFrancisco, Tivoli, Lotus, and Component Broker
  - – Supports AIX Development Environment

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

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

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

With VisualAge for Java, the developer can develop 100% compliant Java JDK 1.1 applications and applets all from the same development environment. This enables customers and business partners to migrate to Java-based Web applets at their own pace along an incremental path, including:

- Implementing Java extensions to their applications
- Developing whole Java applications
- Moving to client/server Java applications
- Developing Web-based Java applets

2.2.2 Integrated Development Environment

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

The IDE enables a developer to build and run applications, applets, and code snippets interactively without needing to run the compile statement (JAVAC) from the command line. All applications can be run from within the IDE without exporting the Java source or class files. This is achieved through the provision of a JDK 1.1 compliant Virtual Machine (VM) within the IDE. Because you can interactively modify code and run it without compilation, developers can debug code on the fly. They can spot errors in their code with the debugger, change it, and continue without bringing the running application down—all within the VisualAge for Java IDE.

VisualAge for Java is an open IDE. Developers can easily import and export Java source and class files, as well as JavaBeans that may have been purchased by the company or made available on the Internet. The JavaBeans support in VisualAge for Java also enables a developer to import an existing JavaBean (for example, from the Internet) into VisualAge for Java, modify the bean, and export it again for use within another JDK 1.1 compliant development environment (for example, Symantic Cafe and Borland JBuilder).

Version 1 of VisualAge for Java supports JDK 1.1.4. Version 2 supports JDK 1.1.7 (the most recent released version at the time of publication). Along with the current JDK support, VisualAge for Java also supports all the most current standards for Java development (for example, Java Database Connectivity (JDBC) and so forth), which is discussed later. Because of the portability of JDK 1.1 compliant Java code, code that is developed using VisualAge for Java can run without change on the native AS/400 Java Virtual Machine, which is now available.

2.2.3 Components and Features

VisualAge for Java has two components that extend its capabilities to make client/server programming easier. The Enterprise Access Builders (EAB) provide components to aid connection to DB2 compliant data sources, Customer Information Control System (CICS) transactions, and other programs. In addition, the AS/400 Toolbox for Java provides a series of classes specifically designed to access many AS/400 features (all without using Client Access/400 as a prerequisite).

The initial release of the product runs on OS/2 Warp Version 4.0, Windows NT 4.0, or Windows 95/98.

VisualAge for Java Enterprise Edition Version 2.0 offers the following features:

- Support for JDK 1.1.7 and Swing 1.0.3
- An Integrated Development Environment (IDE) with visual programming support for creating Java applets and Swing beans

- Visual Composition Editor (VCE)
- Support for a team of programmers to share and maintain source code in a single repository
- Wizards for string externalization to assist in building multi-lingual applications
- Complete support for object serialization
- Ability to import GUIs built in other Java development environments
- Support for JavaDoc output
- JavaBeans for easy access to data
- Integration with VisualAge TeamConnection, ClearCase, and PVCS
- Enterprise Toolkits for AS/400 and Workstation, including High Performance Compilers for Java and a Remote Java Debugger (the Enterprise Toolkit for OS/390 is not included in this product)

  Note: IBM intends to deliver VisualAge for Java, Enterprise Edition for OS/390. It will include a high-performance compiler for compiling class files compatible with Java Development Kit (JDK) 1.1.7, remote debug capabilities (Windows NT client only), and a performance analyzer (Windows NT client only).
- Enterprise Access Builder for SAP R/3 using SAP R/3 BAPI business objects
- Enterprise Access Builder for Data for JDBC access to enterprise data
- Enterprise Access Builder for Java to C++ for access to C++ programs
- Enterprise Access Builder for RMI for creating distributed Java applications
- Enterprise Access Builder for Persistence for transforming relational schemas into Enterprise JavaBeans components
- Enterprise Access Builder for interacting with existing applications and data
- IDL Development Environment for building CORBA-compliant applications
- Servlet Builder for creating and testing servlets
- Tool Integrators API for extending VisualAge for Java
- Wizards for building applications from San Francisco Application Components
- Domino AgentRunner tool for running and debugging your Lotus Domino agents in the IDE
- Tivoli beans to make your Java applications "ready to manage" with Tivoli's enterprise management software
- MigrationAssistant to help you migrate ActiveX controls to JavaBeans
- HTML documentation with advanced search capabilities

All of the preceding components use the JDK 1.1 and Java Virtual Machine Support of VisualAge for Java. Note that the Professional Edition does not include all of these mentioned features.

2.3 Integrated Development Environment (IDE)

This section of the chapter covers the Integrated Development Environment (IDE) component of VisualAge for Java.

2.3.1 Java Support

Java is a collection of classes built from the ground up, following object-oriented (OO) principles. In Java, everything is an object except for the standard data types inherited at the top of the hierarchy from the root class, object.

Java classes are contained in packages. The concept of a package in Java is a useful way of grouping classes that are related. A Java package is similar in concept to an AS/400 ILE service program.

JDBC is the Java standard to manipulate enterprise data stored in relational databases. It is the Java equivalent to ODBC, a widely accepted standard developed by Microsoft. JDBC provides a standard SQL database access interface. Constructs such as database connections, SQL statements, result sets, and database meta data are included. With JDBC, it is possible to develop Java applications independently of the target relational database management system (R-DBMS). Many vendors already provide (or will provide in the near future) JDBC drivers targeted at accessing dozens of database management systems. The AS/400 system is no exception and IBM Rochester provides a JDBC driver to access DB2/400 Database as part of the AS/400 Toolbox for Java set of classes.

In conjunction with JDBC, JavaSoft has released a JDBC-to-ODBC bridge. Such a bridge provides a way for Java applications developed to the JDBC standard to gain access to any database using the existing ODBC drivers.

Remote Method Invocation (RMI) lets programmers create Java objects whose methods can be invoked from another Java Virtual Machine. RMI is equivalent to a Remote Procedure Call in the non-object world.

The JavaBeans API defines a portable, platform-neutral set of APIs for software components. JavaBeans components can plug into existing component architectures such as IBM's OpenDoc, Microsoft's OLE/COM/Active-X architecture, or Netscape's LiveConnect.

Java Native Interface was known previously as the native method interface in JDK 1.0. It provides the capability for a Java object to call a native platform function typically written in C, C++, or any other language.

The internationalization support allows the development of localized applets and applications. The global Internet demands global software that can be developed independently of the countries or languages of its users, and be localized for multiple countries or regions. JDK 1.1 provides a rich set of Internationalization APIs for developing global applications. These APIs are based on the Unicode 2.0 character encoding and include the ability to adapt text, numbers, dates, currency, and user-defined objects to any country's conventions.

Java Archive (JAR) is a platform-independent file format that aggregates many files into one, similar in concept to a ZIP file. Multiple Java applets and their requisite components (class files, images, and sounds) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction, which greatly improves the download speed. The JAR format also supports compression, which reduces the file size and further improves the download time. In addition, the Applet author can digitally sign individual entries in a JAR file to authenticate their origin. It is fully backward-compatible with existing applet code and is fully extendible, being written in Java.

The Core Java JDK 1.1 API includes the following packages:

- **Java.lang**

  This package contains all the classes and interfaces of the base Java language.

- **Java.util**

  This package contains various utility classes and interfaces, including random numbers, system properties, and other useful classes.

- **Java.io**

  This package provides the input/output classes and the interfaces for files and streams.

- **Java.net**

  This package is composed of classes and interfaces for handling network operations such as TCP/IP, Sockets, and URL.

- **Java.awt**

  This package allows for the definition of GUI constructs that are portable to multiple windowing systems. This is the only package in the core API to include sub-packages. The following sub-packages are part of the Java.awt package:

  – *Java.awt.image*

    Provides the classes necessary to handle images in various formats, such as GIF and JPEG.

  – *Java.awt.peer*

    Provides hidden classes that map their Java.awt equivalents and are designed to implement the GUI constructs on specific platforms such as Apple's Macintosh, Microsoft's Windows 95/98/NT, or UNIX's Motif.

- **Java.applet**

  This package is designed to provide the behavior specifically for applets.

For a full description of the Java class library and core API, visit the JavaSoft JDK Web site at: http://www.java.sun.com:80/products/jdk/1.1/docs/index.html

2.3.2 Navigating within VisualAge for Java

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

- Starting VisualAge for Java
- The Workbench and its hierarchy:

  – Projects
  – Packages
  – Classes

  - Interfaces
  - Managing
  - All Problems

- Browsers:
  - Project
  - Package
  - Class

2.3.2.1 Starting VisualAge for Java

During the installation of VisualAge for Java, an item is added to the Windows taskbar — *IBM VisualAge for Java for Windows*. This item has a number of sub-items, and selecting IBM VisualAge for Java starts VisualAge for Java. Follow a similar process if you are using the OS/2 or Windows NT client (see Figure 10).

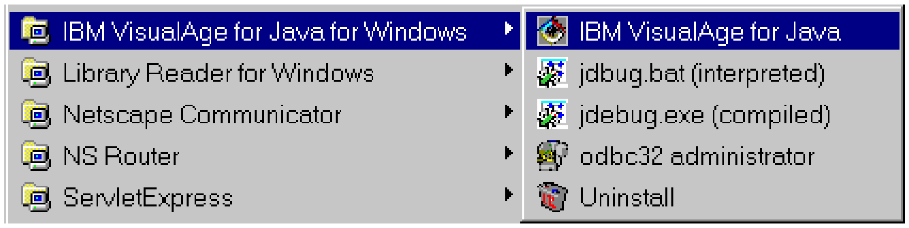


Figure 10. Starting VisualAge for Java

During the start-up process, VisualAge for Java loads the development image. Since this image can be 8MB or larger (typically in the 15MB to 25MB range), the start-up process can take one to two minutes because the entire image must be loaded into memory. The development image is also known as the *workspace* and these two terms are used interchangeably in this chapter. If this is the first time VisualAge for Java is started, the first window displayed is the Welcome dialog window (see Figure 11 on page 19).

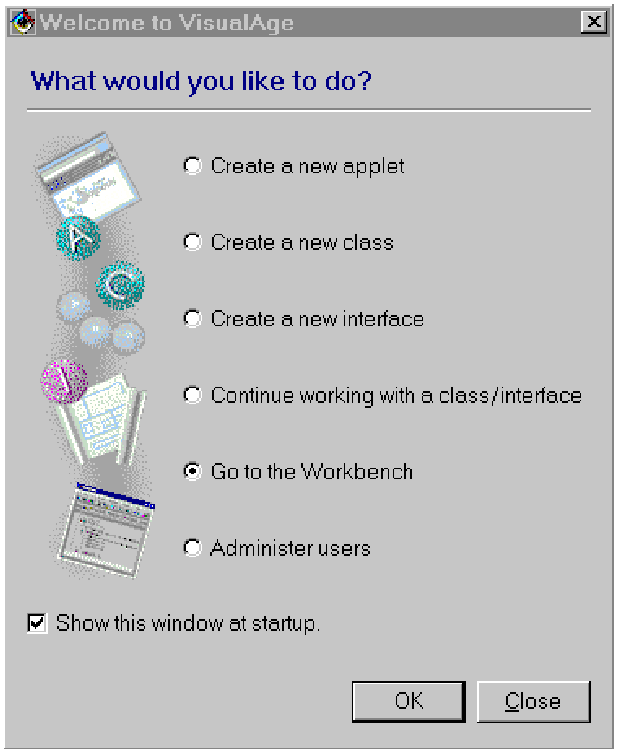
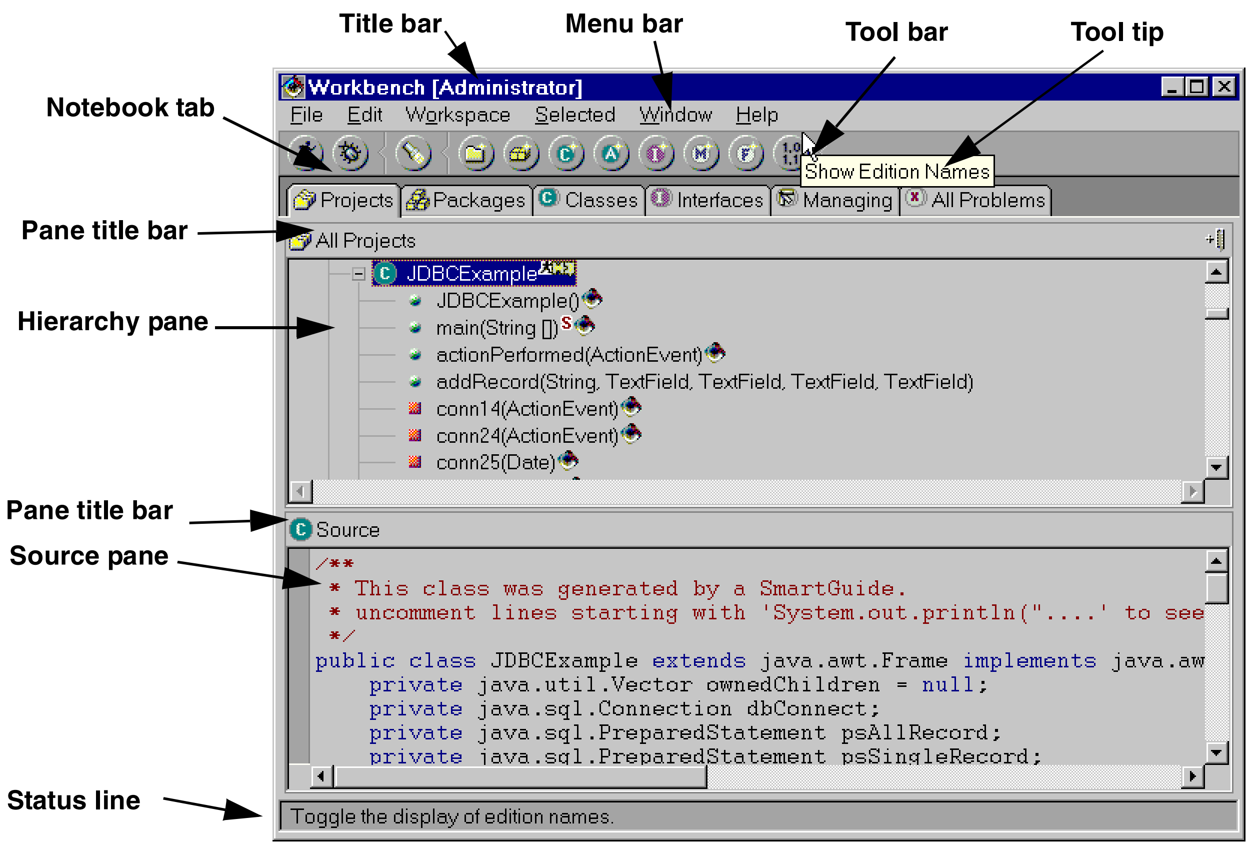


Figure 11. Welcome Dialog Window

The Welcome dialog window provides a single point to perform most of the simple tasks. However, as you gain more experience using VisualAge for Java, you may decide to stop this window from appearing at start-up. To do so, click on the corresponding check box. If you wish to reactivate the Welcome dialog window later, refer to Figure 73 on page 77.

Select **Go to the Workbench** and press **OK** to go to the Workbench window. The window shown in Figure 12 on page 20 appears.



Title bar
Menu bar
Tool bar
Tool tip
Notebook tab
Pane title bar
Hierarchy pane
Pane title bar
Source pane
Status line
Workbench [Administrator]
File Edit Workspace Selected Window Help
Show Edition Names
Projects Packages Classes Interfaces Managing All Problems
All Projects
JDBCExample
JDBCExample()
main(String [])
actionPerformed(ActionEvent)
addRecord(String, TextField, TextField, TextField, TextField)
conn14(ActionEvent)
conn24(ActionEvent)
conn25(Date)
Source
Toggle the display of edition names.

```
/**
 * This class was generated by a SmartGuide.
 * uncomment lines starting with 'System.out.println("....' to see
 */
public class JDBCExample extends java.awt.Frame implements java.aw
    private java.util.Vector ownedChildren = null;
    private java.sql.Connection dbConnect;
    private java.sql.PreparedStatement psAllRecord;
    private java.sql.PreparedStatement psSingleRecord;
```

Figure 12. VisualAge for Java Workbench

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

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

- **Title bar**

  Contains the title and current user.

- **Menu bar**

  Provides access to all functions.

- **Tool bar**

  Provides fast access to most used menu items.

- **Tool tip**

  Shows the meaning of the toolbar buttons.

- **Notebook tab**

  Provides a view of the four fundamental components of VisualAge for Java (projects, packages, classes, and interfaces), as well as a tab for managing them and one for displaying any unresolved problems.

- **Hierarchy pane**

  Typically displays the component being browsed in context with its contained components. For example, a project browser shows all of its packages and each package is expandable to show all of the classes/interfaces it contains, and so forth.

- **Source pane**

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

- **Pane title bar**

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

- **Status line**

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

2.3.2.2 Component Hierarchy

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

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

You are probably already aware of the package, class or interface, and method or constructor components that are part of the standard Java language. In addition, VisualAge for Java includes a higher grouping level called *projects*, which enables the grouping together of various packages. Each higher level component can have multiple lower level components. For example, a project can contain one or more packages.

Various icons are used in each of the browsers to depict each component. Examples of the icons used are shown in Figure 13 through Figure 17. Details about Icons used in VisualAge for Java can be found in the on-line help under **Reference —> Integrated Development Environment —> IDE Symbols** (see Section 2.3.7.6, “VisualAge for Java Help” on page 82).

Figure 13. Project Icon

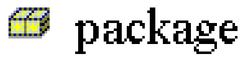


Figure 14. Package Icon

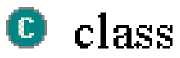


Figure 15. Class Icon

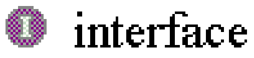


Figure 16. Interface Icon

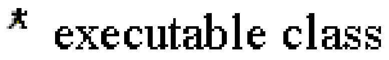
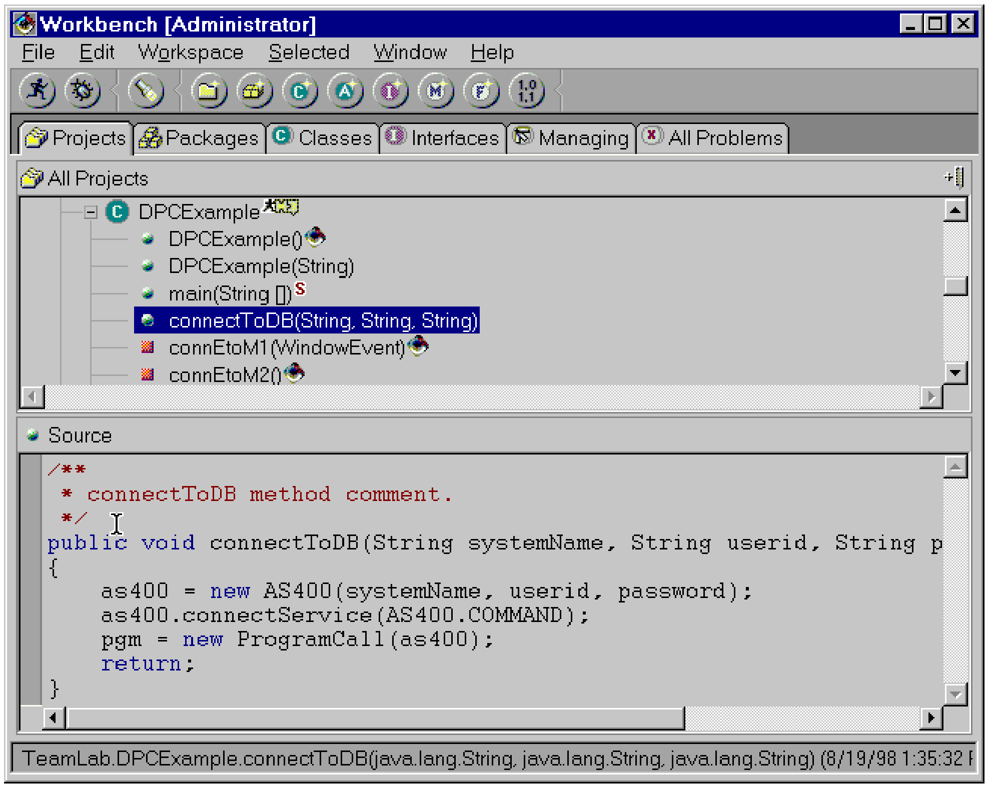


Figure 17. Executable Class Icon

2.3.2.3 Workbench Window

In the Workbench window, the TeamLab project is expanded to show its packages. One of these packages, the TeamLab Package, is expanded to show its classes and interfaces (classes only in this case). One of these classes, the DPCExample Class, is expanded to show its methods. One of its methods, the connectToDB (String, String, String) method, is selected and its source is shown in the source pane (see Figure 18 on page 23).


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
DPCExample
DPCExample()
DPCExample(String)
main(String [])
connectToDB(String, String, String)
connEtoM1(WindowEvent)
connEtoM2()
Source

```
/**
 * connectToDB method comment.
 */
public void connectToDB(String systemName, String userid, String p
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400);
    return;
}
```

TeamLab.DPCExample.connectToDB(java.lang.String, java.lang.String, java.lang.String) (8/19/98 1:35:32 P

Figure 18. VisualAge for Java Workbench

2.3.2.4 Component Browsers

This section discusses the three component browsers used extensively within VisualAge for Java (projects, packages and types, which includes classes and interfaces). You can open each of these browsers by selecting the menu Workspace from the menu bar. There, you select the menu item of the browser you want to open (see Figure 19).

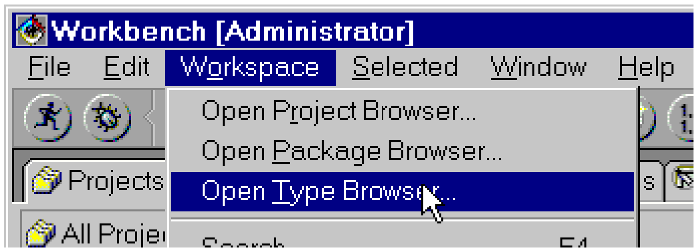
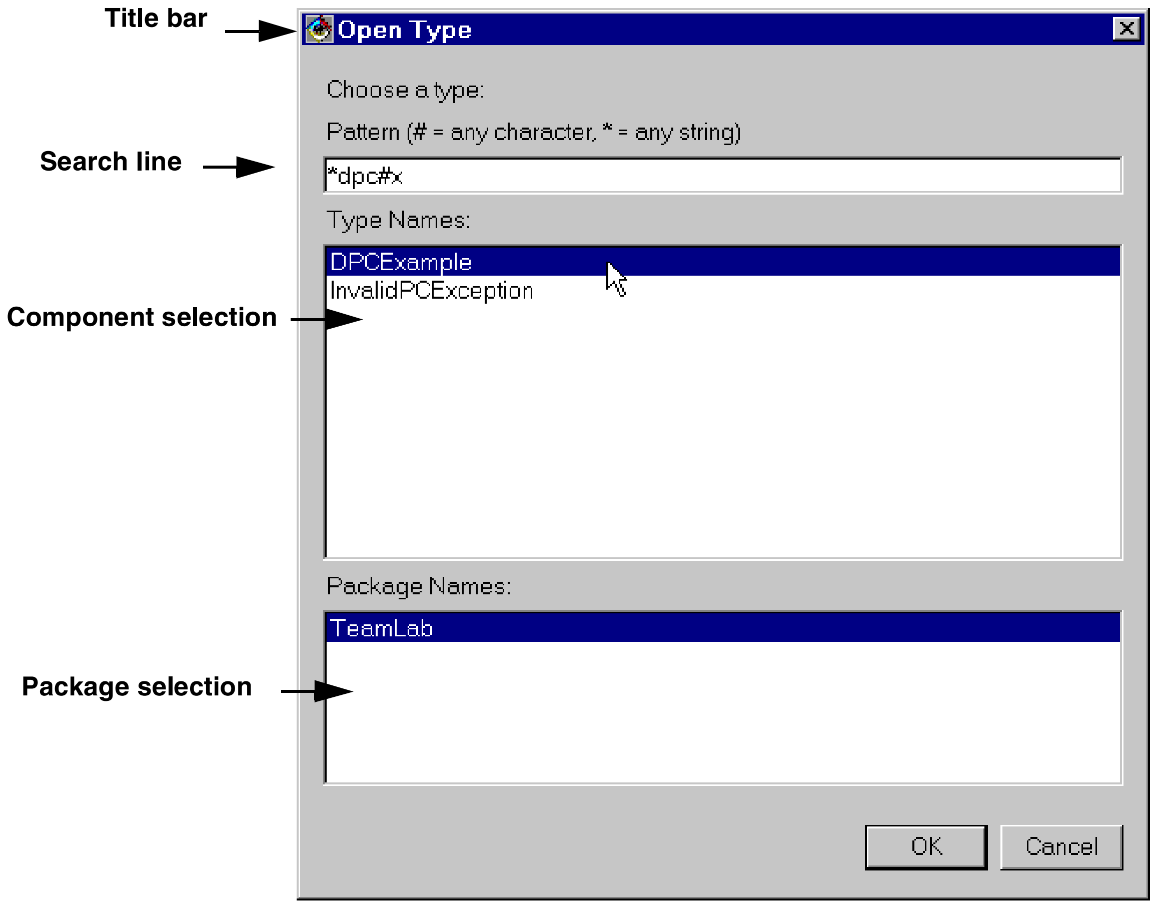


Figure 19. Browser Selection

You are prompted to choose one of the corresponding components, either project, package, or type. The browser appears after you make a selection from that window (see Figure 20 on page 24).



Choose a type:

Pattern (# = any character, * = any string)

*dpc#x

Type Names:

DPCExample
InvalidPCException

Package Names:

TeamLab

Figure 20. Component Selection

Figure 20 shows the information and functions that the Component Selection window provides. They include:

- **Title bar**

  Indicates which browser you are about to open.

- **Search line**

  Provides a search including wildcards.

- **Component selection**

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

- **Package selection**

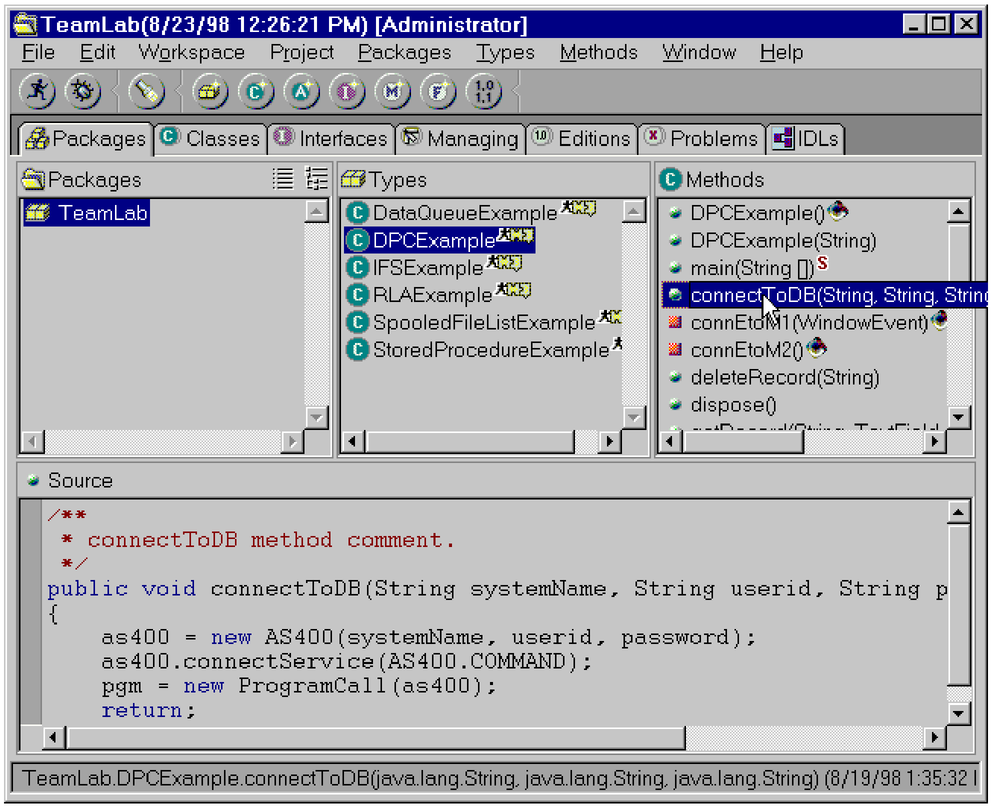
  If a component exists in more than one package, you decide here from which of the packages the component should be taken. This applies only for types, since packages and projects must have unique names in your workspace.

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

There are other ways to open a browser directly. For example, right-click a component shown on the hierarchy pane of the Workbench (see Figure 12 on page 20) and select Open from its pop-up menu. After that, the corresponding browser opens. Since Open is the default response for double-clicking, you receive the same result.

2.3.2.5 Project Browser

The project browser displays details of all the components within the project, including the packages, classes, interfaces, methods, comments, and source code. If you select a project, package, type, or method, you can display or edit the comments or source code directly in the source pane (see Figure 21).



```
/**
 * connectToDB method comment.
 */
public void connectToDB(String systemName, String userid, String p
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400);
    return;
```

TeamLab.DPCExample.connectToDB(java.lang.String, java.lang.String, java.lang.String) (8/19/98 1:35:32

Figure 21. Project Browser Packages View

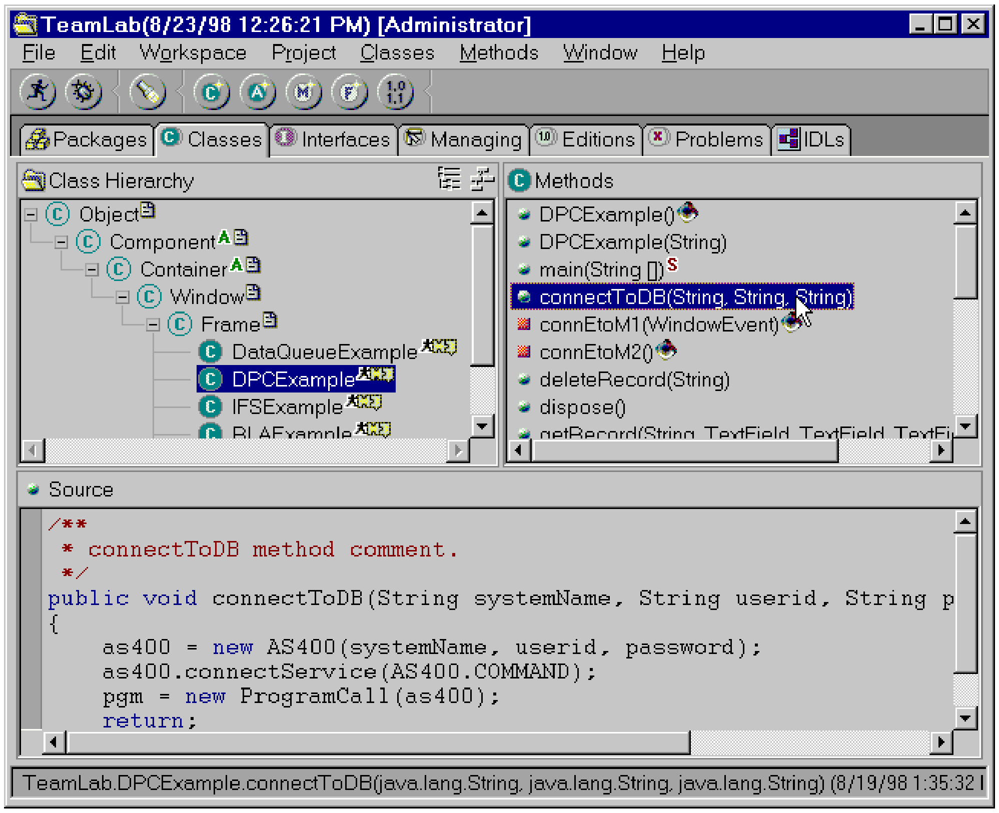
The project browser window has seven different views that you can access by pressing their notebook tabs:

- **Packages view**

  Provides detailed information about all packages, types, and methods within the project. All of this information can be displayed and edited in the source pane (see Figure 21).

- **Classes view**

  Provides detailed information about the class hierarchy, all classes, and their methods within the project (see Figure 22 on page 26).


TeamLab(8/23/98 12:26:21 PM) [Administrator]
File Edit Workspace Project Classes Methods Window Help
Packages Classes Interfaces Managing Editions Problems IDLs
Class Hierarchy
Object
Component
Container
Window
Frame
DataQueueExample
DPCExample
IFSExample
RLAExample
Methods
DPCExample()
DPCExample(String)
main(String [])
connectToDB(String, String, String)
connEtoM1(WindowEvent)
connEtoM2()
deleteRecord(String)
dispose()
getRecord(String, TextField, TextField, TextFi
Source

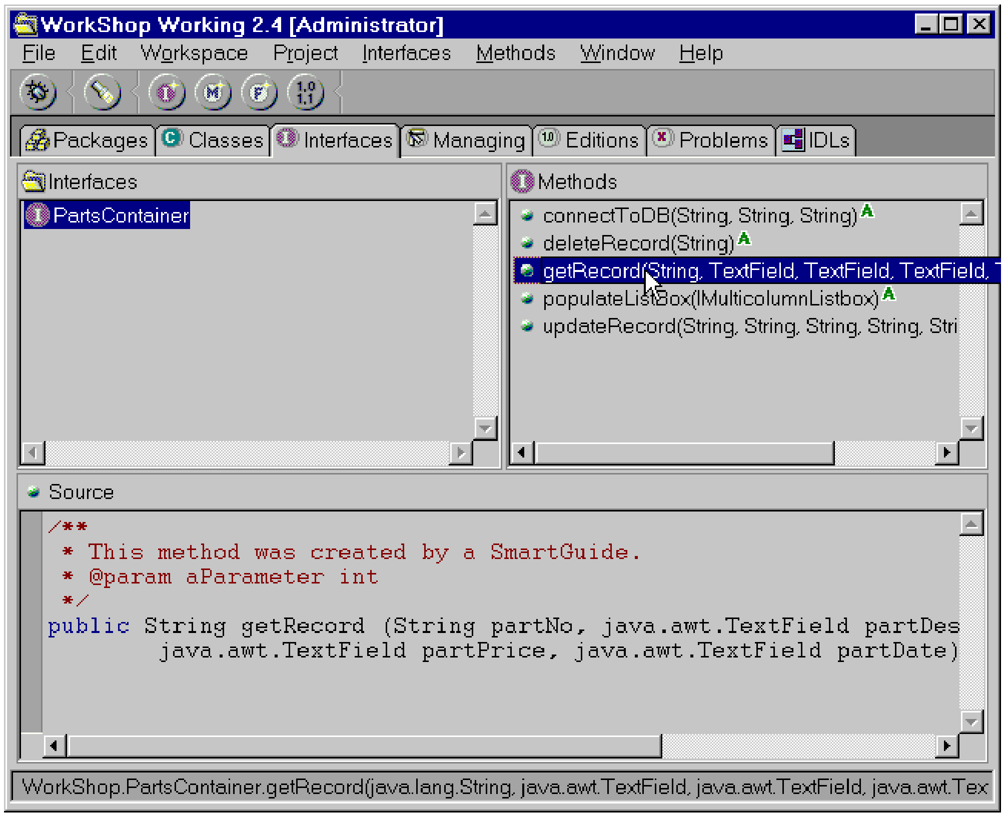
```
/**
 * connectToDB method comment.
 */
public void connectToDB(String systemName, String userid, String p
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400);
    return;
```

TeamLab.DPCExample.connectToDB(java.lang.String, java.lang.String, java.lang.String) (8/19/98 1:35:32

Figure 22. Project Browser Classes View

- **Interfaces view**

  Provides detailed information about all interfaces and their methods within the project (see Figure 23 on page 27).


WorkShop Working 2.4 [Administrator]
File Edit Workspace Project Interfaces Methods Window Help
Packages Classes Interfaces Managing Editions Problems IDLs
Interfaces
PartsContainer
Methods
connectToDB(String, String, String)
deleteRecord(String)
getRecord(String, TextField, TextField, TextField,
populateListBox(IMulticolumnListbox)
updateRecord(String, String, String, String, Stri
Source

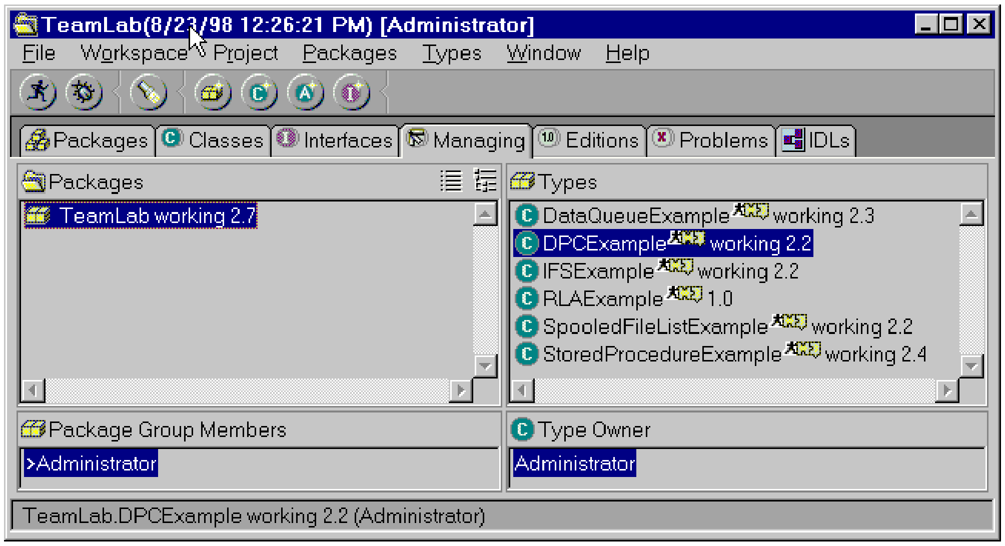
```
/**
 * This method was created by a SmartGuide.
 * @param aParameter int
 */
public String getRecord (String partNo, java.awt.TextField partDes
        java.awt.TextField partPrice, java.awt.TextField partDate)
```

WorkShop.PartsContainer.getRecord(java.lang.String, java.awt.TextField, java.awt.TextField, java.awt.Tex

Figure 23. Project Browser Interfaces View

- **Managing view**

  Is only available in Enterprise Edition. Provides detailed information about package group members (project team members) and ownership of types within the project. All of this information can be managed in the corresponding pane (see Figure 24).

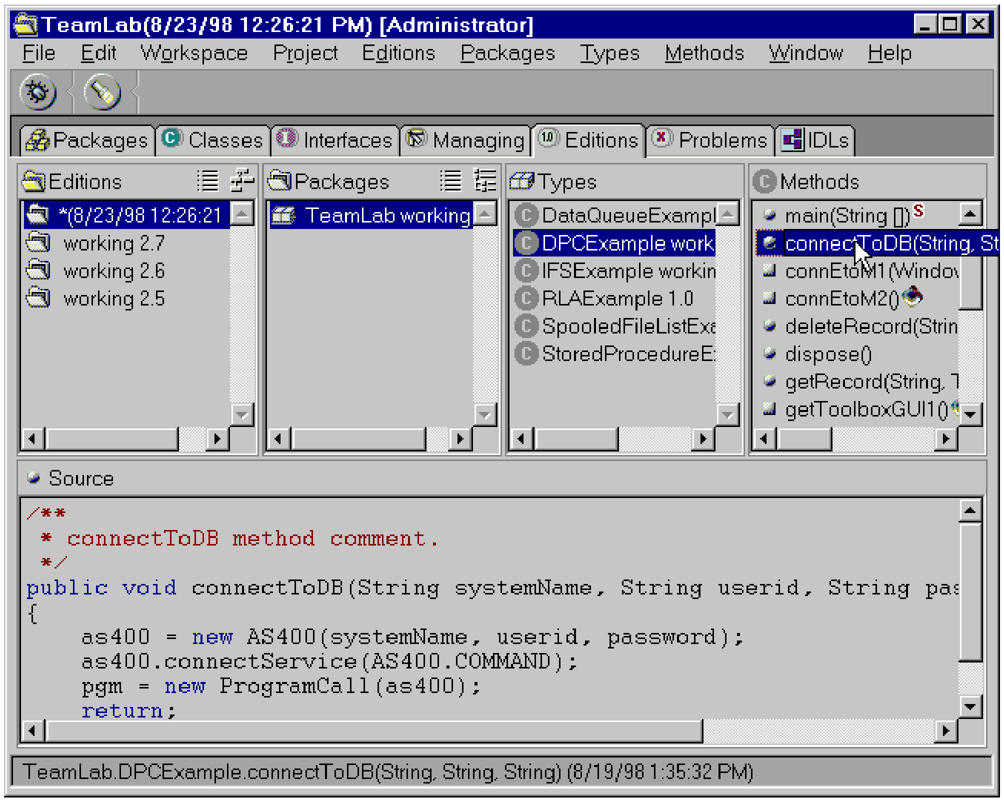

TeamLab(8/23/98 12:26:21 PM) [Administrator]
File Workspace Project Packages Types Window Help
Packages Classes Interfaces Managing Editions Problems IDLs
Packages
TeamLab working 2.7
Types
DataQueueExample working 2.3
DPCExample working 2.2
IFSExample working 2.2
RLAExample 1.0
SpooledFileListExample working 2.2
StoredProcedureExample working 2.4
Package Group Members
>Administrator
Type Owner
Administrator

TeamLab.DPCExample working 2.2 (Administrator)

Figure 24. Project Browser Managing View

- **Editions view**

  Provides detailed information about all versions and editions of packages, types and methods within the project. It enables the developer to manage multiple versions and editions of packages, classes, interfaces, and methods (see Figure 25).



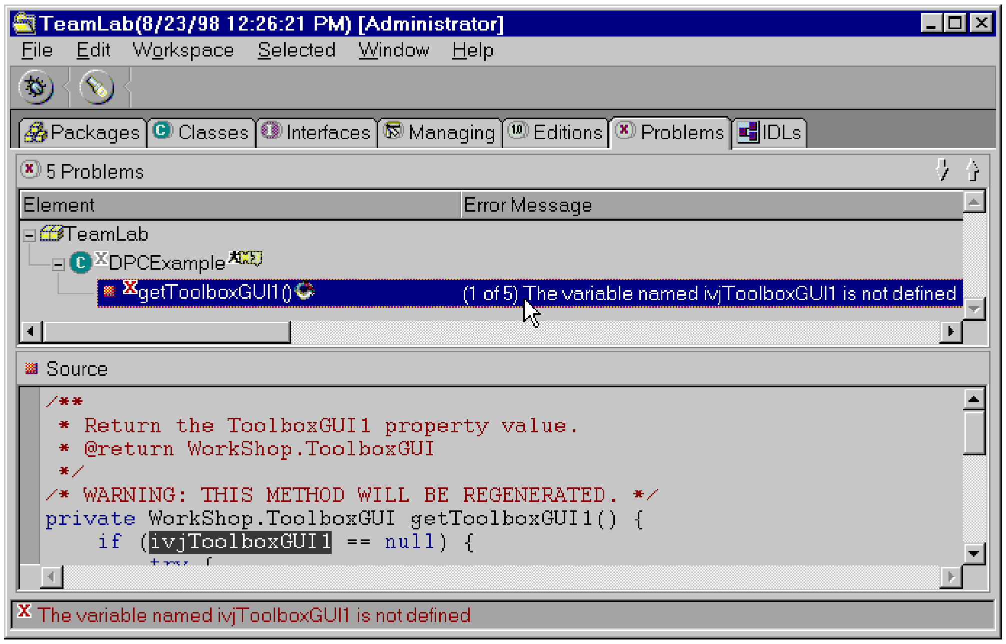
```
/**
 * connectToDB method comment.
 */
public void connectToDB(String systemName, String userid, String pas
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400);
    return;
```

TeamLab.DPCExample.connectToDB(String, String, String) (8/19/98 1:35:32 PM)

Figure 25. Project Browser Editions View

- **Problems view**

  Provides detailed information about all unresolved problems within the project. All of this information can be managed in the corresponding pane (see Figure 26 on page 29).


TeamLab(8/23/98 12:26:21 PM) [Administrator]
File Edit Workspace Selected Window Help
Packages Classes Interfaces Managing Editions Problems IDLs
5 Problems
Element
Error Message
TeamLab
DPCExample
getToolboxGUI1()
Source

(1 of 5) The variable named ivjToolboxGUI1 is not defined

```
/**
 * Return the ToolboxGUI1 property value.
 * @return WorkShop.ToolboxGUI
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private WorkShop.ToolboxGUI getToolboxGUI1() {
    if (ivjToolboxGUI1 == null) {
```

The variable named ivjToolboxGUI1 is not defined

Figure 26. Project Browser Problems View

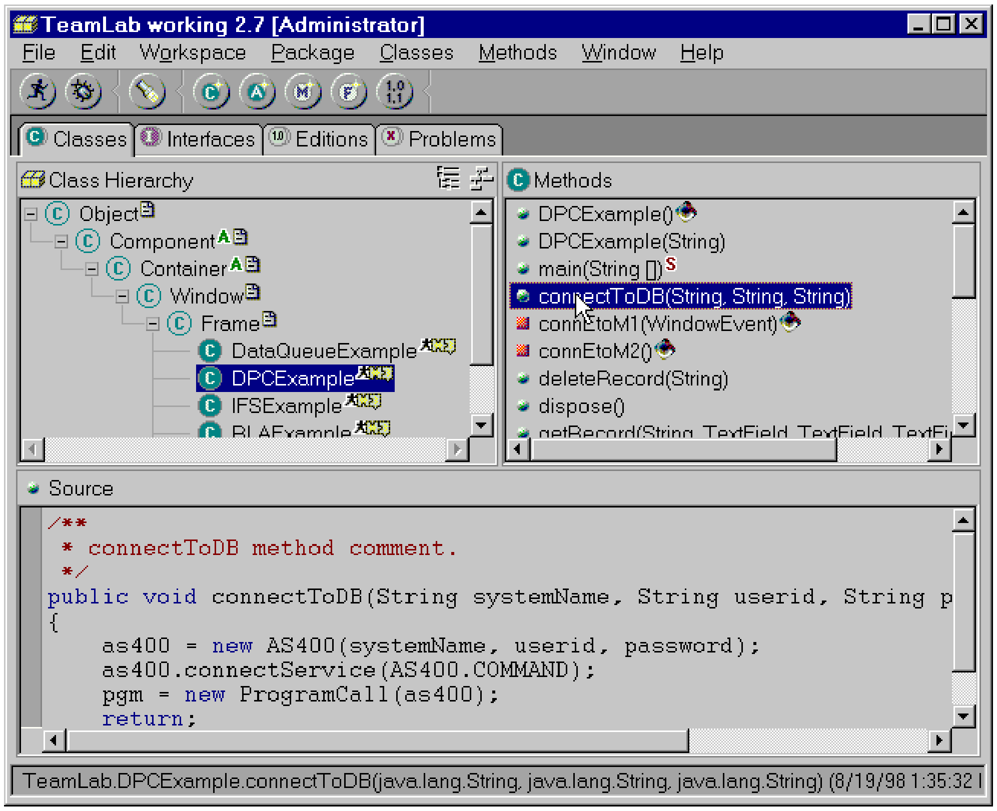
- **IDLs view**

  Is only available in the Enterprise Edition. It provides detailed information about all IDLs (Interface Definition Language) within the project.

2.3.2.6 Package Browser

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

The views in this browser have a slightly different layout, but they work basically the same way as the corresponding views of the project browser. The only difference is that you navigate within a package and not within a project. For this reason, the number of upper panes is reduced to two, and there is more space for class and method names (see Figure 27 on page 30).



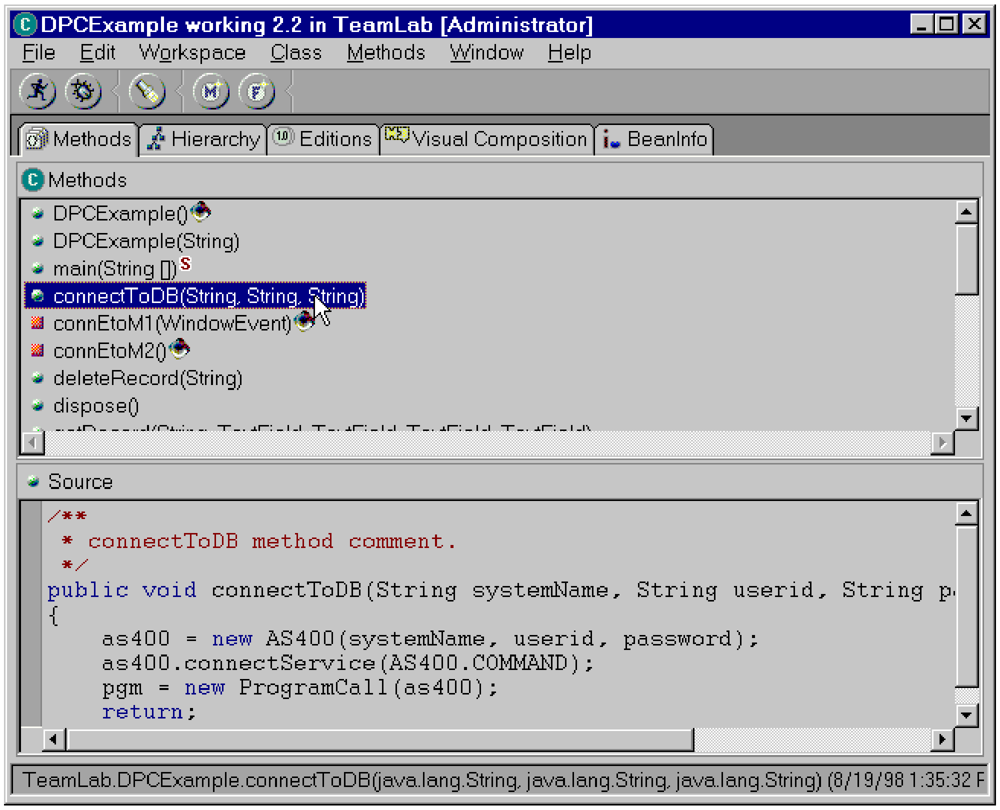
```
/**
 * connectToDB method comment.
 */
public void connectToDB(String systemName, String userid, String p
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400);
    return;
```

TeamLab.DPCExample.connectToDB(java.lang.String, java.lang.String, java.lang.String) (8/19/98 1:35:32

Figure 27. Package Browser Classes View

2.3.2.7 Type Browser

The type browser is a little different in its implementation when compared with the project and package browsers. The type browser is used to display classes and interfaces (types) in the upper pane and their methods, comments, and source code in the lower pane (see Figure 28 on page 31).

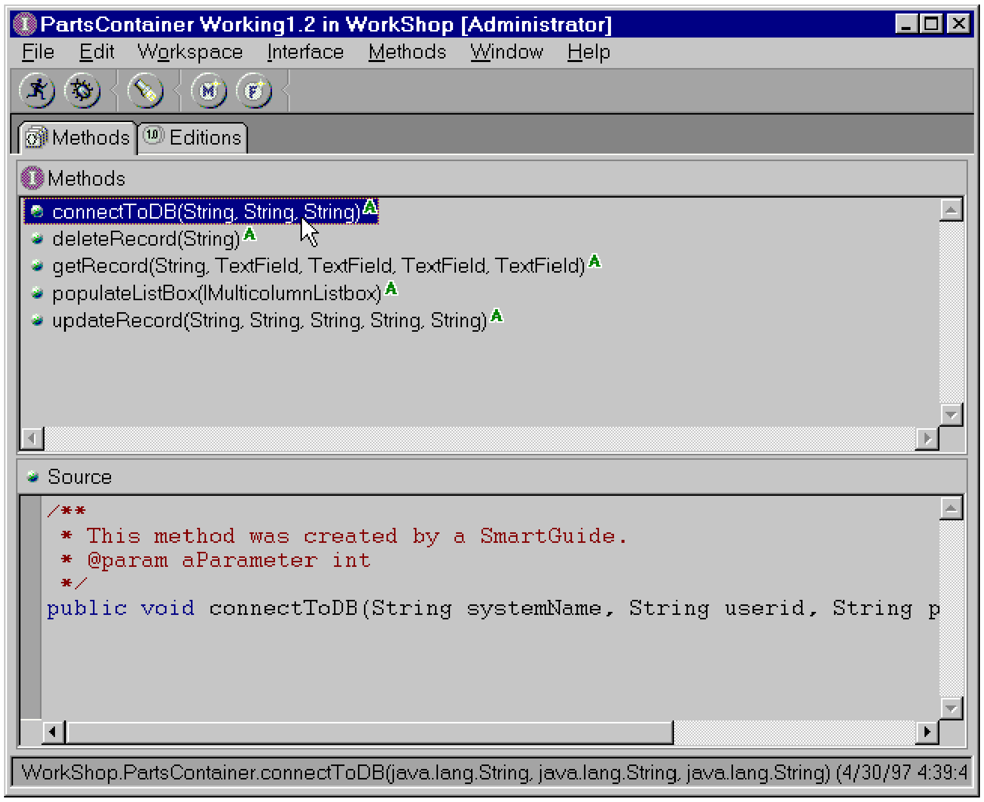

DPCExample working 2.2 in TeamLab [Administrator]
File Edit Workspace Class Methods Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Methods
DPCExample()
DPCExample(String)
main(String [])
connectToDB(String, String, String)
connEtoM1(WindowEvent)
connEtoM2()
deleteRecord(String)
dispose()
Source

```
/**
 * connectToDB method comment.
 */
public void connectToDB(String systemName, String userid, String p
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400);
    return;
```

TeamLab.DPCExample.connectToDB(java.lang.String, java.lang.String, java.lang.String) (8/19/98 1:35:32 F

Figure 28. Type Browser Browsing Class

The hierarchy view and editions view work in the same way as the other two browsers. In the hierarchy pane, you can switch between the tree and graph layout. When displaying interfaces with the type browser, you only have two views, one for methods and one for editions (see Figure 29 on page 32).


PartsContainer Working1.2 in WorkShop [Administrator]
File Edit Workspace Interface Methods Window Help
Methods
Editions
Methods
connectToDB(String, String, String)
deleteRecord(String)
getRecord(String, TextField, TextField, TextField, TextField)
populateListBox(IMulticolumnListbox)
updateRecord(String, String, String, String, String)
Source

```
/**
 * This method was created by a SmartGuide.
 * @param aParameter int
 */
public void connectToDB(String systemName, String userid, String p
```

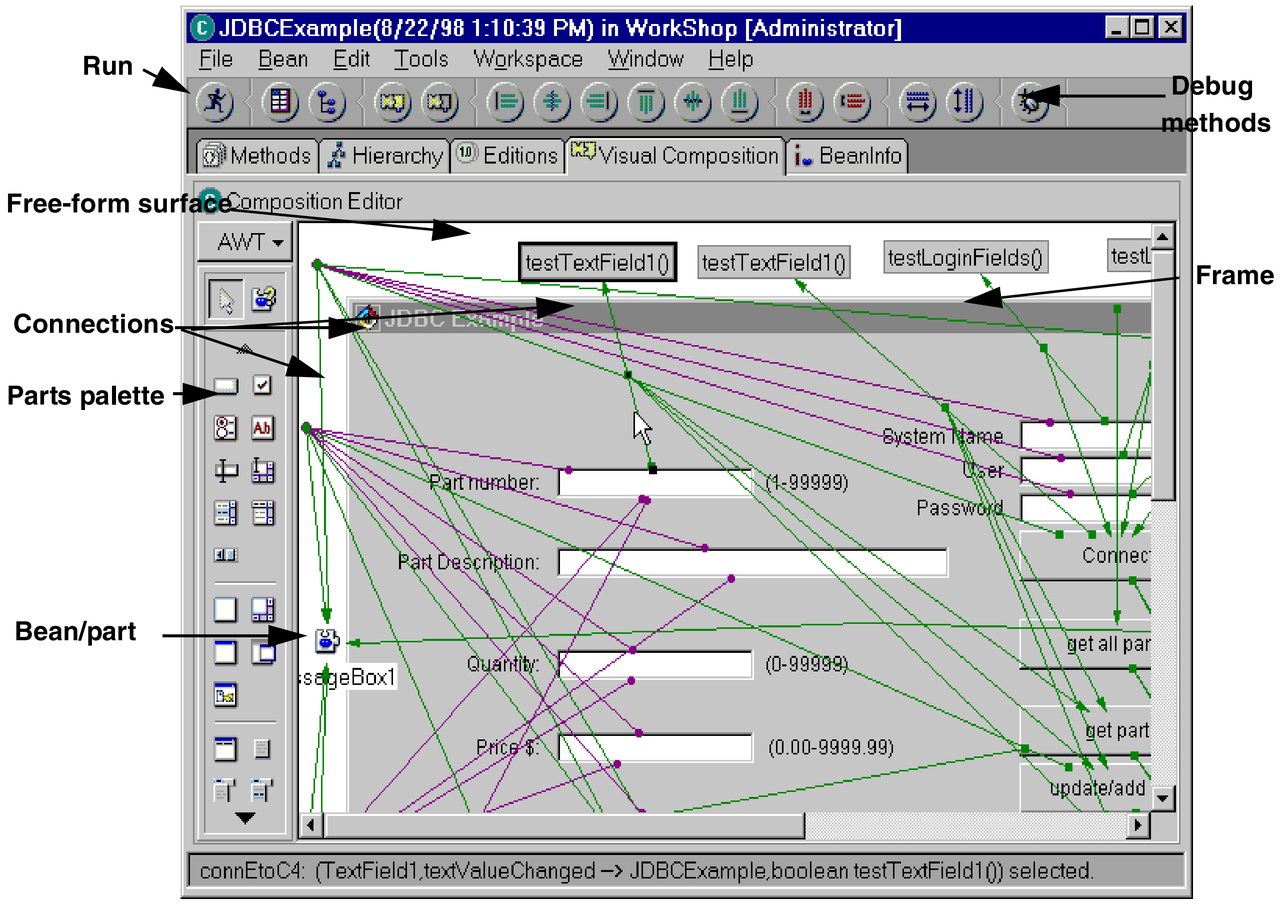
WorkShop.PartsContainer.connectToDB(java.lang.String, java.lang.String, java.lang.String) (4/30/97 4:39:4

Figure 29. Type Browser Browsing Interface

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

- **Visual Composition view**

  Provides a VCE for the design of classes (see Figure 30 on page 33).

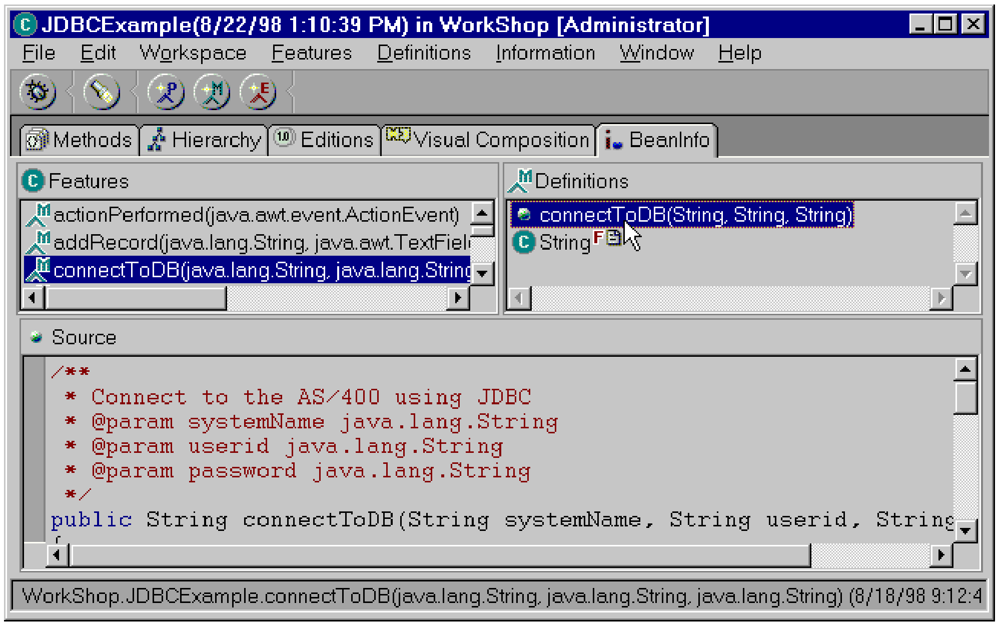


connEtoC4: (TextField1,textValueChanged --> JDBCExample,boolean testTextField1()) selected.

Figure 30. Type Browser Visual Composition View

- **BeanInfo view**

  Provides all information about the features that have been defined for the class (if any), and also allows the BeanInfo to be modified (see Figure 31 on page 34).


JDBCExample(8/22/98 1:10:39 PM) in WorkShop [Administrator]
File Edit Workspace Features Definitions Information Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Features
Definitions
Source

actionPerformed(java.awt.event.ActionEvent)
addRecord(java.lang.String, java.awt.TextFiel
connectToDB(java.lang.String, java.lang.Strin

connectToDB(String, String, String)
String

```
/**
 * Connect to the AS/400 using JDBC
 * @param systemName java.lang.String
 * @param userid java.lang.String
 * @param password java.lang.String
 */
public String connectToDB(String systemName, String userid, String
```

WorkShop.JDBCExample.connectToDB(java.lang.String, java.lang.String, java.lang.String) (8/18/98 9:12:4

Figure 31. Type Browser BeanInfo View

2.3.3 How It Fits Together

VisualAge for Java uses three basic components to build reusable JavaBeans and to use JavaBeans that may have been built by other tool vendors. These three components are the VCE, the Features editor, and the Script editor (see Figure 32).

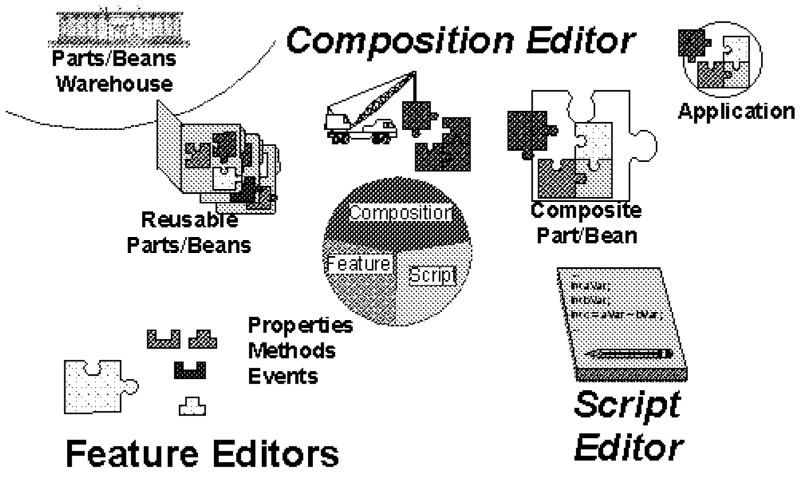


Figure 32. VisualAge for Java Concepts

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

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

2.3.3.1 JavaBeans and Classes

As discussed in Chapter 1, “Object-Oriented Technology Overview” on page 1, a class is a template for objects that have similar behavior (methods) and data elements (variables, properties). To use classes in visual builders (for example, VisualAge for Java, Symantic Cafe), the class needs to have features defined for it that allow it to be connected to other beans within a visual development environment (see Figure 33).

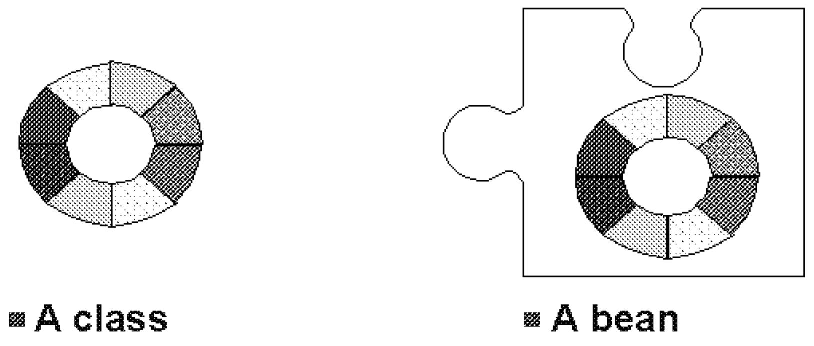


Figure 33. Class and Bean Difference

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

Individual JavaBeans vary in functionality, but most share certain common defining features. These include:

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

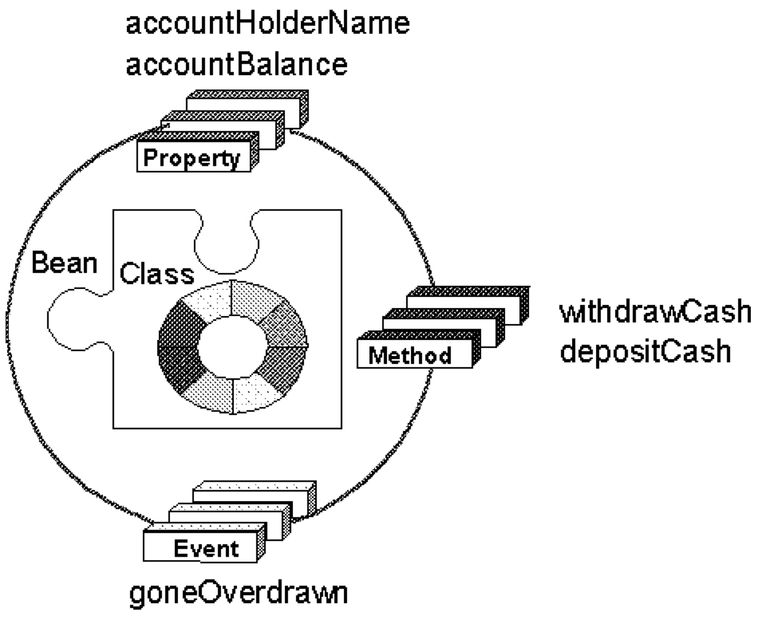


Figure 34. Account Example

In Figure 34, an account class is defined with functions/methods and variables. In addition to the account class definition, bean features are defined for the following definitions:

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

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

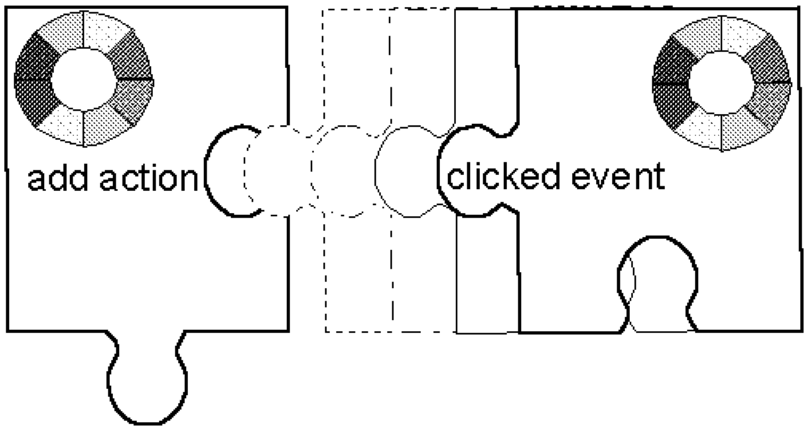


Figure 35. Example Connection of Two Beans

In Figure 35, there are two classes packaged as beans. The bean (for example, a push button) on the right side has a connection to the bean (for example, a list box) on the left. When the clicked event occurs, the list box performs the add function. In VisualAge for Java, this connection is made through a series of simple steps that connect the two beans together.

Beans can either be a single bean made up of individual beans/classes, or they can be composite beans made up of two or more classes/beans. In Figure 36, the push button is a single bean, where the window is a composite bean made up of a push button and a list. The same concept applies also to non-visual classes/beans. For example, an array contains a number of strings.

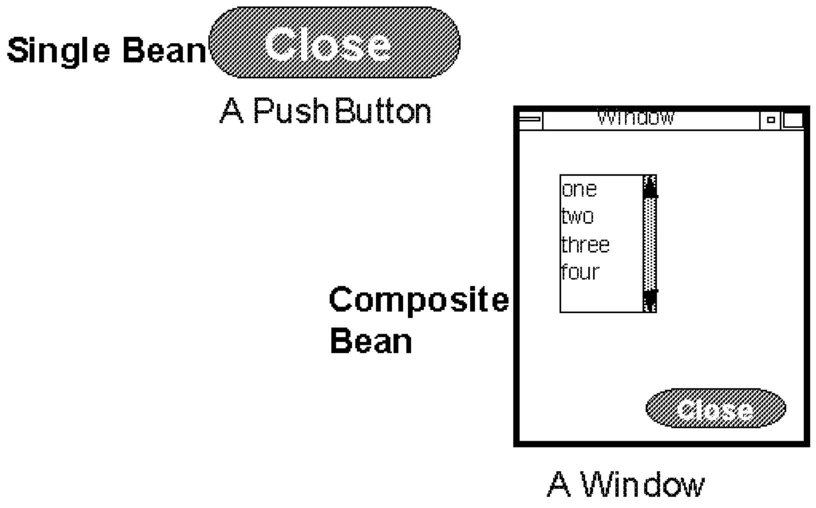


Figure 36. Single and Composite Beans Example

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

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

Table 1. Connection Types

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

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

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

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

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

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

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

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

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

2.3.4 Building a Sample Application

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

Earlier in this chapter, VisualAge for Java was started, and you navigated past the Welcome dialog window to the Workbench window.

To follow along in building the application, from the Workbench window, now perform the following steps:

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

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

   Selected—>Add—>Project...

   In your workbench window, this action appears as shown in Figure 37.

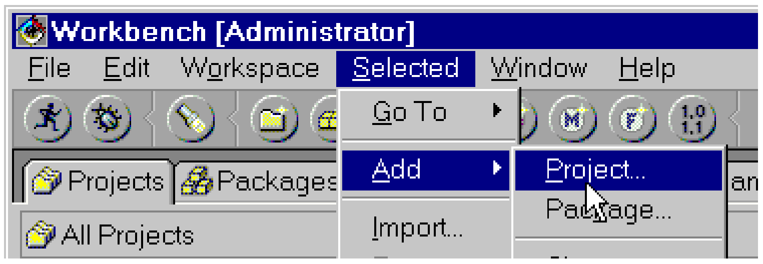


Figure 37. Add Project Selection

   The SmartGuide Add Project window is shown (see Figure 38 on page 40). Note that you can also add existing projects from a repository.

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

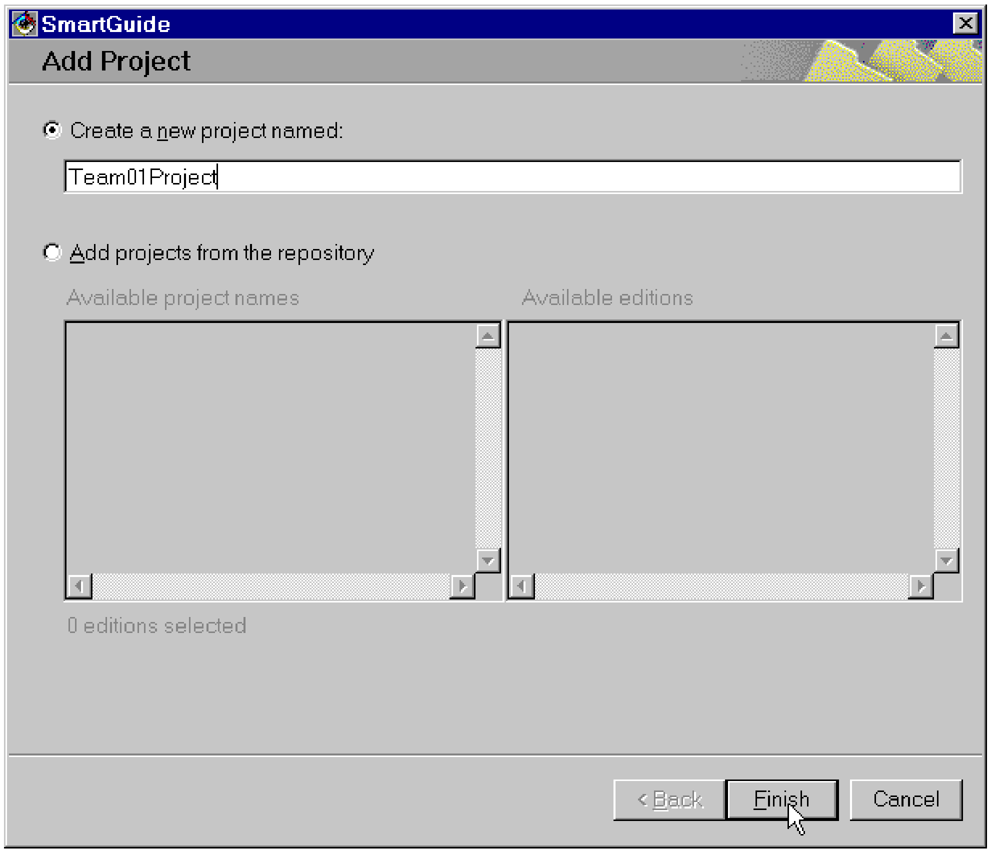


Figure 38. Add Project Example

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

2.3.4.1 Opening the Team01Project

To open the project, perform these steps:

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

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

2.3.4.2 Adding a New Package to the Team01Project Project

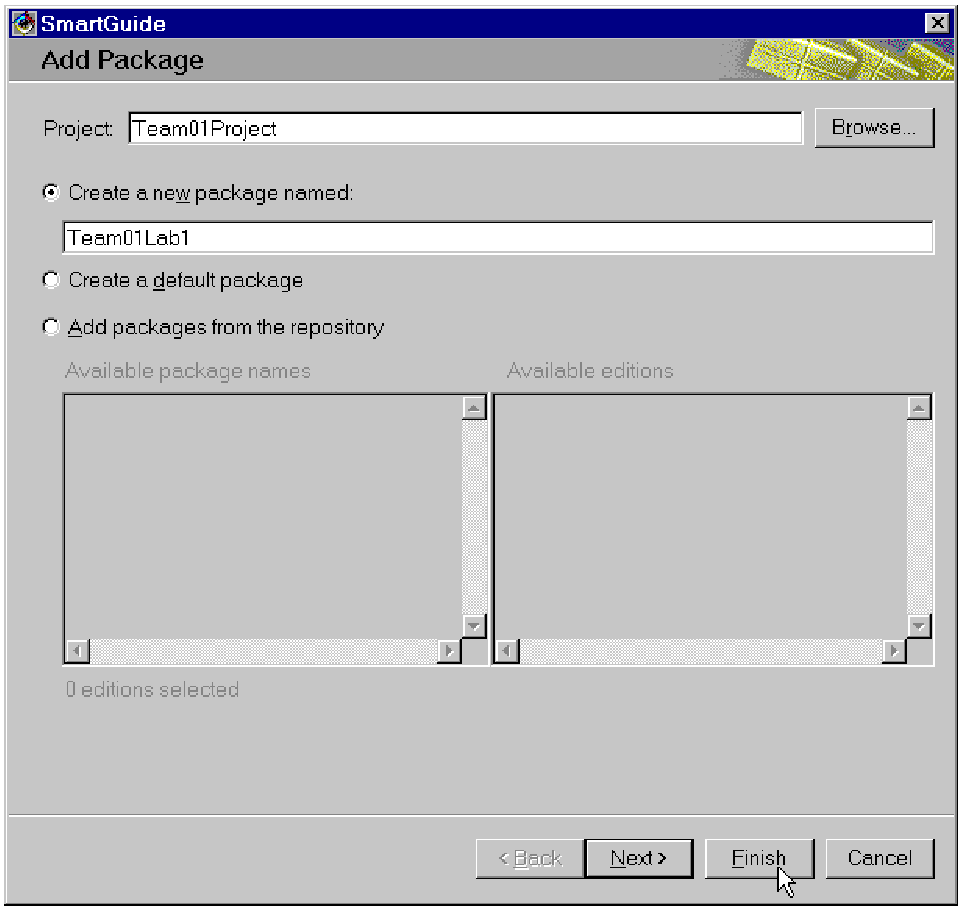
To add a new package, perform the following tasks:

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

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

3. Enter `Team01Lab1` as the package name, and click **Finish** to create it.

See Figure 39. Note that you can also add existing packages from a repository.



Project: Team01Project

(●) Create a new package named: Team01Lab1

( ) Create a default package

( ) Add packages from the repository

Figure 39. Add Package Example

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

2.3.4.3 Adding a New Class to a Package

To add a new class, follow these steps:

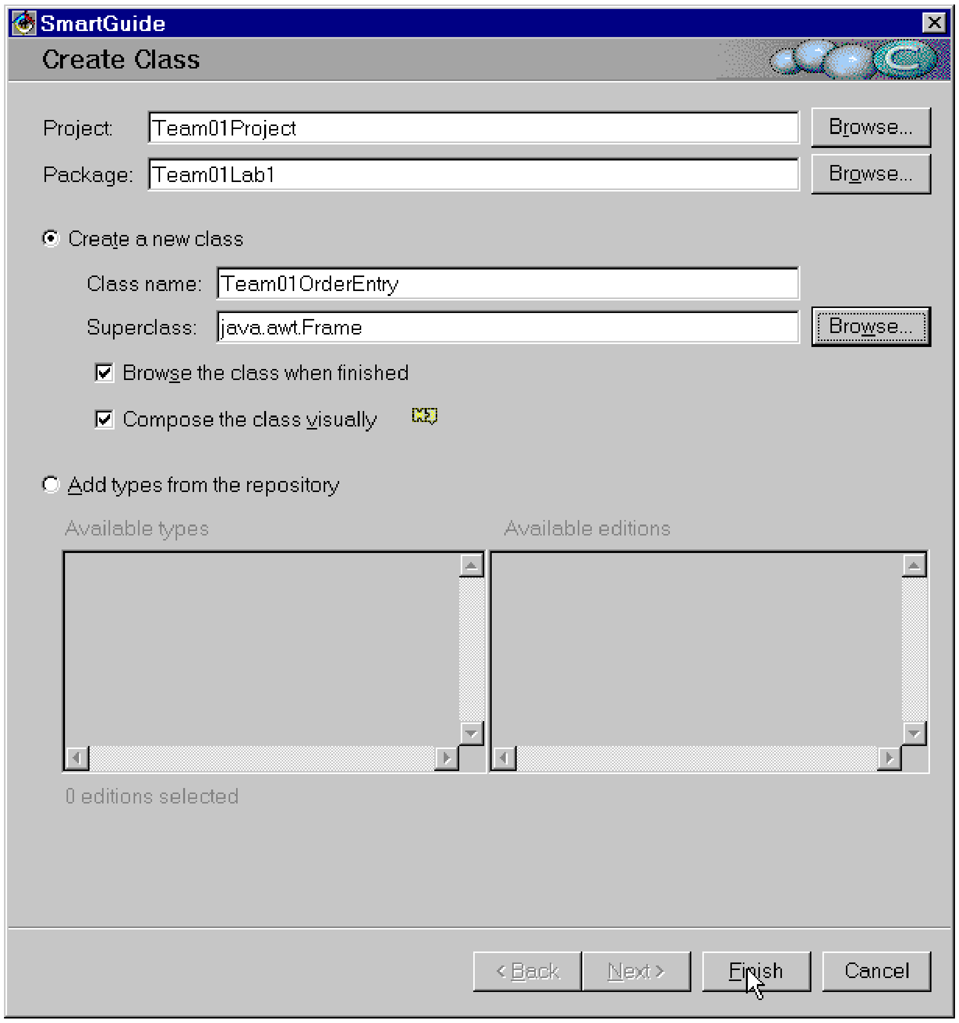
1. Select the **Types—>Add—>Class...** menu item.
2. Enter `Team01OrderEntry` as the class name.
3. Enter `java.awt.Frame` as the superclass name.

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

To create the class, perform these steps:

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

See Figure 40 on page 42.



SmartGuide

Create Class

Project: Team01Project

Package: Team01Lab1

(●) Create a new class

Class name: Team01OrderEntry

Superclass: java.awt.Frame

[✓] Browse the class when finished

[✓] Compose the class visually

( ) Add types from the repository

Available types

Available editions

0 editions selected

Figure 40. Add Class Example

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

The title of the window is Team01OrderEntry (mm/dd/yy hh:mm:ss am) in Team01 Lab1 [UserID]. The suffix time-stamp element of the window title is an indication of the date/time when this edition of the class was created. You can also see the package to which the class belongs to (see Figure 41 on page 43).

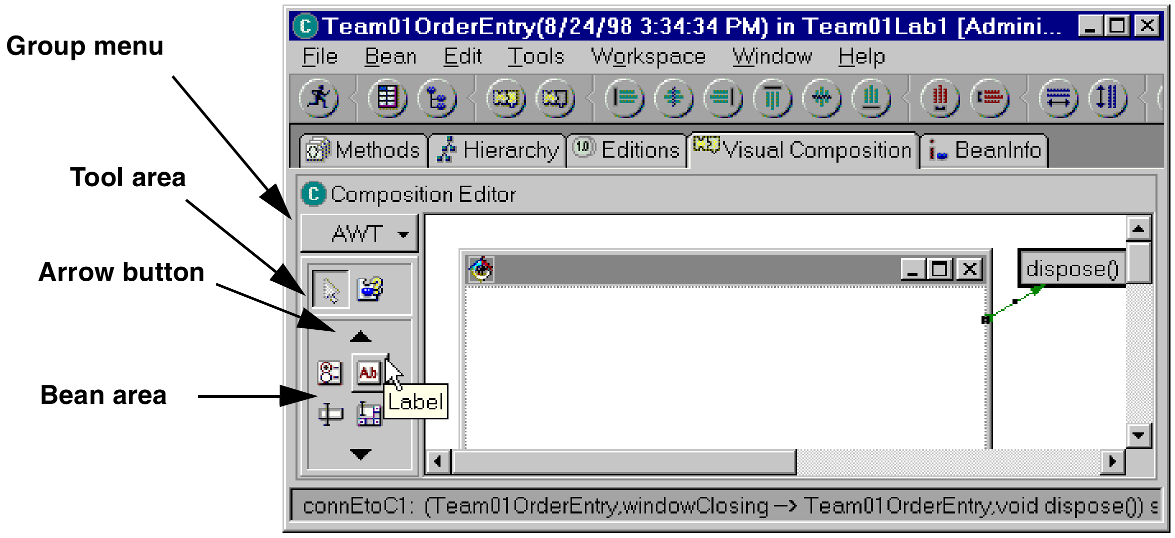

Group menu
Tool area
Arrow button
Bean area
Team01OrderEntry(8/24/98 3:34:34 PM) in Team01Lab1 [Admini...
File Bean Edit Tools Workspace Window Help
Methods
Hierarchy
Editions
Visual Composition
BeanInfo
Composition Editor
AWT
Label
dispose()
connEtoC1: (Team01OrderEntry.windowClosing --> Team01OrderEntry.void dispose()) s

Figure 41. VCE Example

Take a moment to review the window in Figure 41. There are various components in the VCE:

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

2.3.4.4 Adding the Visual Components to the Window

The completed application for this section looks similar to Figure 42.

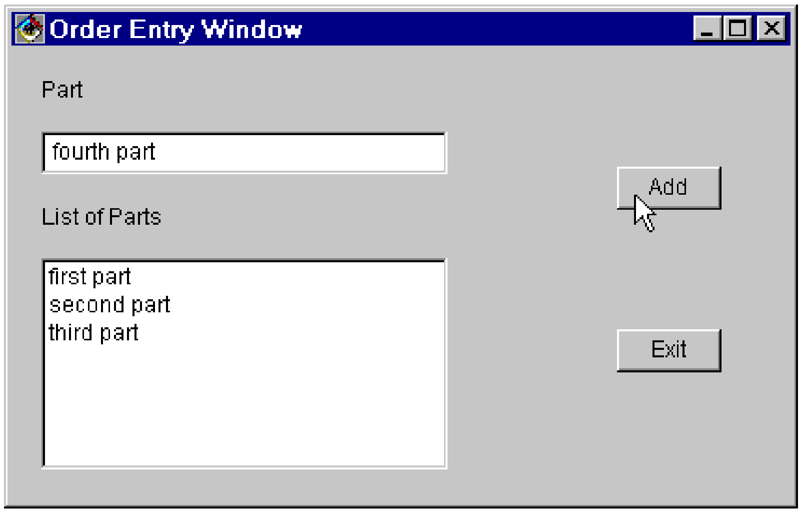


Figure 42. Layout Example

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

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

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

Next, we build the graphical user interface by selecting beans from the parts palette and placing them on the window. Use the completed window shown in Figure 42 as a guide.

To build the interface, perform the following steps:

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

   Note: Use the tool tip to recognize the beans in the parts palette. Move the mouse pointer over the top of the part and view the on-line help.
2. Make sure that the AWT category from the category selection is selected (see Figure 43 on page 45).

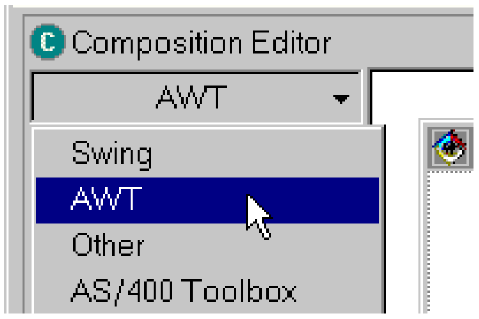


Figure 43. Composition Editor Selecting a Bean Category Example

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

1. Move the cursor over the bean area and left-click on the TextField bean.

   This loads the cursor with the TextField bean. Note the category and bean name indicated on the status line and the cursor changing into a crosshair when moved into the design area (see Figure 44).

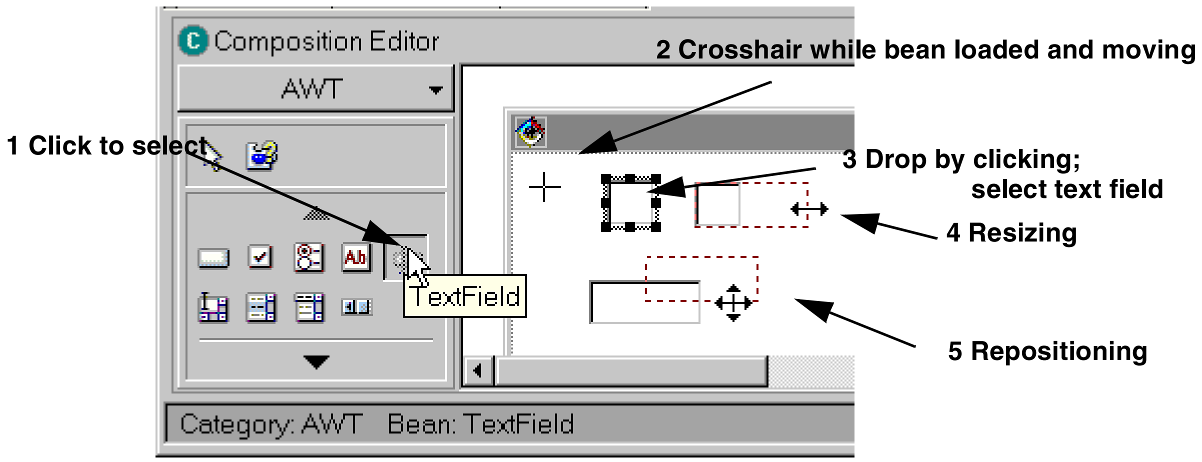

Composition Editor
AWT
1 Click to select
2 Crosshair while bean loaded and moving
3 Drop by clicking; select text field
4 Resizing
5 Repositioning
TextField
Category: AWT Bean: TextField

Figure 44. Composition Editor Creating a Text Field Example

2. Move the cursor into the design area, onto the frame, near to the upper left edge. Left-click to drop the TextField into position.

The field is now drawn inside the frame and selected. You can recognize a selected object by its resize handles (see Figure 44). To deselect an object, simply click onto an empty spot in free-form surface around the frame.

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

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

   The cursor changes into a two-headed arrow, as soon as you are in the resizing mode (see Figure 44).

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

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

The cursor changes into a four-headed arrow, as soon as you are in the repositioning mode (see Figure 44 on page 45).

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

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

   The result of these actions is a new text field on the frame (see Figure 44 on page 45).

To add the buttons, perform these tasks:

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

To add the list, follow these steps:

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

To add the label, complete these steps:

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

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

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

Becoming familiar with shortcut keys improves your speed in visual programming greatly. The most frequently used actions all have shortcut keys (see Figure 45 on page 47).

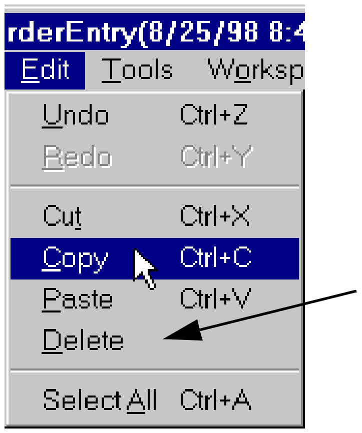


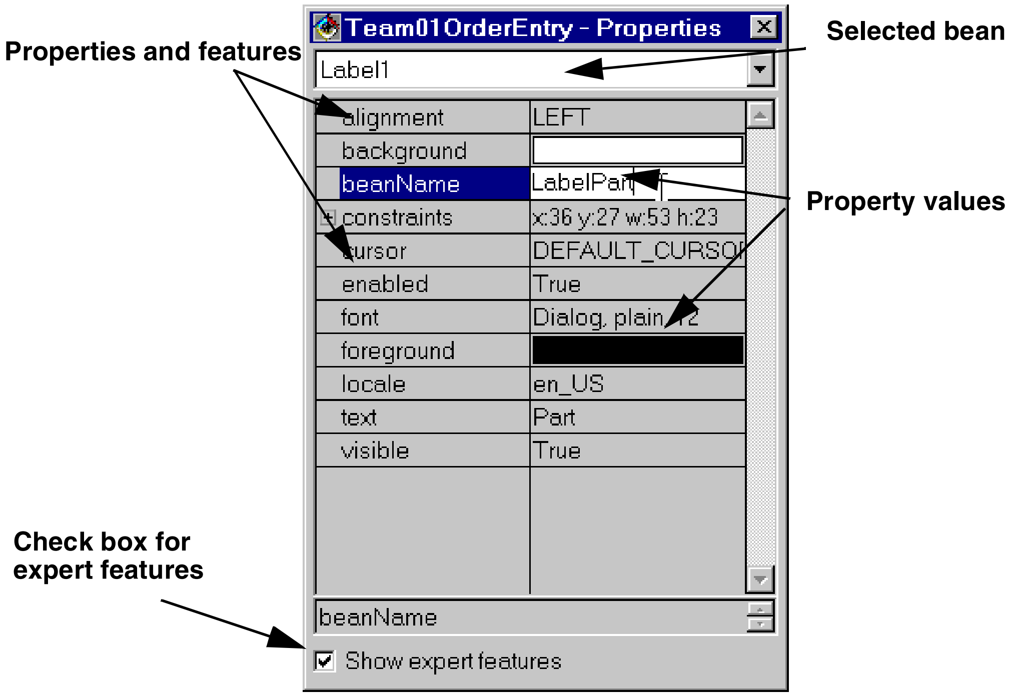
Figure 45. Composition Editor Shortcut Keys

2.3.4.5 Making the Frame Look Good

In this section, we improve the appearance of the frame. First, we work with the labels. Move the cursor over the Label1 just above the TextField1 and double-click.

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

Note: Whenever you cannot find a property, make sure that the Show expert features check box is checked (see Figure 46).



Label1

alignment	LEFT
background	
beanName	LabelPart
constraints	x:36 y:27 w:53 h:23
cursor	DEFAULT_CURSOR
enabled	True
font	Dialog, plain, 12
foreground	
locale	en_US
text	Part
visible	True

beanName

Figure 46. Composition Editor Properties Window Example

Now we are going to change two property values for bean Label1 in the properties window. Change the text Label1 to "Part" and the bean name Label1 to "LabelPart".

Note: It is helpful to change a bean's name. It is much easier to search for a LabelPart than for a Label9 out of 15 labels inside the Java source code, which is generated later. Giving the label a bean name that refers to the corresponding object makes it also easier to determine which label belongs to which object. This applies to any bean you build in the VCE, as well as for buttons, text fields, frames, and so on.

To change the Label1 property value, perform these steps:

1. Click on the property value for the text property.
2. Overwrite Label1 with "Part".
3. Click on the property value for beanName.
4. Overwrite Label1 with "LabelPart".
5. Click on the X on the upper right corner to close the property window.

When changing the text property, you see a small button on the right end of the property value. If pressed, it brings up the text property editor, which can be used to externalize strings. We do not need this option in our examples. Simply overwriting the text does the job for us. Note that for different property types, you have different property value editors. For example, consider a color property value editor, a boolean property value editor, or a font property value editor (see Figure 47).

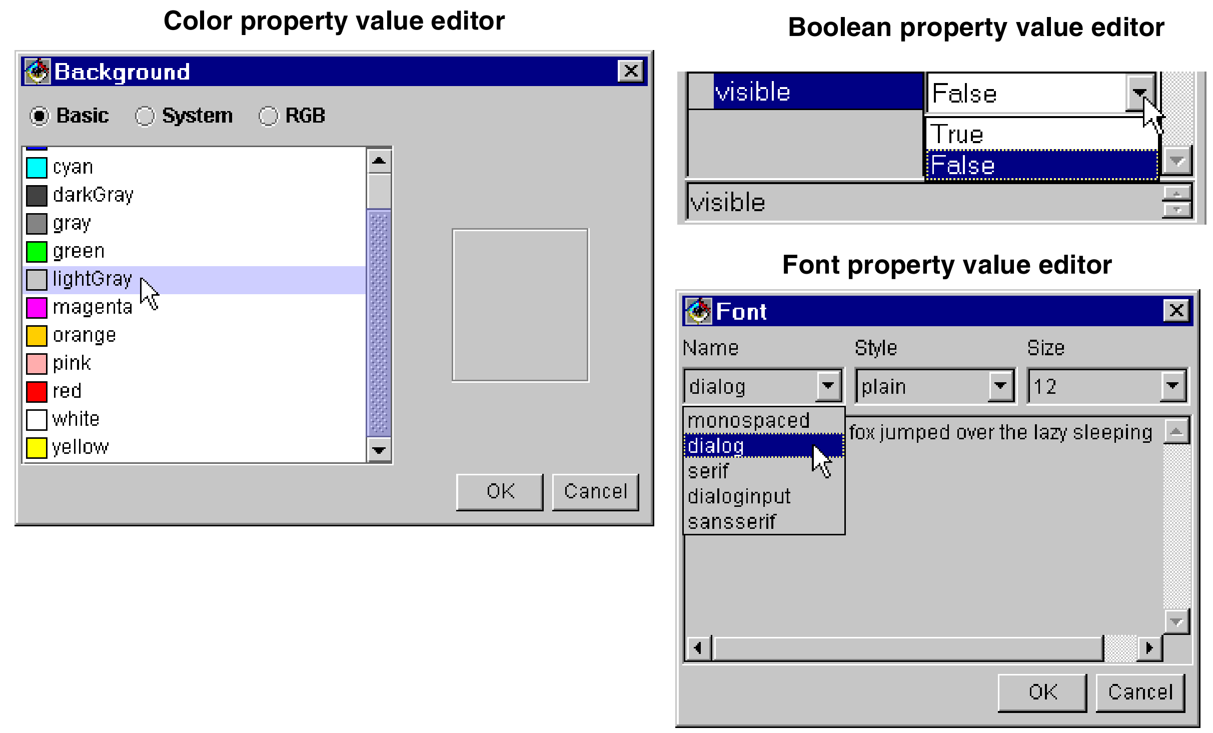


Figure 47. Composition Editor Property Editor Examples

The label we changed before should now appear as Part and have the name LabelPart. Check this by selecting (clicking) the bean again, and see which text appears in the status line. Following the same procedure, you can now change all the other beans in the design area. Keep the property window open while you select one bean after the other. The changed values are saved, even if the property window is not closed. To change the background color, use the color property value editor as shown in Figure 47 on page 48.

To complete property changes, perform these steps:

1. Open the property window for the Team01OrderEntry (double-click on the title bar of the frame).
2. Change its title to "Order Entry Window".
3. Change its background to "lightGray".
4. Select TextField1 and change its beanName to "Part".
5. Change its background to "White".
6. Select the Label2 and change its beanName to "LabelListParts".
7. Change its text to "List of Parts".
8. Select the List1 and change its beanName to "ListParts".
9. Change its background to "White".
10. Select the Button1 and change its beanName to "ButtonAdd".
11. Change its label to "Add".
12. Select the Button2 and change its beanName to "ButtonExit".
13. Change its label to "Exit".

After this exercise, your frame looks almost the same as the one in Figure 42 on page 44. It is a good idea to save the work now. To save your work, perform either of the following options:

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

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

Switch back to the VCE view since there is some resizing, repositioning and alignment work left to do. Look at the tool bar before completing the frame (see Figure 48).

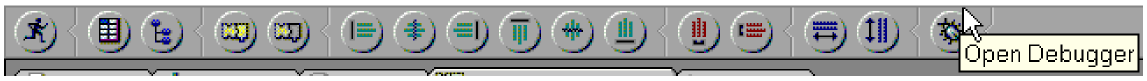


Figure 48. Composition Editor Tool Bar

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

You can select multiple objects when sizing, positioning, or aligning the components. To select multiple objects, hold down the Ctrl key and left-click sequentially all of the desired objects. If you are working with multiple object selections, the last selected object is always the reference object for all of the other objects in your selection. The size and position of the reference object affects the size and position of all the other objects in your selection. This applies when you are resizing or repositioning the reference object or when you are using one of the alignment, distribution, or match tools. You recognize the reference object by its filled (black) resize handles, while the other objects in the selection have hollow (white) handles (see Figure 49).

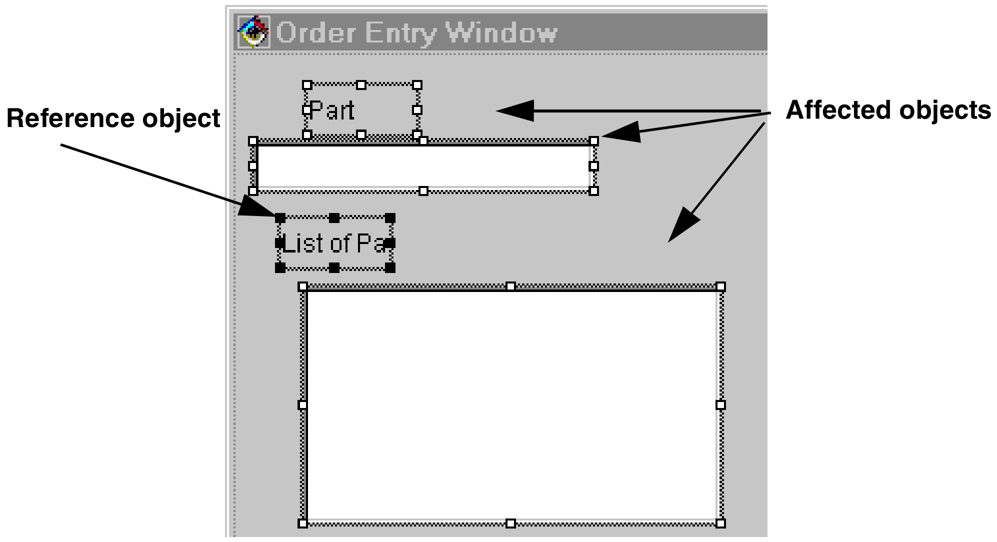


Figure 49. Composition Editor Multiple Bean Selection Example

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

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

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

To align the beans, perform these steps:

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

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

To resize the ListParts label, follow this sequence:

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

To align the buttons, complete these steps:

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

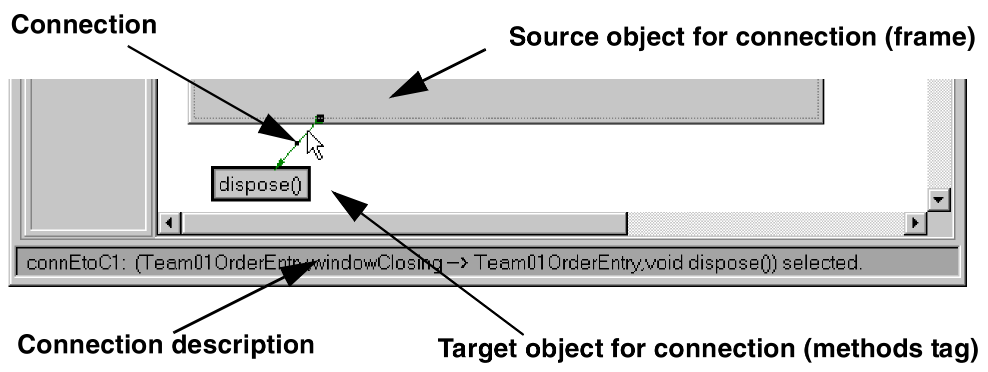
To run the application, perform the following tasks:

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

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

2.3.4.6 Adding the Function

First, you need to make the Exit button work. If a user presses the Exit button, the frame should close in the same way as if the X button in the top right corner of the frame was clicked. The only visible connection at this time provides exactly that function. To find out what a connection stands for, left-click on the line connecting the two objects together and view the text displayed in the status line (see Figure 50). Select the connection from the frame bean to the dispose() method.


Connection
Source object for connection (frame)
dispose()
Connection description
Target object for connection (methods tag)

connEtoC1: (Team01OrderEntry,windowClosing --> Team01OrderEntry,void dispose()) selected.

Figure 50. Composition Editor Connection Example

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

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

Select the frame as the target in this example. As soon as you select that object, the connection target pop-up window is displayed. From this window, choose the desired feature of the target object. In this case, you want the frame to be closed or disposed, so select the dispose() feature. As a result of this, a green connection is displayed between the Exit button and the frame (see Figure 51 on page 52).

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

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

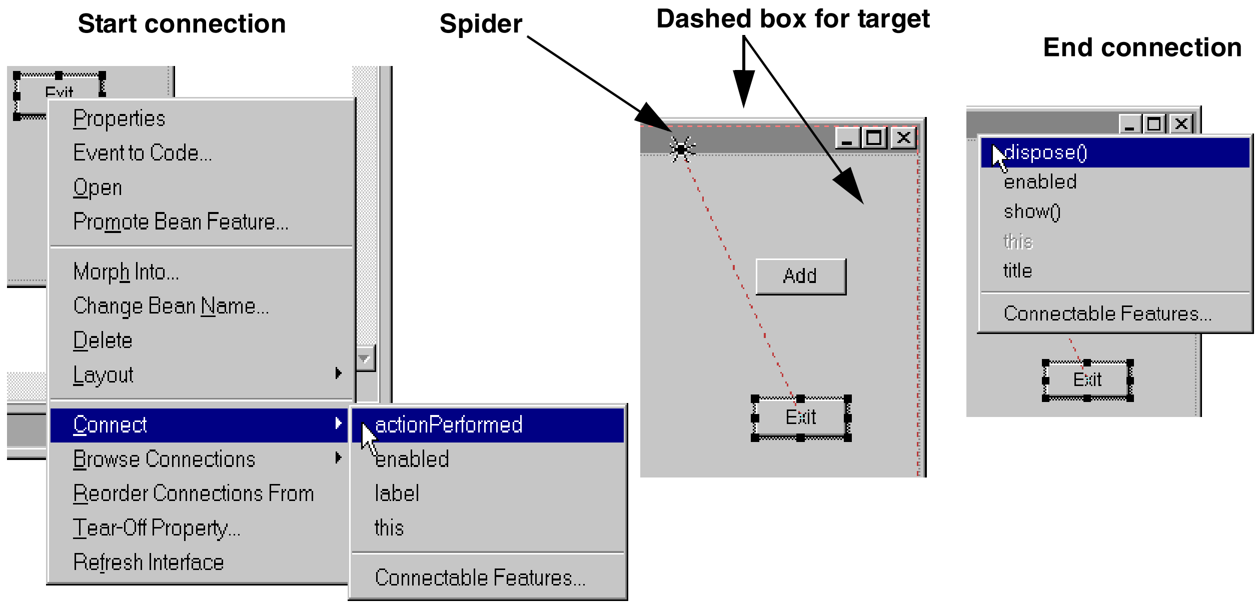
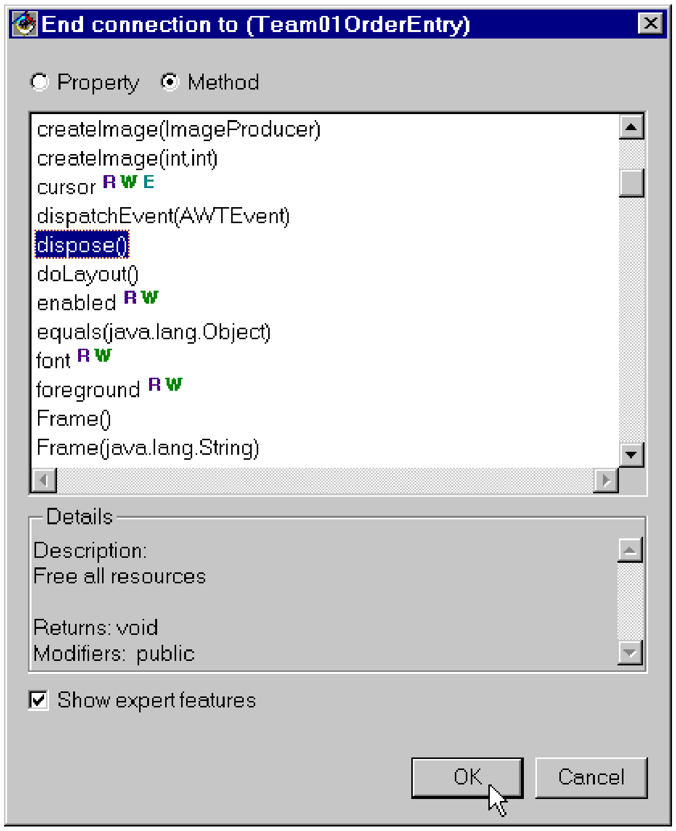


Figure 51. Connecting Example

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

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

1. Click on the connection from the Exit button to the frame.
2. Press **Delete** to delete the connection.
3. Select the **Exit button** and right-click to bring up its pop-up menu.
4. Select **Connect—>actionPerformed**.
5. Move the spider to an empty spot in the free-form surface, and left-click on it.
6. From the target feature pop-up window, select **Connectable Features...** and choose **dispose()**. See Figure 52 on page 53.
7. Click on **OK**.



End connection to (Team01OrderEntry)

( ) Property [✓ Method]

createImage(ImageProducer)
createImage(int,int)
cursor R W E
dispatchEvent(AWTEvent)
dispose()
doLayout()
enabled R W
equals(java.lang.Object)
font R W
foreground R W
Frame()
Frame(java.lang.String)

Details
Description:
Free all resources

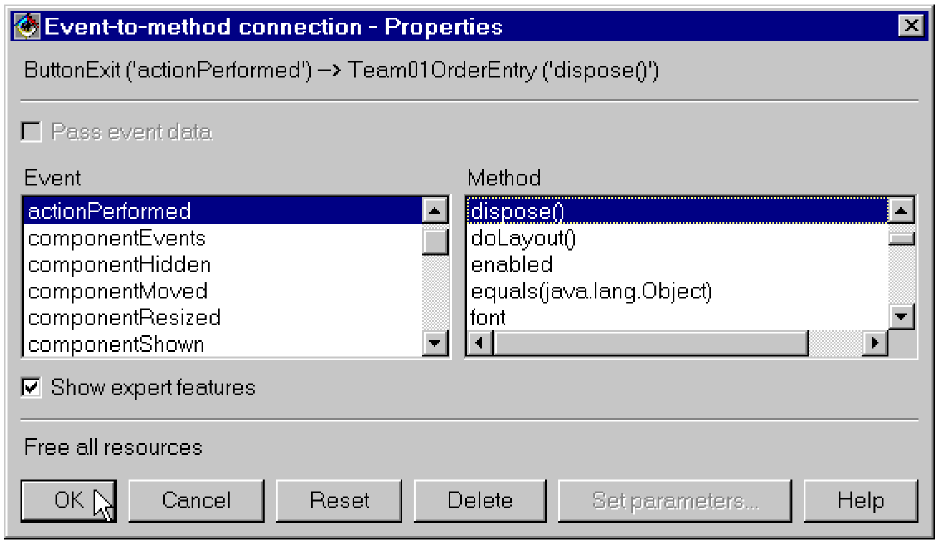
Returns: void
Modifiers: public

[✓] Show expert features

Figure 52. Connectable Features Example

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

Note, that depending on the type of bean selected, you can choose between the properties, methods, and events that you want to connect. If you cannot find a feature, make sure that the *Show expert features* check box is checked. You can also change an existing connection by double-clicking on it. This action brings up the connection properties window. There you can change source and target features and set parameters for the connection if required. Note that this window also has a check box for expert features (see Figure 53 on page 54).

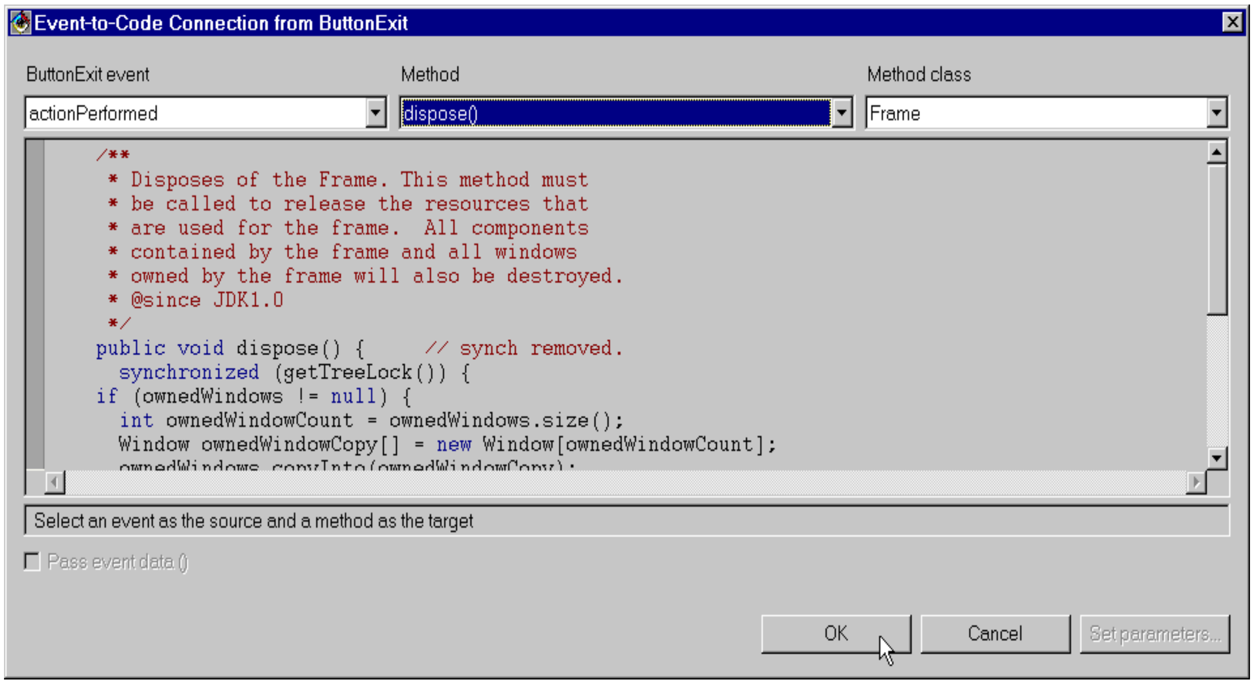

Event-to-method connection - Properties

ButtonExit ('actionPerformed') --> Team01OrderEntry ('dispose()')

Free all resources

Figure 53. Connection Property Editor Example

The third way to create a connection is to connect a source feature directly to the methods source code. This can be achieved by selecting Event to Code... from the connection target pop-up window. That brings up a source editor from which you can select the source event, the target method, and the target methods class. For the frame that you are designing, you can also create a new method for the target feature in the connection. The source editor also appears when you double-click on a methods tag (grey box with methods name) inside the design area. For all details of the source editor, see Figure 54.


Event-to-Code Connection from ButtonExit

ButtonExit event: actionPerformed
Method: dispose()
Method class: Frame

```
  /**
   * Disposes of the Frame. This method must
   * be called to release the resources that
   * are used for the frame.  All components
   * contained by the frame and all windows
   * owned by the frame will also be destroyed.
   * @since JDK1.0
   */
  public void dispose() {     // synch removed.
    synchronized (getTreeLock()) {
  if (ownedWindows != null) {
    int ownedWindowCount = ownedWindows.size();
    Window ownedWindowCopy[] = new Window[ownedWindowCount];
```

Select an event as the source and a method as the target

Figure 54. Connection Source Editor Example

To create the connection again, perform the following steps:

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

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

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

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

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

   The connection target pop-up window appears.

What happens when the Add button is pressed? You want to add a text/string to the list that was entered in the TextField. Select add(String) from the pop-up window. A dashed green connection is displayed between the Add button and the list.

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

1. Move the cursor over the dashed green connection from the Add button to the ListParts.
2. Left-click over the connection to select it. Selection handles are shown along the connection to show that it has been selected.
3. With the cursor still over the connection (but not on a selection handle), right-click to bring up the Connections pop-up menu.
4. Select **Connect—>Item**.

5. Move the spider over **Part**.
6. Left-click and select text.

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

You have now completed the window for this section. Test it out by pressing the Run smarticon. Enter some values in the Part, and check if the Add button adds them to the ListParts. Also try using the Exit button. Figure 55 shows the completed VisualAge frame (your window may look similar).

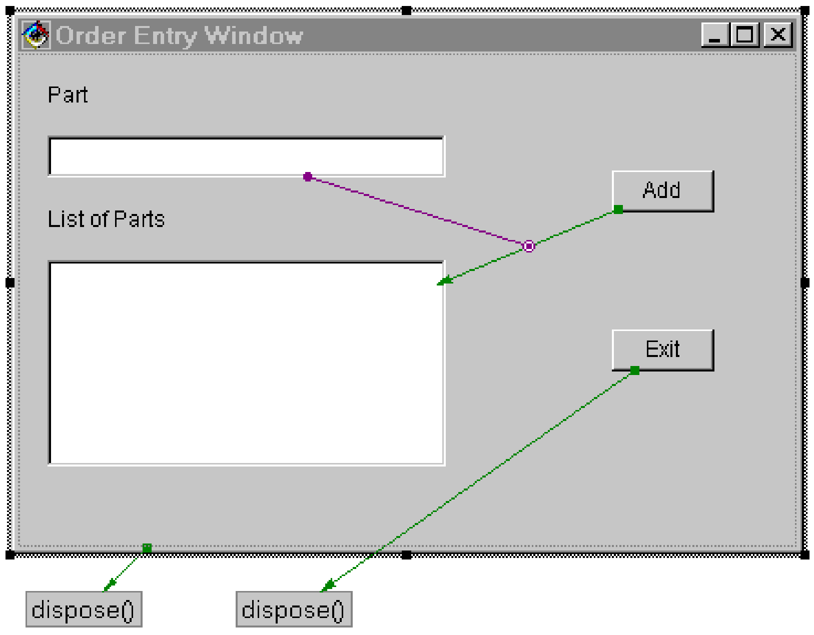


Figure 55. VCE Example

2.3.4.7 Creating a Version for Your Application

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

To version the application, perform these steps:

1. Left-click on the **Hierarchy** tab.

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

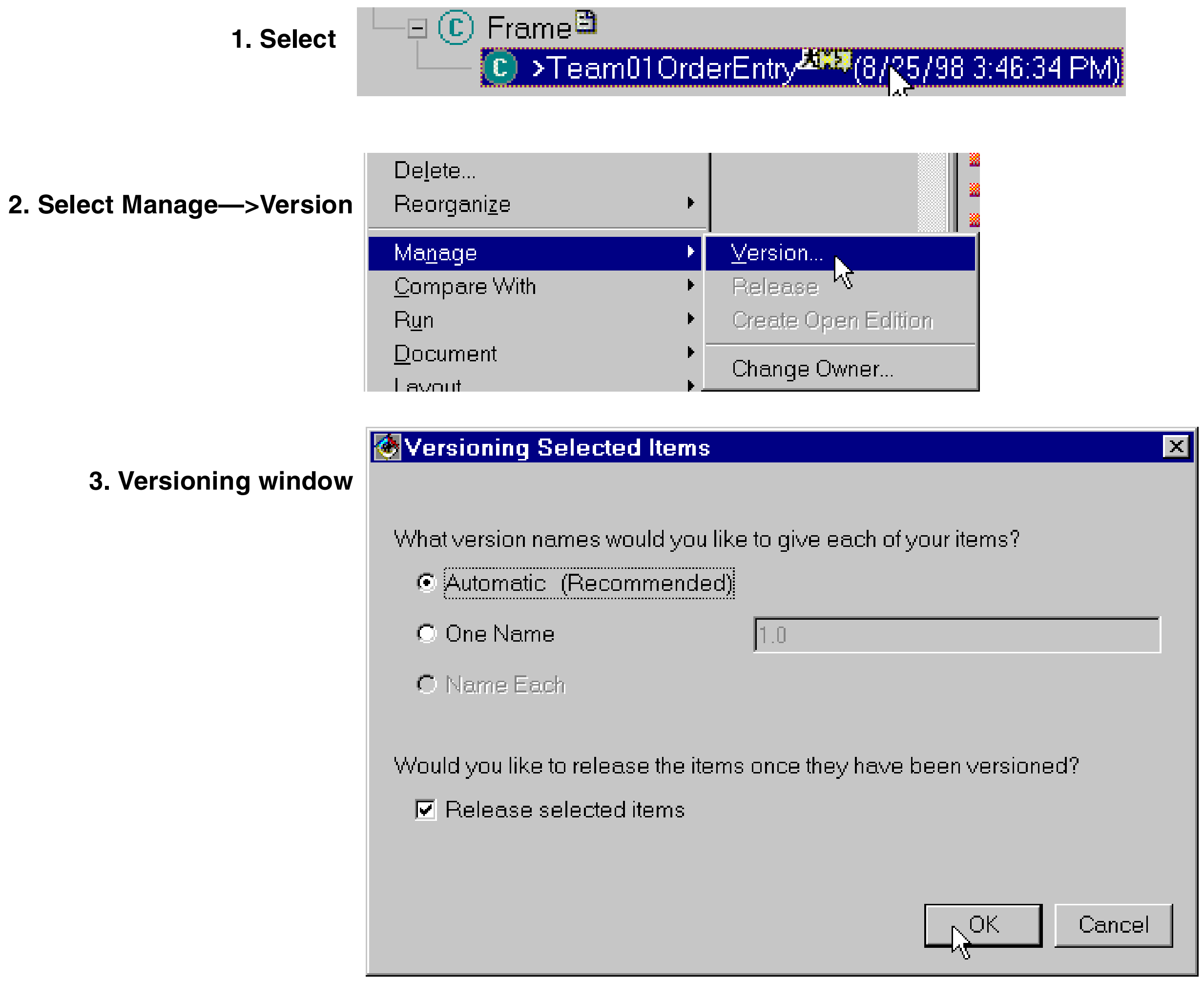
2. Press the **Show Edition Names** smarticon.

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

3. Left-click on **Team01Lab1.Team01OrderEntry** to select it.

4. Select the **Classes—>Manage—>Version...** menu item

5. Make sure the **Automatic** radio button is selected and press **OK** (see Figure 56 on page 57).



What version names would you like to give each of your items?

[✓ Automatic (Recommended)] One Name 1.0 Name Each

Would you like to release the items once they have been versioned?

[✓ Release selected items]

Figure 56. Create Version Example

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

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

To create an open edition, complete these tasks:

1. Left-click on **Team01Lab1.Team01OrderEntry** to select it.
2. Select the **Classes—>Manage—>Create Open Edition** menu item from the menu bar.

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

2.3.4.8 Extending the Application

The application is now extended. Perform the following actions:

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

The completed development window should appear similar to the window shown in Figure 57.

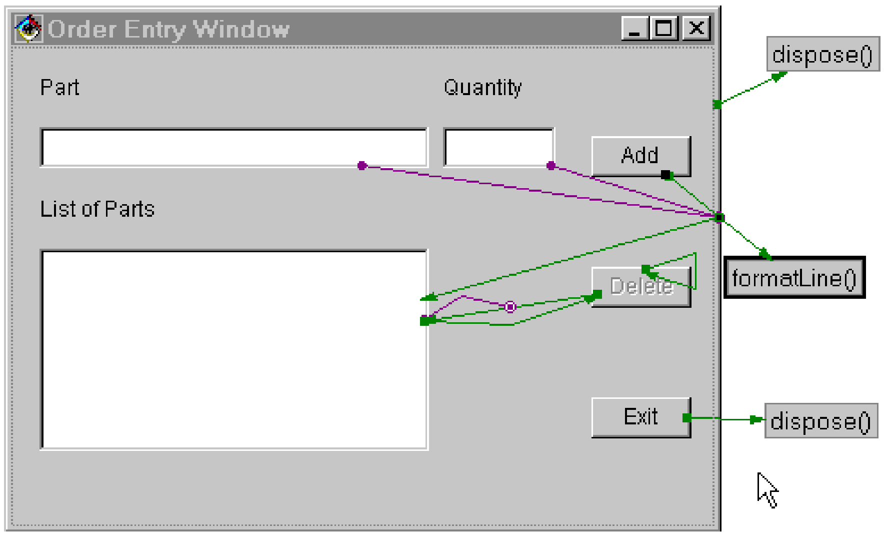


Figure 57. VCE Finished Example

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

1. Add a Delete button to the window:
   a. Select the **Button bean** from the bean area and drop it between the ButtonAdd and the ButtonExit.
   b. Double-click this button to bring up its properties window.
   c. Change the beanName property to **ButtonDelete**.
   d. Change the label property to **Delete**.
   e. Change the enabled property to **false**.
   f. Allow the properties window to remain open.
2. Add a TextField that allows the quantity of parts to be input (called the Quantity) later:
   a. Select the **Text Field bean** from the bean area and drop it level with and a little to the right of the Part.
   b. Resize and reposition this field until you are satisfied with the result.

   c. In the properties window, change the beanName property to "Quantity" and the background property to "white".

      Note: Do not close the properties window.

3. Add a label and change its text property to Quantity:

   a. Select the **Label bean** from the bean area, and drop it above the Quantity.

   b. In the properties window, change the beanName property to "LabelQuantity" and the text property to "Quantity".

   c. Left align the LabelQuantity with Quantity, and middle align it with Part.

   d. Close the properties window.

You can add items from the text fields Part and Quantity to the ListParts by using a script in this section. Therefore, the current connection from the ButtonAdd (actionPerformed) to the ListParts (add(String)), taking the text property from the Part, is no longer needed. To delete it, follow this sequence:

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

The connection and any connections it supported are deleted.

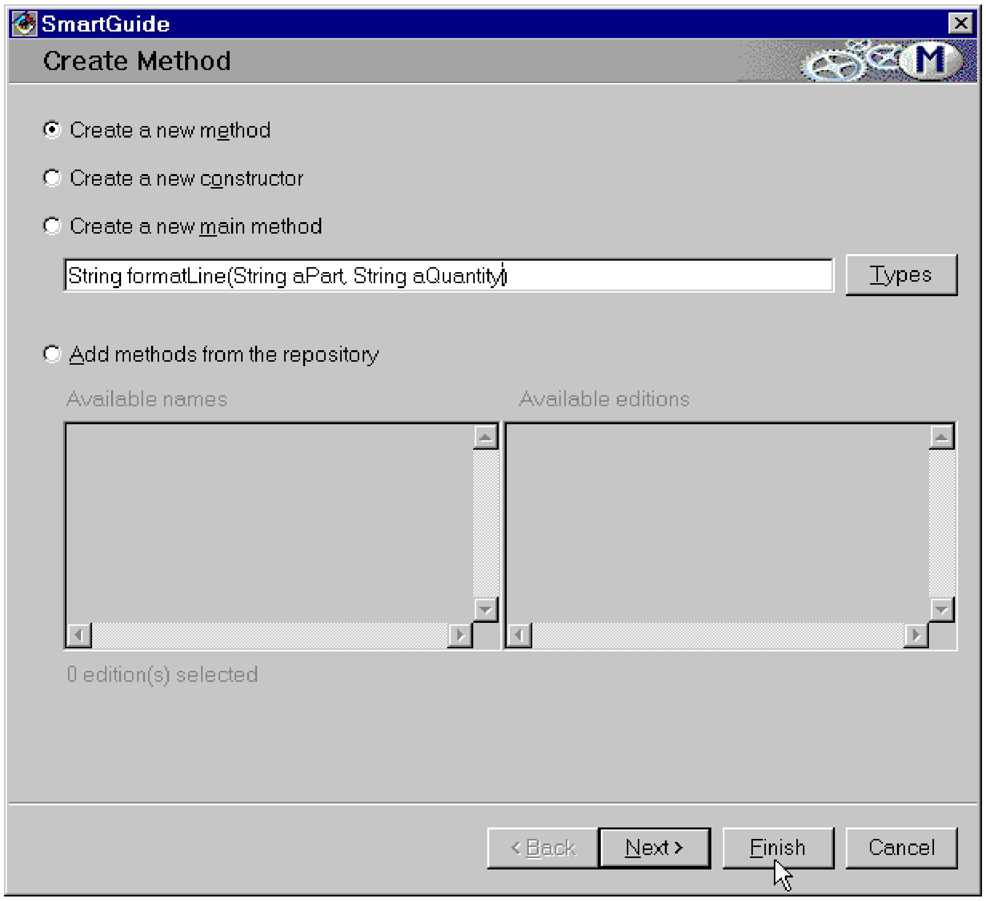
2.3.4.9 Writing a New Java Method

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

To create a new method, perform these tasks.

1. Click on the **notebook** tab for Methods.
2. On the tool bar, press the **Create Method** or **Constructor** smarticon.
3. Check the **Create a new metho**d radio button.
4. In the **Method Name entry field** of the Method Properties window, type `String formatLine (String aPart, String aQuantity)`.
5. Click on the **Finish** button.

See Figure 58 on page 60.


SmartGuide

Create Method

[✓ Create a new method]

Create a new constructor

Create a new main method

String formatLine(String aPart, String aQuantity)

Add methods from the repository

Available names

Available editions

0 edition(s) selected

Figure 58. Create Method Example

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

To update the formatLine method, complete these steps:

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

The method's source should appear as shown in the following example:

```
/**
 * This method was created in VisualAge.
 * @return java.lang.String
 * @param aPart java.lang.String
 * @param aQuantity java.lang.String
 */
public String formatLine(String aPart, String aQuantity) {
   return aPart + " " + aQuantity;
}
```

Make sure that your source code matches this example. Left-click on the VCE notebook tab to return to editing the frame.

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

1. Select the **ButtonAdd** and right-click.
2. From the popup menu, select **Connect—>actionPerformed**.
3. Move the cursor to the free-form surface and left-click.
4. Select **Event to Code...** from the pop-up menu.
5. In the source editor, select **formatLine(String, String)** from the **Methods** drop-down list.
6. Click on **OK**.

Complete the last step as shown in Figure 54 on page 54, but choose the Method Class Team01OrderEntry instead of Frame, and the Method formatLine(String, String) instead of dispose(). A green dashed connection is shown from the ButtonAdd to the methods tag formatLine() on the free-form surface. A dashed connection means that the connection requires parameters that have not yet been supplied. For this reason, you have to connect the text properties of Part and Quantity to the connection by using these steps:

1. Select the dashed connection from the ButtonAdd to the formatLine(String, String) methods tag and right-click on it.
2. From the pop-up menu, select **Connect—>aPart** and move the spider over the Part.
3. Left-click and select the text.
4. Select the connection again and right-click on it.
5. From the pop-up menu, select **Connect—>aQuantity** and move the spider over Quantity.
6. Left-click and select the text.

This provides the new method formatLine(String, String) with its input parameters. Now, you have to add the returned concatenated string to the ListParts using these steps:

1. Select the connection again and right-click on it.
2. From the popup menu, select **Connect—>normalResult** and move the spider over ListParts.
3. Left-click and select **add(String)**.
4. Left-click the **run smarticon** to test the application.

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

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

1. Left-click on the **Delete button** to select it, and right-click to bring up its pop-up menu.
2. Select **Connect—>actionPerformed**.
3. Move the spider over ListParts.
4. Left-click and select **remove(String)**.

There is a new dashed green connection from the Delete button to the Parts list.

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

1. Left-click on the **Parts list** to select it.
2. Right-click and select **selectedItem property** from the popup menu.
3. Move the spider over the middle of the connection from the ButtonDelete and ListParts. A small dashed box indicates that here is a possible end-point for the connection.
4. As soon as the small dashed box appears in the middle of the connection, left-click and select the item feature.

   The dashed green line should have changed to a solid green line.

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

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

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

1. Select **ButtonDelete**, and right-click. Select **Connect—>actionPerformed**.
2. Move the spider over **ButtonDelete**, and left-click. Select **enabled**.
3. Double-click on the dashed green connection.
4. Press the **Set** parameters button.
5. Set the value to "False" and press **OK** twice.

In case the dashed green line does not change into a solid green line, set the value to True and confirm by pressing OK. After that, change the value back to False and confirm again. The line should now appear green and solid. This little inconvenience does not have any effect on the way your class behaves. The enabled property is set to false in any way, even if there is a dashed line.

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

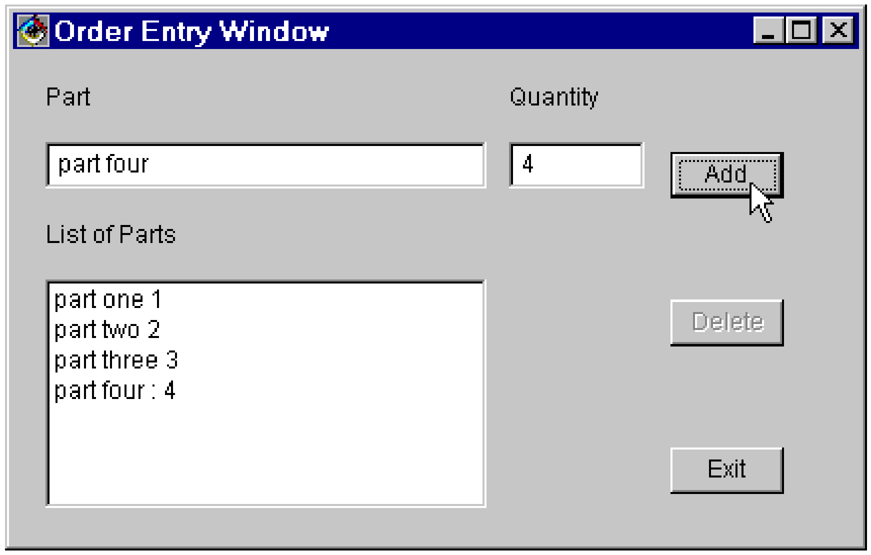
2.3.4.10 Debugging, Setting Breakpoints, and Changes 'On the Fly'

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

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

1. Return to the VCE.
2. Press the **Methods** notebook tab to switch to the methods view.
3. Select the method **formatLine(String, String)** by left-clicking on it.
4. In the source pane, change the line that returns the concatenated string to:
   `return aPart + " : " + aQuantity;`
5. Press **Ctrl+S** to save the method.
6. Select the test window and add another part or quantity item.

The code you changed was used to add this new part. Your test window should look similar to the one shown in Figure 59.



```
part one 1
part two 2
part three 3
part four : 4
```

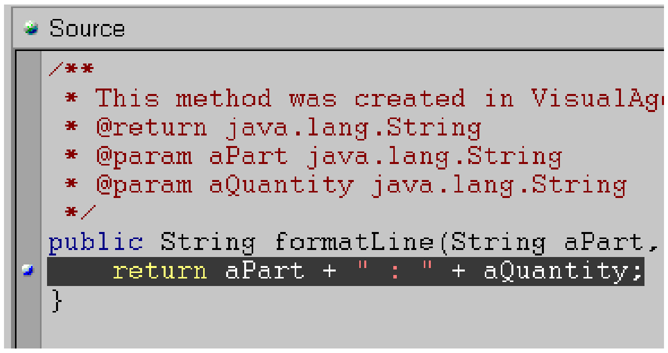
Figure 59. Finished Application Example

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

To set a breakpoint, follow this sequence:

1. Return to the Methods view of the type browser.
2. Edit the method **formatLine(String, String)**.
3. Move the cursor to the line that returns the concatenated string.
4. Right-click the mouse and select **Breakpoint**.

A blue breakpoint marker is shown on the left of the line. This is the point where the code stops prior to executing it and opens up a debugger window. If the blue breakpoint marker does not appear, you probably were not in the first column or you were on an incorrect line (see Figure 60 on page 64).

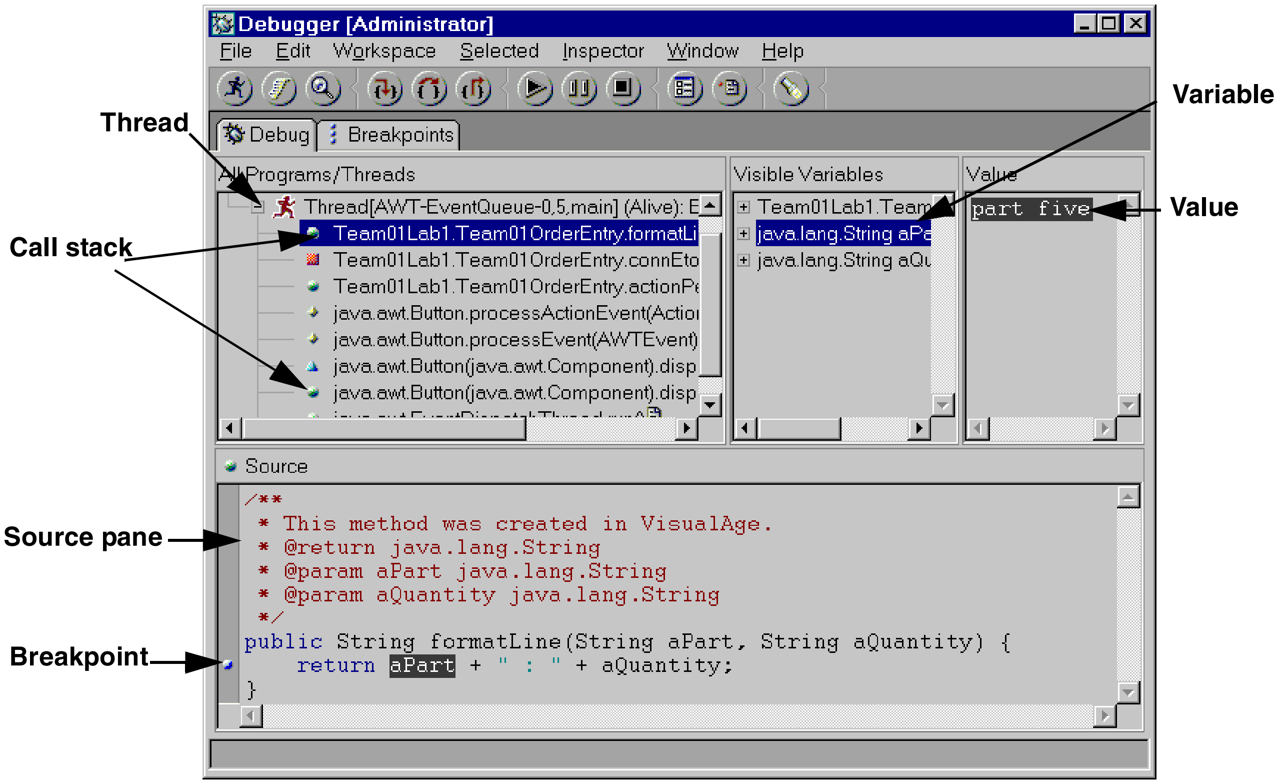


```
/**
 * This method was created in VisualAg
 * @return java.lang.String
 * @param aPart java.lang.String
 * @param aQuantity java.lang.String
 */
public String formatLine(String aPart,
    return aPart + " : " + aQuantity;
}
```

Figure 60. Breakpoint Example

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

The debugger window appears. The code stops prior to executing the statement. In the three upper panes, the left-hand pane (All Programs/Threads) shows a tree view of all current threads when the debugger was invoked. The threads are expanded with their call stacks with the most recent method at the top. The center pane (Visible Variables) shows the variables that are accessible, and the right pane (Value) shows the value of the currently selected variable. The bottom pane (Source) shows the current line in the source code of the current method (see Figure 61).



```
Thread[AWT-EventQueue-0,5,main] (Alive): E
  Team01Lab1.Team01OrderEntry.formatLi
  Team01Lab1.Team01OrderEntry.connEto
  Team01Lab1.Team01OrderEntry.actionPe
  java.awt.Button.processActionEvent(Actio
  java.awt.Button.processEvent(AWTEvent)
  java.awt.Button(java.awt.Component).disp
  java.awt.Button(java.awt.Component).disp
```

```
Team01Lab1.Team
java.lang.String aPa
java.lang.String aQu
```

```
part five
```

```
/**
 * This method was created in VisualAge.
 * @return java.lang.String
 * @param aPart java.lang.String
 * @param aQuantity java.lang.String
 */
public String formatLine(String aPart, String aQuantity) {
    return aPart + " : " + aQuantity;
}
```

Figure 61. VisualAge for Java Debugger

To view a variable, single left-click on the java.lang.String aPart variable in the visible Variables pane.

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

Expand the Part variable you have selected with a single left-click on the plus (+). Then, expand the resulting char[] value entry. Select entry 0, then 1, then 2, and so on.

When examining the single characters inside the array, you may recognize all of the characters from your typed part appearing one after the other in the Value pane (see Figure 62).

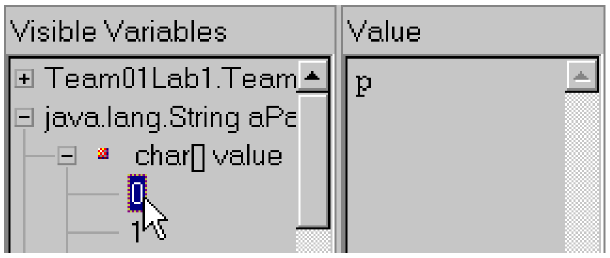
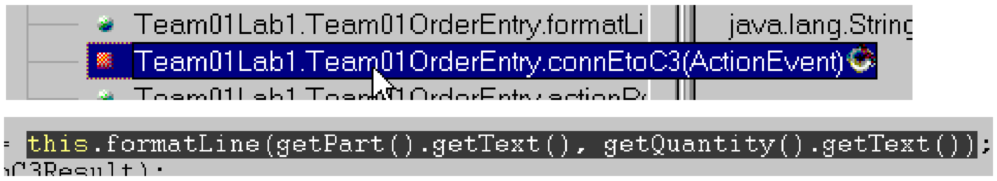


Figure 62. Examining Variables Example

Left-click on the next entry in the All Programs/Threads pane under the Team01Lab1.Team01OrderEntry.formatLine(String, String) entry.

This should be one of the type connEtoCx(ActionEvent). Notice that the Visible Variables, Value, and Source pane are all updated. In the Source pane, the actual code that called the current method is highlighted (see Figure 63).


Team01Lab1.Team01OrderEntry.formatLi
java.lang.String
Team01Lab1.Team01OrderEntry.connEtoC3(ActionEvent)

```
this.formatLine(getPart().getText(), getQuantity().getText());
```

Figure 63. Examining Call Stack Example

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

To change the code, perform these steps:

1. Reselect the top entry in the All Programs/Thread pane (not the thread above), **Team01Lab1.Team01OrderEntry.formatLine(String, String)**.
2. Modify the code so that the string " : " now reads " :- ". Save the method.
3. Left-click on the **Breakpoints** notebook tab to switch to the Breakpoints view.
4. Select the **Methods—>Clear** menu item from the menu bar to remove the breakpoint.

5. Switch back to the **Debug** view and left-click on the **Resume** smarticon in the tool bar of the debugger window. The debugger window blanks out since that thread has now run to completion.
6. Close the debugger window and navigate back to the running test window.

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

To test the Delete button, complete these steps:

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

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

2.3.4.11 Closing the Application and Versioning

To version the application, follow this sequence:

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

You should now have a working knowledge of the VisualAge for Java Integrated Development Environment. You can now follow the AS/400 client/server programming examples in Appendix A, “Example Programs” on page 395.

2.3.5 Team Development

Team development is enabled in VisualAge for Java with the incorporation of the ENVY/Developer from OTI, an IBM Subsidiary company. Team development is available as part of VisualAge for Java Enterprise Edition.

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

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

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

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

Just as in the Professional Edition, the Enterprise Edition records all changes made to any component and who made that change. In the Enterprise Edition, there are facilities to enable the access control rights for individual developers to every component within the repository.

Therefore, because of the ease of development with fallback facilities, the development in a Rapid Application Development (RAD) type environment is positively encouraged by the tool, with all the management controls should they be necessary.

The configuration of VisualAge for Java places a development image/workspace on the client and a repository on the client/file server in the Professional Edition. In the Enterprise Edition, a shared repository has to be placed on a shared file server. The repository holds a copy of every version of every component for the development team, where the image/workspace contains only the requested version of a sub-set of components. For example, Developer1 may work on GUI projects, packages, and classes, where Developer2 may be work on AS/400 access projects, packages, and classes. The shared repository (the Enterprise Edition) holds every edition and version of all these components. For example, Developer2 image/workspace holds only the AS/400 access components, not the GUI components.

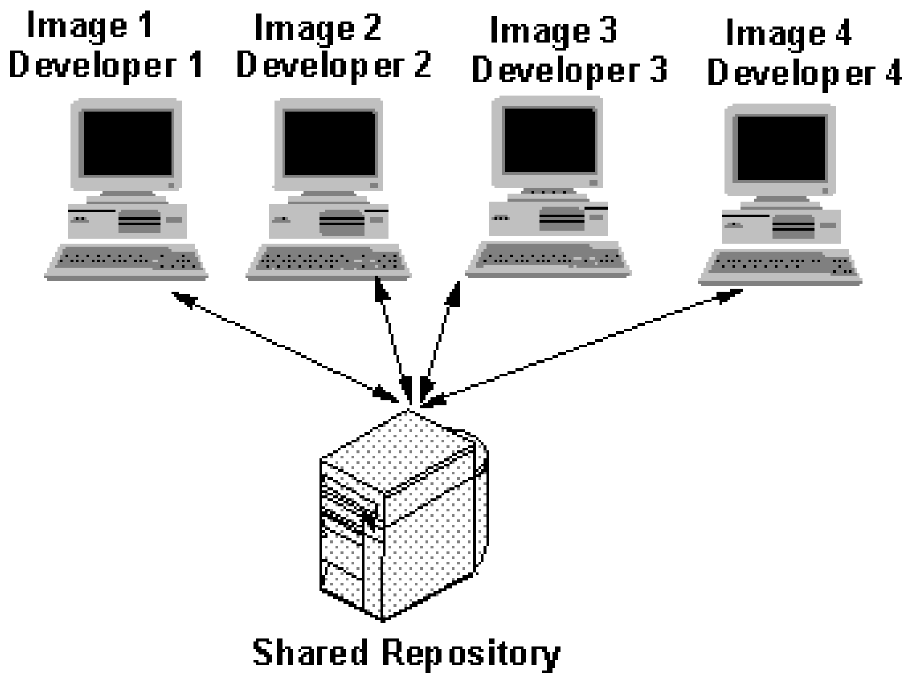


Figure 64. Team Development Configuration

In a team development environment (Figure 64), using the AS/400 IFS as the file server for the repository code, changes made by a developer to any component are written back immediately to the repository. Therefore, the component change is immediately made available to all other developers who may be using the component. On a nightly basis as part of the regular systems management procedures, the repository should be backed up to external media.

When a developer starts VisualAge for Java, the image/workspace is copied from disk into memory. The developer works with this copy of the image when adding, deleting, or changing components. It is vital that the developer saves this "in-memory" image to disk on a regular basis (for example, once per hour). It is not catastrophic if the developer receives a GPF after an entire series of changes since every component is still available in the repository. However, rebuilding the image from scratch may be time consuming.

In addition, at regular intervals (for example, at lunch time and at the end of the day), each individual developer should copy their working image/workspace to the AS/400 system. These, again, should be backed up on a nightly basis.

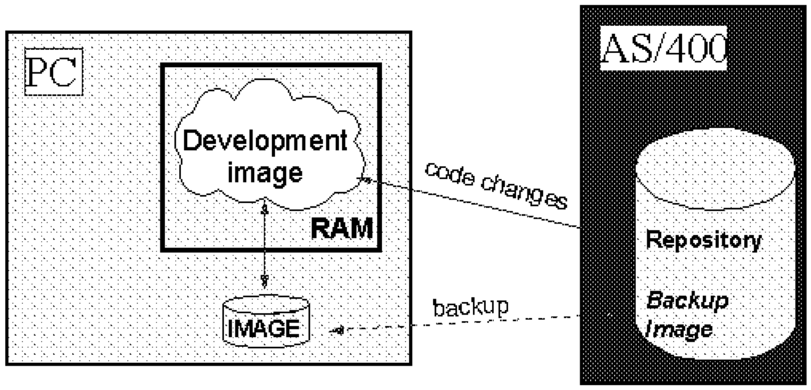


Figure 65. Team Development Backup Procedures

The team development facilities enable editing and creating versions of components. This is a simple process where the developer can create a version of a component at any time where a version is a frozen component that cannot be changed. Figure 66 shows three separate versions of the component. The developer can assign each version a unique name. In the example, the versions are 1.0, 1.1, and 2.0. As with most things in VisualAge for Java, a component can be any class/interface, package, or project. The developer explicitly versions these components. Methods are the exception because every change to them that is saved causes the creation of an new edition of the method.

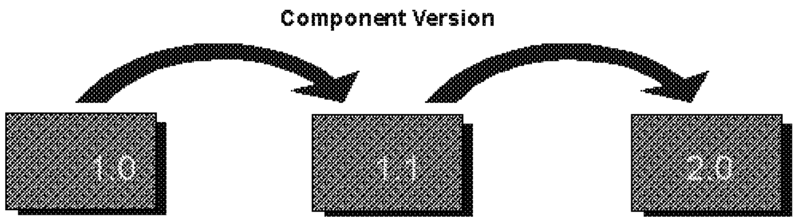


Figure 66. Team Development Versions

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

1. Create Class A (it gets created as an edition):
   - Write methods.
   - Define variables.
2. Version Class A as Class A 1.0:

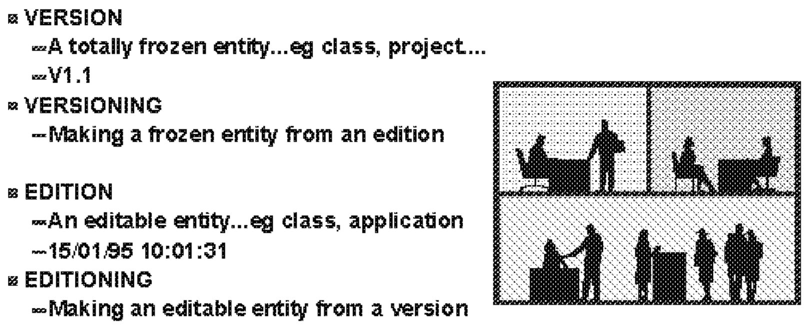
   Class A is frozen and cannot be changed.

3. Edition Class A:

   Class A can now be edited again, but version 1.0 is still available should it need to be restored.

4. Version Class A as Class A 2.0

See Figure 67.



- VERSION
  - A totally frozen entity...eg class, project...
  - V1.1
- VERSIONING
  - Making a frozen entity from an edition
- EDITION
  - An editable entity...eg class, application
  - 15/01/95 10:01:31
- EDITIONING
  - Making an editable entity from a version

Figure 67. Team Development Process

For more detailed information on team development related issues, please refer to the redbook *VisualAge for Java Enterprise Team*, SG24-5245.

2.3.6 Applets and Applet Viewer

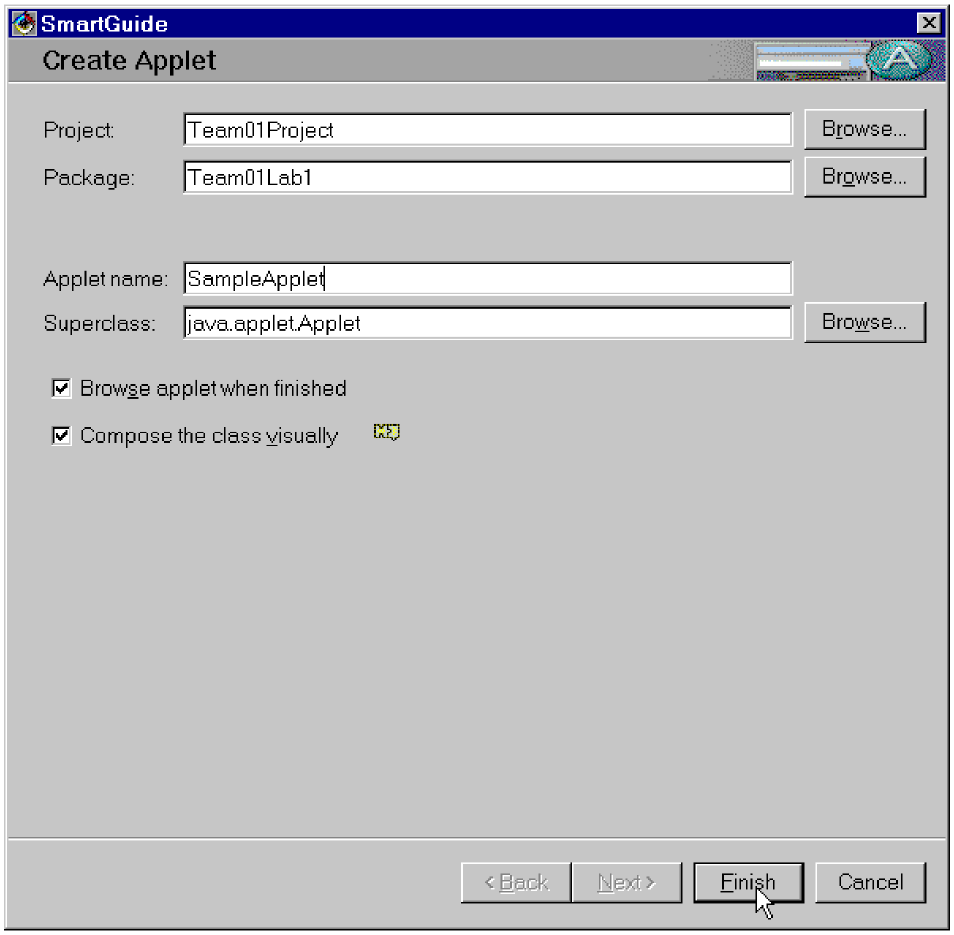
The VisualAge for Java Applet Viewer is incorporated into the IDE. This enables a developer to develop Java applets and to test them without bringing up a separate Web browser (for example, Netscape). The Applet Viewer is a primitive viewer and should only be used for debugging purposes with the final testing being performed in a real-life Web browser. However, because the Applet Viewer comes with VisualAge for Java, it supports the level of the JDK supported by the IDE (currently JDK 1.1.6). You may not be certain of this level of support in some Web browsers. For example, the current level of Netscape supports most, but not all, JDK 1.1.6 APIs.

VisualAge for Java has an applet creation SmartGuide that is accessed through its Create Applet smarticon on the tool bar (See Figure 68 on page 71). The applet creation SmartGuide walks the developer through the process of creating an applet and completing the tasks that usually are hand-coded into the applet. One of the windows that is displayed as part of the SmartGuide is included here as an example of the type of information the applet creation SmartGuide can process. The SmartGuide — Applet Properties window allows the setting of applet/application and thread details. Many applets can be run as stand-alone applets and stand-alone applications. In the latter case, a main() method needs to be created. In addition, should the applet perform a long running task or repeatable task (such as repeating animation), we advise that you write this as a separate thread. Again, the SmartGuide provides the option of creating the applet to use its own thread.

To create an applet, perform these steps:

1. Select the package **Team01Lab1** from project **Team01Project** in the workbench window.
2. Click the **Create Applet** smarticon on the tool bar.

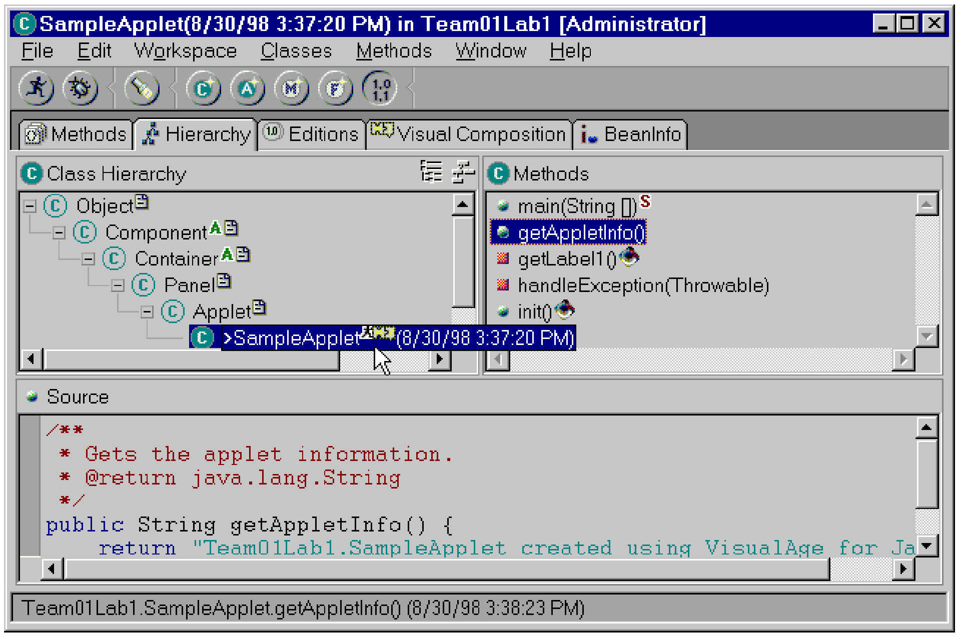
3. Enter **SampleApplet** for the Applet name and press the **Finish** button.
4. In the VCE, add a label in the design area.
5. Change the text property of the label to "Hello World," and adjust its width.
6. Save the Applet and switch to the **Hierarchy** view of the type browser.



Project: Team01Project
Package: Team01Lab1
Applet name: SampleApplet
Superclass: java.applet.Applet
☑ Browse applet when finished
☑ Compose the class visually

Figure 68. SmartGuide — Create Applet

After the applet is saved, you can see its place in the class hierarchy in the type browsers Hierarchy view. As you expect, the applet inherits its required methods from java.applet.Applet (see Figure 69 on page 72).


SampleApplet(8/30/98 3:37:20 PM) in Team01Lab1 [Administrator]
File Edit Workspace Classes Methods Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Class Hierarchy
Object
Component
Container
Panel
Applet
>SampleApplet (8/30/98 3:37:20 PM)
Methods
main(String [])
getAppletInfo()
getLabel1()
handleException(Throwable)
init()
Source

```
/**
 * Gets the applet information.
 * @return java.lang.String
 */
public String getAppletInfo() {
    return "Team01Lab1.SampleApplet created using VisualAge for Ja
```

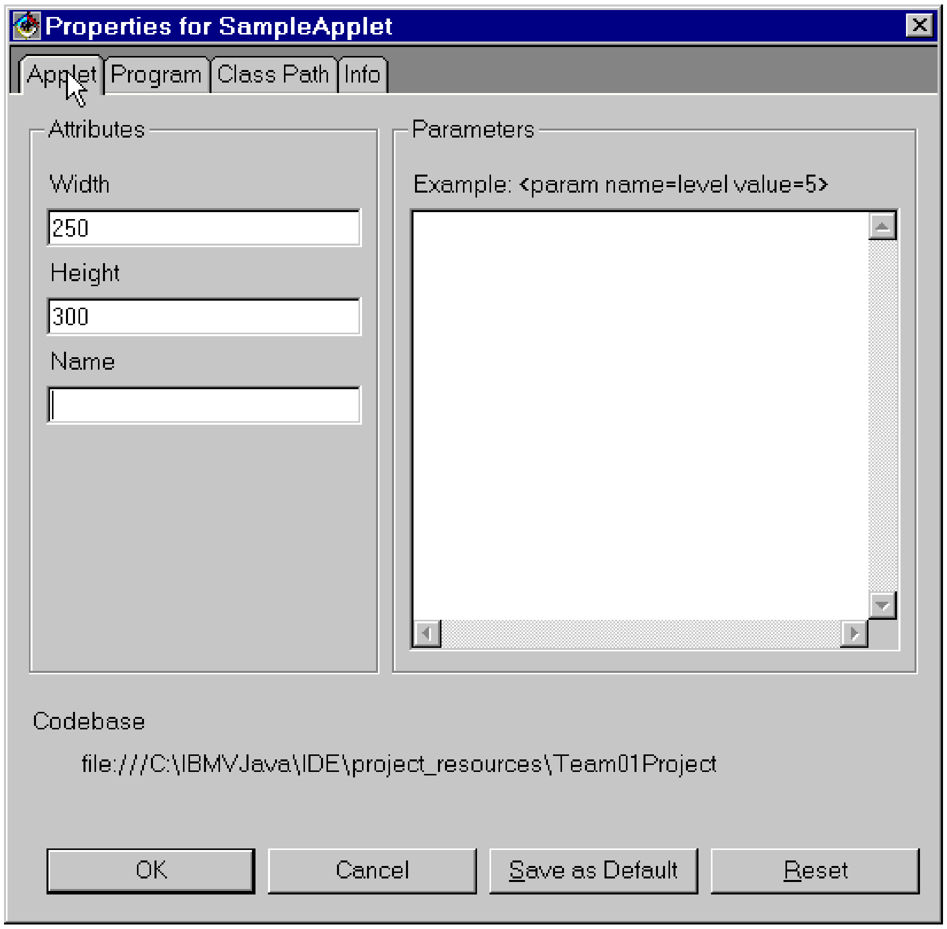
Team01Lab1.SampleApplet.getAppletInfo() (8/30/98 3:38:23 PM)

Figure 69. Sample Applet

To run the applet, follow these steps:

1. Select **Run—>Check Class Path...** from the applets pop-up menu.
2. Switch to the Applet view of the property browser by pressing the **Applet** notebook tab.

Outside of the IDE, an HTML file is required to wrap the applet so it can run in a Web browser. The HTML file specifies the width, height, parameters, and so on of the applet. Within the VisualAge for Java IDE, this HTML file is not required since the settings are automatically made for you in the applets properties. You can still change the values proposed by the IDE on the Applet view of the property browser. Note that this window also provides an interface to add or change command line arguments, properties, and classpath information (see Figure 70 on page 73).


Properties for SampleApplet

Applet | Program | Class Path | Info

Attributes

Width: 250

Height: 300

Name:

Parameters

Example: <param name=level value=5>

Codebase

file:///C:\IBMVJava\IDE\project_resources\Team01Project

Figure 70. Applet Properties Example

2.3.7 Editor, Debugger, and SmartGuides

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

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

- **Add or change a method**

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

- **Evaluate an expression**

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

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

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

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

- **Invoke methods**

  As previously discussed, most code can be evaluated "on the spot" without running an application. It follows from this that most methods can also be evaluated/invoked "on the spot".

- **Test, debug, set breakpoints**

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

- **Patch code**

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

- **Compile class/method incrementally**

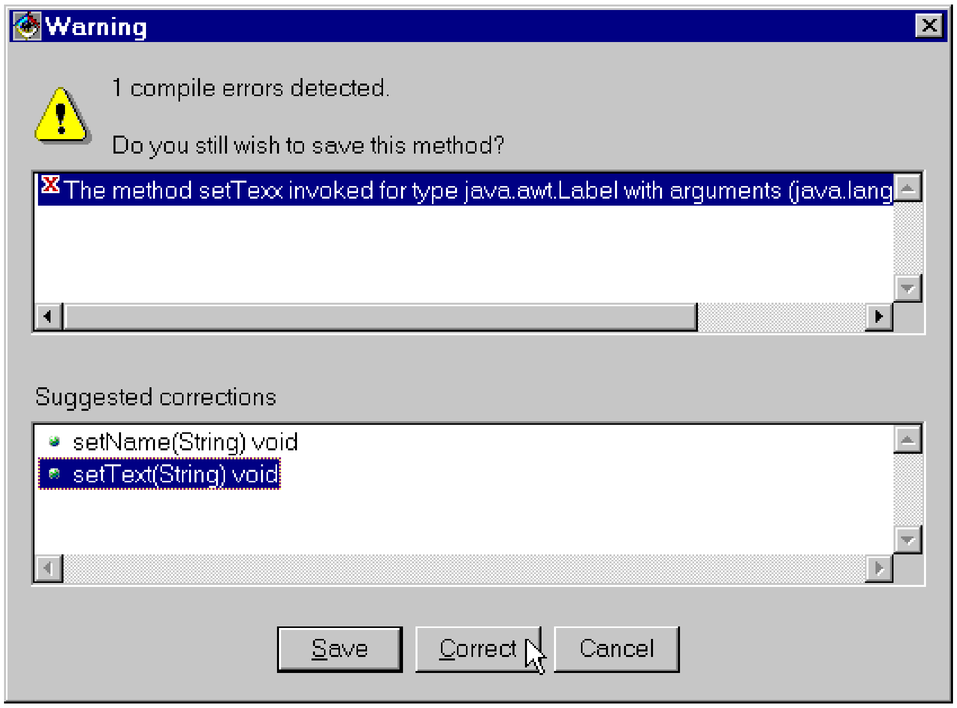
  Outside of the IDE, a developer must modify the class as a complete unit. Therefore, if only one line of a method needs modifying, then the entire .JAVA file needs to be edited and compiled. Within the VisualAge for Java IDE, individual methods can be edited and saved incrementally without the need to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

- **Maintain project database**

  The team development environment has already been introduced in this chapter. This team development environment provides a complete project database for the development team.

- **Syntax check code**

  VisualAge for Java detects syntax errors that occur when code violates Java syntax rules. For example, if you misspell a keyword or forget a semicolon, a message dialog box informs you of the type of syntax error when you try to save the code. In addition, the input cursor in the Source pane automatically selects the piece of code that caused the problem. Better than that, it even makes a proposal for the correction by giving you a list of known names to choose from if there are any available (see Figure 71 on page 75).


Warning

1 compile errors detected.

Do you still wish to save this method?

The method setTexx invoked for type java.awt.Label with arguments (java.lang

Suggested corrections

setName(String) void

setText(String) void

Figure 71. Syntax Error Suggested Corrections Example

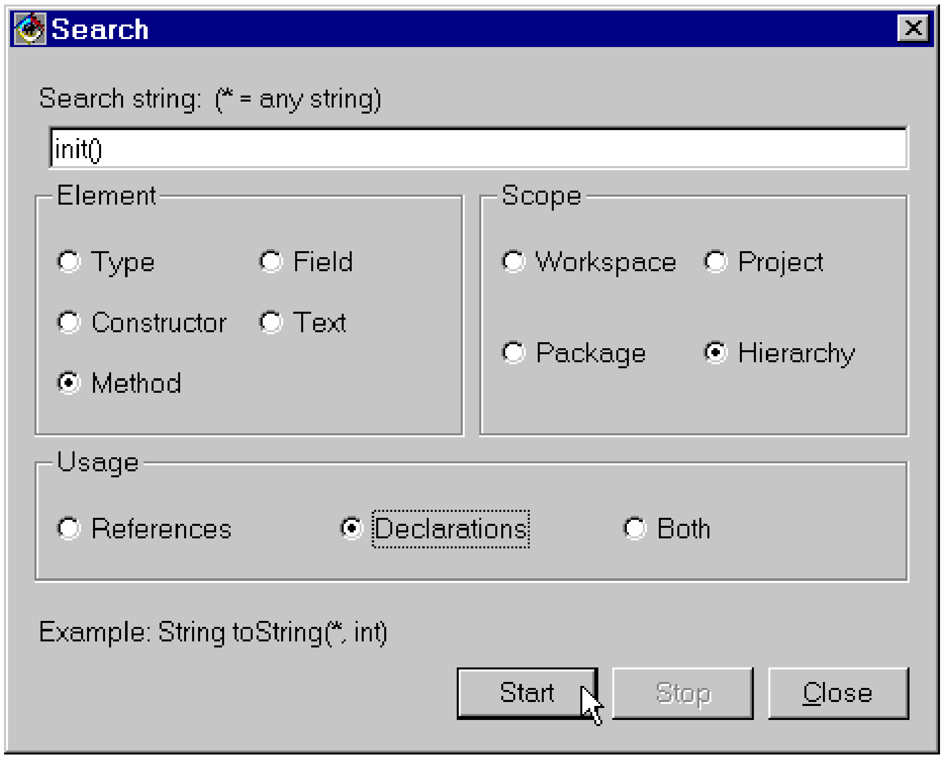
2.3.7.1 The Editor Pane

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

- Perform editing operations.
- Undo/Redo:

  This option is accessed from the Edit menu item.
- Search in the workspace (image) for highlighted text:

  A developer can highlight some text and select Search from the pop-up menu to search the workspace, project, package, or hierarchy for both references to or declarations of the highlighted text. The search string can be declared as Type, Field, Constructor, Text, or Method (see Figure 72 on page 76).



Search string: (* = any string)

init()

Element: Type, Field, Constructor, Text, [✓ Method]

Scope: Workspace, Project, Package, [✓ Hierarchy]

Usage: References, [✓ Declarations], Both

Example: String toString(*, int)

Figure 72. Search Example

- Insert and remove breakpoints for debugging:

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

- Save your changes:

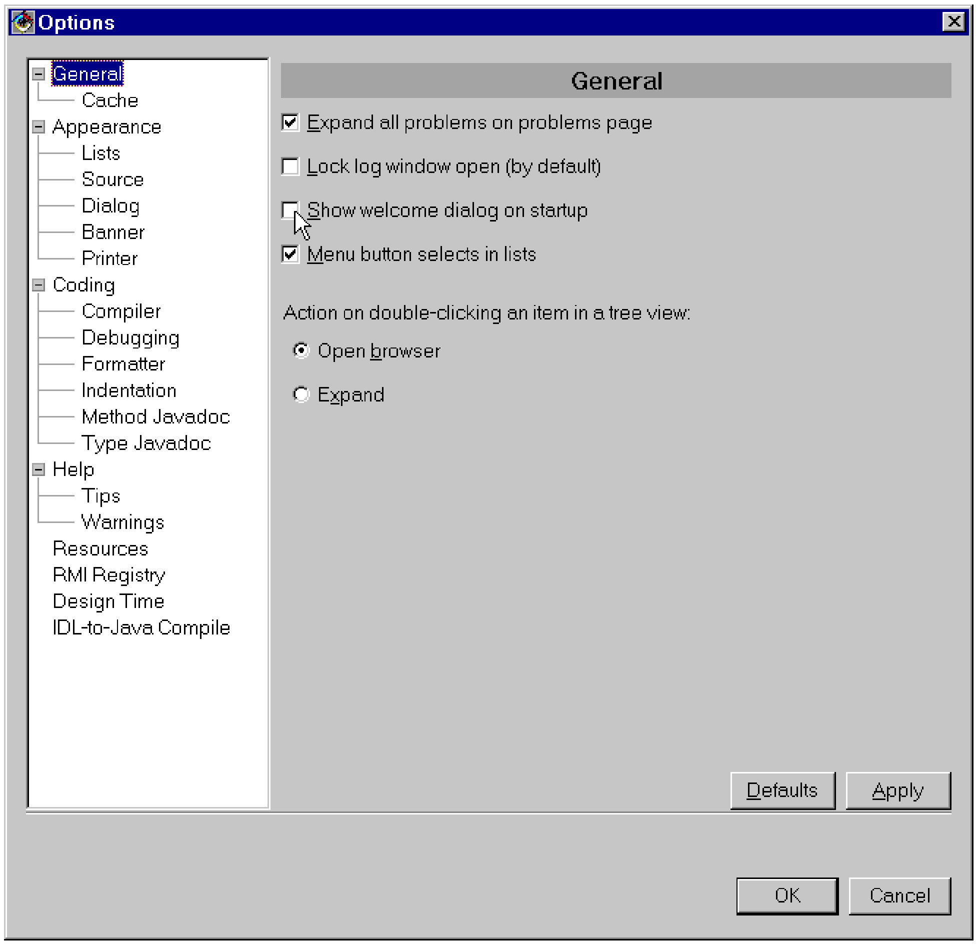
  When changes are saved for a method, the entire method is syntax checked before it is saved. At any time, the previous version can be restored.

- Cancel your changes:

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

- IDE setup:

  The IDE has some default settings and these can be modified by selecting the **Window—>Options...** from the workbench window (see Figure 73 on page 77).



General

- [✓] Expand all problems on problems page
- [ ] Lock log window open (by default)
- [ ] Show welcome dialog on startup
- [✓] Menu button selects in lists

Action on double-clicking an item in a tree view:

● Open browser

○ Expand

Figure 73. VisualAge for Java Options Window

2.3.7.2 The Debugger

In addition to the features mentioned in Section 2.3.4.10, “Debugging, Setting Breakpoints, and Changes ‘On the Fly’” on page 63, we discuss other interesting aspects of the debugger in VisualAge for Java. As you work in the integrated development environment, you do not need to launch a special debugger virtual machine or start the virtual machine in the debug mode. The debugger opens automatically when you need it. It opens when:

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

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

VisualAge activates the debugger when one of a program's threads encounters a breakpoint. The top left pane (All Programs/Threads) displays the current thread that was created when you started the applet/application and the debugger invoked for whatever reason. In VisualAge, you create a thread (or multiple threads) whenever you run a program or evaluate code in the Scrapbook. When the debugger opens on a breakpoint, the threads pane displays the thread that caused the debugger to open. The entry consists of an internal identifier for the thread and an indication of what caused the debugger to open. Displayed under the thread is its program stack from which each entry corresponds to a method that was called. Program stacks are in reverse chronological order (the most recent method is the top entry). The debugger lets you manipulate thread execution by dropping to a particular method. This is particularly useful if the debugger opens on an uncaught exception, since it lets you back up and repeat the steps that caused the exception to be thrown (see Figure 61 on page 64).

Stepping through Methods

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

- **Step Into**

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

- **Step Over**

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

- **Run to Return**

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

- **Resume**

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

- **Suspend**

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

- **Terminate**

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

2.3.7.3 Inspectors

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

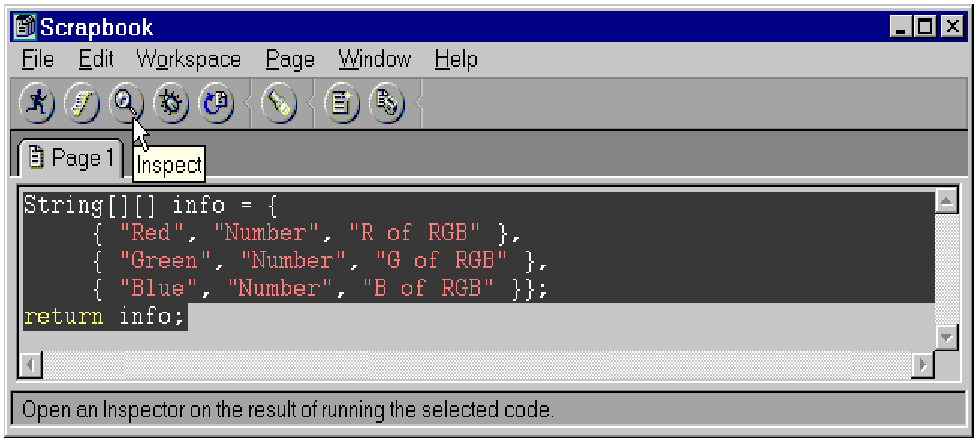
- Inspect the result of evaluating a code fragment in the Scrapbook or in the Variables pane of the debugger.
- Open a browser on the declarations of an object's class.
- Evaluate code fragments in the context of an object.
- Change the value of an object.

Perform the following example (for the Scrapbook window refer to Section 2.3.7.4, “Other VisualAge for Java Windows” on page 80):

1. Open a Scrapbook page by selecting **Window—>Scrapbook** from the Workbench Window. Type the following code into that page:

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

2. Now select all of the code (Ctrl+A), and left-click on the **Inspect smarticon** on the tool bar. See Figure 74.



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

Open an Inspector on the result of running the selected code.

Figure 74. Scrapbook Example

The inspector appears and shows the array object stored in the info variable. The title bar displays the identifier for the class of the inspected object, which is a two-dimensional string array. The title bar also shows the context from which you opened the inspector (from Page 1).

The Fields pane shows the elements of the array. The Value pane shows the value of a selected field.

The info array maps to a table with three rows and three columns (indexed 0 through 2). The top-level items in the Fields pane map to the three rows. By expanding items 0 through 2, you see that each row consists of three columns. Select the second row in the first column (info[1][.0]).

It holds the parameter name Green. Internally, the string Green is represented as an array of characters that you can view in more detail by expanding the tree in the Fields pane. The icon to the left of the character array indicates that the internal representation is private (see Figure 75).

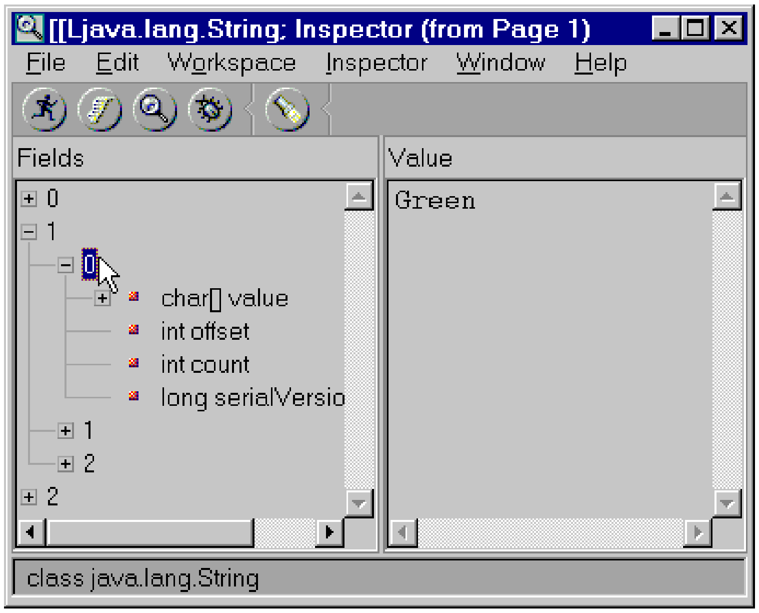


Figure 75. Inspector Example

You can change the value of fields while you are inspecting an object by following these steps:

1. In the Fields pane, select the field that you want to modify.
2. In the Value pane, replace the text with the value that you want in the field.
3. Select **Save** from the pop-up menu.

The expression in the Value pane is evaluated. If the result can be assigned to the object, it is. When the code resumes, it uses the value. If the result cannot be assigned, the inspector displays an error message.

2.3.7.4 Other VisualAge for Java Windows

This section describes other windows available for VisualAge for Java.

The Scrapbook

The Scrapbook helps you organize code fragments and notes. You can run any Java statement or expression from the scrapbook and control the context in which it is compiled.

To open the scrapbook, select **Scrapbook** from any window pull-down menu. The scrapbook appears with an empty page. From the scrapbook, you can run the code fragment or open an inspector on the object that is returned as the result of running the code. To open an inspector, select **Inspect** from the pop-up menu of the selected code fragment (see Figure 74 on page 79).

For example, most programming languages and environments take developers through the "Hello World" application as the first exercise in learning a new language or environment. With VisualAge for Java, this can be achieved in under a minute.

Hello World in Under a Minute

Complete the following steps:

1. Select **Window—>Scrapbook** and type this statement:

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

2. Select the line of code that you typed.
3. Press the run smarticon.

The console (the standard output device) appears and displays the string Hello World!. The code is automatically compiled by the built-in Java compiler and run by the built-in Java virtual machine.

The Console

The console is the standard output device (System.out) for Java programs that you run in VisualAge.

The Repository Explorer

With the Repository Explorer, you can explore the repository to view program components that are not present in the workspace/image.

The Log

The log displays messages and warnings from VisualAge.

2.3.7.5 SmartGuides/Wizards

The VisualAge for Java IDE comes with various SmartGuides (also known as Wizards in other IDEs) that guide the developer through the repeatable process of creating a component.

For example, the Class Creation SmartGuide takes the developer through the standard process of creating a class including the following setup parameters:

- Which project is the class defined in?
- Which package is the class defined in?
- What is the class name?
- Which class is the superclass?
- What happens when the SmartGuide completes?
  - Open a VCE (for example, if the class inherits from java.awt.Frame).
  - Open a class browser.
  - Do not open a browser.
  - Which interfaces (if any) does the class implement?
  - Which modifiers should be implemented?
    - Public
    - Abstract
    - Final
  - Should stub methods be generated?

There are a number of SmartGuides including class creation, interface creation, method creation, and applet creation in the Professional Edition.

In VisualAge for Java Version 2.0 Enterprise Edition, you can find many different SmartGuides, such as the San Francisco Wizard, SmartGuides for Data Access, C++ Access, RMI and so on.

2.3.7.6 VisualAge for Java Help

The VisualAge for Java IDE comes with extensive documentation consisting of numerous HTML-Files that are mostly stored locally but can also contain links to pages on the Internet. Therefore, it is advantageous to have access to the Internet while searching for information. Help for VisualAge for Java can be accessed by selecting a menu item from the Help menu. It starts your default browser and displays the page that you chose. First, read the Help on Help section to become familiar with how to use Help. Then, navigate through the various topics on the left window to gain an idea of what can be found where inside the documentation of VisualAge for Java. Try also the VisualAge for Java Search, which works same as any search in the Internet.

2.4 Enterprise Access Builders (EAB)

The VisualAge for Java Enterprise edition includes the following Access Builder components:

- **Enterprise Access Builder for Data**

  This component allows access to any relational database that supports either an ODBC driver or a JDBC driver.

- **Enterprise Access Builder for Java to C++**

  This component allows access to C++ services by generating JavaBeans and C++ code to allow interoperability between Java and C++.

- **Enterprise Access Builder for RMI**

  This component is used for creating distributed Java applications. RMI allows a Java object running on one virtual machine to send messages to another Java object running on another Java virtual machine. These objects can even be on different systems.

- **Enterprise Access Builder for SAP R/3 with SAP R/3 BAPI business objects**
- **Enterprise Access Builder for Persistence Enterprise Access Builder**

  This component is used for transforming relational schemas into Enterprise JavaBeans components.

- **Enterprise Access Builder for interacting with existing applications**

Note: To use any of the Enterprise Access Builders, the corresponding features have to be added to the workspace.

These subcomponents produce JavaBeans for access to transactions and databases. In this redbook, we focus on the Data Access Builder.

2.4.1 Data Access Builder (DAX)

VisualAge for Java — Enterprise Access Builder for Data (referred to as Data Access Builder (DAX)) is an application development tool that you can use to create data access classes customized for your existing relational database tables. It allows you to create object-oriented applications quickly and reliably by generating the source code for you. These data access classes, which are JavaBeans, can be used directly in your Java programs and by the VisualAge for Java IDE.

Some of the key features of the Data Access Builder are:

- *JDBC to access your database*

  Data Access Builder generates code that uses JDBC to access your database. You can use the JDBC driver in IBM DB/2, JDBC-ODBC bridge in JDK Version 1.1, or other JDBC drivers with the generated code.

- *Flexibility in specifying source*

  Data Access Builder generates code from database tables, from database views, or from SQL statements that you type.

- *Quick and simple to use*

  You can simply specify a database table name. Data Access Builder can access the table information and generate Java source code that allows you to add, update, delete, or retrieve the data in that table.

- *Data manipulation operations*

  Generated classes customized to your data help you perform common database tasks such as adding, retrieving, updating, and deleting data. Classes are also generated to allow you to use a cursor to fetch rows from database queries that return result sets.

- *Add your own methods*

  You can add your own methods by typing in SQL statements. Data Access Builder generates the Java source code for you.

- *Stored procedure support*

  You can use Data Access Builder to generate code that calls stored procedures.

- *Generate code for table joins*

  You can specify table joins using SQL statements, and Data Access Builder can generate Java classes for them.

- *Connection and transaction services*

  Separate services are provided for connection and disconnection from your databases. In addition, commit and rollback methods are generated to handle transaction services.

For more detailed information on DAX and examples of how to use it to build Java applications and applets that access the AS/400 system, refer to Chapter 6, “Enterprise Access Builder for Data (DAX)” on page 267.

2.5 System Requirements and Prerequisites for Version 2.0

Table 2 defines the system requirements for VisualAge for Java depending on which version you choose.

Table 2. System Requirements

Requirement/Version	**Professional**	**Enterprise**
Hardware Requirements	Intel Pentium or higher compatible processor	Intel Pentium or higher compatible processor
	48MB of RAM (64MB recommended to accommodate UNICODE support)	64MB of RAM (80MB recommended to accommodate UNICODE support)
Hardware Requirements	Hard disk space: 190MB minimum, 250MB or more recommended	Hard disk space: 250MB minimum, 300MB or more recommended
	CD-ROM drive	CD-ROM drive
	Mouse or pointing device	Mouse or pointing device
	Display: SVGA, 800x600 (1024x768 recommended)	Display: SVGA, 800x600 (1024x768 recommended)
Software Requirements	Windows 95/98 OR Windows NT 4.0 service pack 3 OR OS/2 Warp 4.0	Windows 95/98 OR Windows NT 4.0 service pack 3 OR OS/2 Warp 4.0
Software Prerequisites	A frames capable browser to access the HTML-based help and Web documentation such as: • Netscape Navigator Version 4.04 or later • Microsoft Internet Explorer Version 4.01 or later	A frames capable browser to access the HTML-based help and Web documentation such as: • Netscape Navigator Version 4.04 or later • Microsoft Internet Explorer Version 4.01 or later
	TCP/IP communications protocol configured and running	TCP/IP communications protocol configured and running
		The DB2 Universal Database adapters are included. To use the Enterprise Access Builder for Data with other relational databases, the JDBC device driver at JDK 1.1 level or ODBC driver is required.

Requirement/Version	Professional	Enterprise
Supported Languages (Levels)	Java Developers Kit Version 1.1 Java-enabled browsers with JDK 1.1 support **Note:** Must be JDK 1.1.2 for Swing-enabled apps	Java Developers Kit Version 1.1 Java-enabled browsers with JDK 1.1 support **Note:** Must be JDK 1.1.2 for Swing-enabled apps

2.6 Migration from VisualAge for Java Version 1.0 to 2.0

Before migrating from VisualAge for Java, Version 1.0 to the new Version 2.0, it is absolutely necessary that you version all projects, packages, and types that you want to migrate. It is advantageous to install the new product on a different computer and keep the old version running until the migration process is completed successfully. Once you finish the installation, you can import versioned projects and packages from your older repository files. Refer to the on-line help of the IDE, for instructions on importing from another repository. Note that there are some differences between migrating from a stand-alone to a stand-alone, and stand-alone to a team development environment. For detailed information about installing and migrating, refer to the readme.txt file on the CD-ROM containing VisualAge for Java Version 2.0.

For an overview of the migration process, see the following phases:

1. Making a decision about which Edition (Professional/Enterprise) and whether a stand-alone or team environment should be used
2. Making a Version of all components that are to be migrated
3. Saving the old repositories
4. De-installing the old and installing the new product
5. Importing all the components to be migrated from the old repository
6. Checking and testing in the new environment

When projects and packages from the old repository are imported, add them into your new workspace and find out if any problems are reported. Check the All Problems view in your Workbench window. See if there are any unresolved problems marked with a red X (see Figure 26 on page 29). Add any necessary additional features into your Workspace, for example IBM Enterprise Toolkit for AS/400 if the AS/400 Toolbox for Java was used in any of the old projects. Any component that was used in any of the types in the old repository must be present in the new work space. Otherwise, the IDE may report errors. This applies also to your own JavaBeans, third-party class-libraries, and imported classes. There may be some problems that can only be solved by changing the types and methods affected.

Note

The old package COM.ibm.ivj.javabeans has been replaced by com.ibm.ivj.eab.dab, which is now in the project named IBM Enterprise Data Access Libraries. For example, if you have used the IMulticolumnListbox bean or the IMessageBox bean in one of your projects, you need to replace the package names of the classes. The procedure is the same when opening a visually designed class with the VCE. You are prompted by a window showing the possible classes from the new package to replace the old ones.

Before testing your applications in the new workspace, eliminate all of the reported problems and make sure that the classpath shown in the Class Path view of the properties window (see Figure 70 on page 73) is correct. If this is not the case, correct the classpath manually or remove the classpath information and have it regenerated automatically by pressing the Compute Now button.

If your applications were based on JDK 1.1.5 or earlier, there may be some warnings reported by a yellow signal for deprecated methods. Since your applications should work correctly, you do not need to correct the methods and types affected by these warnings. If you want to make all unresolved problems disappear, replace the deprecated methods with the corresponding new methods. Normally such a warning has the remark: `method X with argument(Y) is deprecated`. This means that such a method is not to be used in the future. To find the implementation of the deprecated method, start from the class where you have the warning, up the hierarchy, until you find the implementation of that method. One example is the method show(boolean). In this method, you find a comment such as the one that appears in the following example:

```
/**
* @deprecated As of JDK version 1.1,
* replaced by <code>setVisible(boolean)</code>.
*/
```

You can replace all occurrences of the deprecated method with the new method indicated in the comment. After saving the changes, the reported warning disappears. An effective way to handle multiple changes of the same method is to use the Find/Replace menu item from the Edit menu.

The following list shows the deprecated methods that had to be replaced when migrating the examples in this redbook from VisualAge for Java Version 1.01 to Version 2.0:

```
deprecated: java.awt.Component.show()
   replaced by: java.awt.Component.setVisible(true)
deprecated: java.awt.Component.hide()
    replaced by: java.awt.Component.setVisible(false)
deprecated: java.awt.Component.show(boolean)
    replaced by: java.awt.Component.setVisible(boolean)
deprecated: java.awt.Component.layout()
   replaced by: java.awt.Component.doLayout()
deprecated: java.awt.Component.size()
   replaced by: java.awt.Component.getSize()
deprecated: java.awt.Component.reshape(int int int int)
   replaced by: java.awt.Component.setBounds(int int int int)
deprecated: java.awt.Component.bounds()
   replaced by: java.awt.Component.getBounds()
deprecated: java.awt.Component.resize(int int)
   replaced by: java.awt.Component.setSize(int int)
deprecated: java.awt.Component.resize(java.awt.Dimension)
   replaced by: java.awt.Component.setSize(java.awt.Dimension)
```

```
deprecated: java.awt.Component.locate(int int)
   replaced by: java.awt.Component.getComponentAt(int int)
deprecated: java.awt.Component.location()
   replaced by: java.awt.Component.getLocation()
deprecated: java.awt.Component.enable()
   replaced by: java.awt.Component.setEnabled(true)
deprecated: java.awt.Component.disable()
   replaced by: java.awt.Component.setEnabled(false)
deprecated: java.awt.Component.preferredSize()
   replaced by: java.awt.Component.getPreferredSize()
deprecated: java.awt.Component.move(int int)
   replaced by: java.awt.Component.setLocation(int int)
deprecated: java.awt.ComponentPeer.show()
   replaced by: java.awt.ComponentPeer.setVisible(true)
deprecated: java.awt.ComponentPeer.hide()
   replaced by: java.awt.ComponentPeer.setVisible(false)
deprecated: java.awt.ComponentPeer.reshape(int int int int)
   replaced by: java.awt.ComponentPeer.setBounds(int int int int)
deprecated: java.awt.Container.layout()
   replaced by: java.awt.Container.doLayout()
deprecated: java.awt.Container.locate(int int)
   replaced by: java.awt.Container.getComponentAt(int int)

deprecated: java.awt.ScrollPane.layout()
   replaced by: java.awt.ScrollPane.doLayout()
deprecated: java.awt.Rectangle.resize(int int)
   replaced by: java.awt.Rectangle.setSize(int int)
deprecated: java.awt.Rectangle.move(int int)
   replaced by: java.awt.Rectangle.setLocation(int int)
deprecated: java.awt.TextField.setEchoCharacter(char)
   replaced by: java.awt.TextField.setEchoChar(char)
deprecated: java.awt.TextFieldPeer.setEchoCharacter(char)
   replaced by: java.awt.TextFieldPeer.setEchoChar(char)
deprecated: java.awt.Choice.countItems()
   replaced by: java.awt.Choice.getItemCount()
deprecated: java.awt.List.countItems()
   replaced by: java.awt.List.getItemCount()
deprecated: java.awt.Menu.countItems()
   replaced by: java.awt.Menu.getItemCount()
```

2.7 Upgrades Available for VisualAge for Java 2.0

Currently there are a number of upgrades available for VisualAge for Java 2.0. Some of these upgrades provide additional features, while others improve the core Java support. Table 3 highlights some of these upgrades.

Table 3. VisualAge for Java Updates

Update / Fix name	Applies to	Provides
Rollup 2	Professional and Enterprise Editions	JDK 1.1.7A Swing 1.0.3 Windows 98 Support Service fixes
Professional Edition Update	Professional	Integrated WebSphere test environment for building JSP and Servlets
Enterprise Edition Update	Enterprise	All the features of the Professional Edition Update. EJB support: including EJB wizards, EJB class management tools and a deployment tool.

For more information and to download these updates, please see the IBM VisualAge for Java home page at: http://www.software.ibm.com/ad/vajava

Select the link to the VisualAge for Java Developer Domain.

Both the Professional and Enterprise Edition Updates require more workstation resources. Be sure to review the documentation before downloading and installing these updates.

2.8 Summary

In summary, VisualAge for Java is a member of the VisualAge family. It allows application developers to develop applications and Web-based applets, servlets and Java Server Pages using the Java language.

VisualAge for Java includes a powerful and full-function integrated development environment. The IDE is JDK 1.1 compliant, allowing the edit/compile/test of Java applications within the IDE prior to exporting the code for running in other JDK 1.1 compliant virtual machines and Web browsers. Because of its compliance with the JDK 1.1 API, the VisualAge for Java environment supports Java APIs for accessing remote components through the RMI and JDBC APIs. In particular, with the AS/400 Toolbox, the AS/400 development environment is extremely rich. This enables access to the most common application development building blocks on the AS/400 system (files, data queues, programs, and so on).

Because of the portability of JDK 1.1 compliant Java code, code that is developed using VisualAge for Java can run without change on the native AS/400 Java Virtual Machine that is now available.

The IDE enables a developer to build and run applications and code snippets interactively without running the compile statement (javac) from the command line. All applications can run from within the IDE without the need to export the Java source or class files. This is achieved through the provision of a JDK 1.1 compliant Virtual Machine (VM) within the IDE. Because you can interactively modify code and run it without compilation, developers can debug code on the fly, spot errors in their code with the debugger, change it, and continue without bringing the running application down, all within the VisualAge for Java IDE.

VisualAge for Java is an open IDE. Developers can easily import and export Java source, class files, and JavaBeans, which may have been purchased by the company or made available on the Internet. The JavaBeans support in VisualAge for Java also enables a developer to import an existing JavaBean (for example, from the Internet) into VisualAge for Java, modify the bean, and export it again for use within another JDK 1.1 compliant development environment (for example, Symantic Cafe and Borland’s JBuilder).

VisualAge for Java has two components that extend its capabilities to make client/server programming easier. The Enterprise Access Builders (EAB) provide components to aid connection to DB2 compliant data sources, and other programs. The AS/400 Toolbox for Java provides a series of classes specifically designed to access many AS/400 features (all without using Client Access/400 as a prerequisite).

The product will run on OS/2 Warp Version 4.0, Windows NT 4.0, or Windows 95/98.

Chapter 3. AS/400 Toolbox for Java

This chapter covers the AS/400 Toolbox for Java. It includes the following topics:

- Introduction to the AS/400 Toolbox for Java
- Introduction to application examples
- JDBC performance tips
- JDBC example
- Reusable GUI part
- Stored procedures example
- Distributed Data Management Record Level Access example (DDM, RLA)
- Distributed Program Call Example (DPC)
- Data queues example
- Print example
- Integrated file system example

The source code for any of the examples discussed in this chapter is available on the Internet. For download instructions, please refer to Section A.1, “Downloading the Files from the Internet” on page 396.

3.1 Introduction to the AS/400 Toolbox for Java

The AS/400 Toolbox for Java is a library of Java classes that enables the Internet programming model. The classes can be used by Java applications, applets, servlets, and Java Server Pages to easily access AS/400 data and resources. The toolbox does not require additional client support beyond what is provided by the Java Virtual Machine and JDK.

The AS/400 Toolbox for Java is currently available from IBM with OS/400 V4R2 or later as a no charge licensed program product (LPP) 576x-JC1. It is a fully supported licensed program of the AS/400 system. Table 4 shows the version of OS/400, the version of the toolbox shipped, and the minimum level of OS/400 that is supported as a server. The toolbox classes can be used to communicate with this level of OS/400.

Table 4. AS/400 Toolbox for Java Versions

Shipped with OS/400	Toolbox Shipped	LPP Details	Min OS level supported
V4R2	Modification 0	5763JC1 V3R2M0	V3R2M0
V4R3	Modification 1	5763JC1 V3R2M1	V3R2M0
V4R4	Modification 2	5769JC1 V4R2	V4R2M0

Additional support information can be found at the AS/400 toolbox home page at: http://www.as400.ibm.com/toolbox

The toolbox provides support similar to functions available when using the Client Access/400 APIs. It uses socket connections to the existing OS/400 servers as the access mechanism for the AS/400 system. Each server runs in a separate job on the AS/400 system and sends and receives architected data streams on a socket connection.

The AS/400 Toolbox for Java is delivered as a Java package that works with existing servers to provide an Internet-enabled interface to access and update AS/400 data and resources.

The base API package contains a set of Java classes that represent AS/400 data and resources. The classes do not have an end-user interface. They simply move data back and forth between the client program and an AS/400 system under the control of the client Java program. The Java classes in the base API package have these functional responsibilities:

- Describe the public interface for access to AS/400 data and resources
- Manage a set of sockets connections to the server jobs
- Implement the public interface by creating and parsing the data streams defined for the appropriate server

Access to the following AS/400 data and resources is provided:

- AS400 object, infrastructure, and sign-on
- JDBC access to DB2/400 data
- Record-level access to DB2/400 data
- Integrated file system
- Print functions
- Commands
- Program calls
- Data queues

The following functions do not directly access AS/400 data and resources, but provide useful services for Java programmers accessing AS/400 data:

- AS/400 data types
- AS/400 data description
- Access to AS/400 messages generated from a command, program call, or print operation

3.1.1 Installing the Toolbox

If you want to use the AS/400 Toolbox for Java classes inside the VisualAge for Java Integrated Development Environment, you must import these classes inside the IDE. VisualAge for Java Enterprise Edition simplifies this process. After you install VisualAge for Java 2.0 Enterprise edition, the AS/400 Toolbox for Java classes are already available in the repository as part of the IBM Enterprise Toolkit for AS/400 project. If you want to use the Toolbox classes, follow these steps:

1. From the workbench, click on **File**. Then click on **Quick Start**.
2. Click on **Features**—>**Add Feature**. Then, click **OK**.
3. Select **IBM Enterprise Toolkit**, and click **OK**.

This adds the toolbox classes to your workspace. The IBM Enterprise Toolkit for AS/400 is listed under All projects.

The alternative is to use these steps:

1. Install LPP 5763-JC1 (5769-JC1 for V4R4) on an AS/400 system or access the Web site at: http://www.ibm.com/as400/toolbox
2. Download the classes to your workstation.
3. Import the classes into the VisualAge for Java IDE.

3.1.2 V4R3 Enhancements

With OS/400 V4R3, the AS/400 Toolbox for Java (5763JC1 V3R2M1) offers two significant enhancements. Several new access classes are provided, as are the graphical user interface (GUI) classes. New classes are provided to access the following AS/400 resources:

- **Digital certificates**

  Digital certificates are digitally signed statements used for secured transactions over the Internet. Digital certificates can be used on AS/400 systems running on Version 4 Release 3 and later. To make a secure connection using the secure sockets layer (SSL), a digital certificate is required.

- **Jobs**

  The AS/400 Toolbox for Java jobs classes allow a Java program to retrieve the attributes of a job and list the active jobs. The job classes are:

  – Job — Represents an AS/400 job object
  – JobList — Represents a list of AS/400 jobs
  – JobLog — Represents the job log of an AS/400 system

- **Message queues**

  The MessageQueue class allows a Java program to interact with an AS/400 message queue. It acts as a container for the QueuedMessage class. The getMessages() method, in particular, returns a list of QueuedMessage objects. The MessageQueue class can perform these tasks:

  – Set message queue attributes
  – Get information about a message queue
  – Receive messages from a message queue
  – Send messages to a message queue
  – Reply to messages

- **Queued messages**

  The QueuedMessage class extends the AS400Message class. The QueuedMessage class accesses information about a message on an AS/400 message queue. With this class, a Java program can retrieve:

  – Information about where a message originated, such as the program, job name, job number, and user
  – The message queue
  – The message key
  – The message reply status

- **Users and groups**

  The user and group classes allow a Java program to get lists of users and groups on the AS/400 system and information about each user. You use a UserList object to get a list of users and groups on the system. The only property of the UserList object that must be set is the AS400 object that represents the AS/400 system from which the list of users is to be retrieved.

- **User space**

  The UserSpace class represents a user space on the AS/400 system. Required parameters are the name of the user space and the AS400 object that represents the AS/400 system that has the user space. Methods exist in a user space class to perform the following functions:

  – Create a user space
  – Delete a user space
  – Read from a user space
  – Write to user space
  – Get the attributes of a user space (A Java program can get the initial value, length value, and automatic extendible attributes of a user space.)
  – Set the attributes of a user space (A Java program can set the initial value, length value, and automatic extendible attributes of a user space.)

With the GUI classes, you can visually represent your AS/400 data and resources. Please refer to Chapter 4, “AS/400 Toolbox for Java — GUI Classes” on page 181, for more information and examples of using the new GUI classes and the new access classes. You can also refer to redbook *Building AS/400 Internet Based Applications with Java*, SG24-5337, for more information and examples of using the new access classes.

3.1.3 V4R4 Enhancements

The AS/400 Toolbox for Java Modification 2 (5769JC1 V4R2) adds significant enhancements and new features to the AS/400 Toolbox for Java. The following sections briefly summarize the enhancements. For details, refer to Chapter 5, “AS/400 Toolbox for Java Modification 2” on page 213.

3.1.3.1 Additional or Modified Visual Classes

The Visual classes were enhanced to include:

- **Spooled file viewer**

  The SpooledFileViewer is a new class that can be used to dynamically convert an AS/400 spooled file into a graphical image that can be viewed on a client. However, be aware that this class requires the OS/400 level to be V4R4 and have the AFP Utilities licensed product installed.

- **Jobs**

  VJobList and VJob objects can be added to AS/400 panes (such as the AS400ExplorerPane) to display many different views of the jobs on a specified AS/400 system.

- **Permission**

  Permission classes enable a Java application to set or get various security information related to an AS/400 object. The visual permissions information can be obtained with VIFSFile and VIFSDirectory classes through an AS/400 plane.

- **System values**

  Adding a VSystemValue object to an AS/400 pane allows the user to view and change system values. Different views of system values can be obtained by adding them to different types of AS/400 panes.

- **Users and groups**

  Groups and individual users can be viewed and modified with the VUserAndGroup and VUserList classes. Some potentially useful methods are the setUserInfo() and setGroupInfo() methods that are available with VUsersAndGroup objects.

3.1.3.2 Additional or Modified Access Classes

The access classes were enhanced to include:

- **SSL support**

  New for V4R4 is the ability for host servers to communicate using SSL (Secure Sockets Layer) support. To provide this function in Java, the AS/400 Toolbox for Java now has a SecureAS400 Object. SSL conversations can only take place between an AS/400 Toolbox for Java class and a V4R4 system that has SSL enabled Host Servers. For details on how to use this new support, see Section 10.2, “Securing Applications with SSL” on page 371.

- **JDBC 2.0**

  The JDBC 2.0 specification is a core part of Java 2 (JDK 1.2). By using JDBC 2.0 on a client, it is possible to use scrollable cursors to traverse a result set. In addition, JDBC 2.0 support enables an application to optimize data transfer from a database server. A JDBC 2.0 application can control the number of rows downloaded as a result of executing an SQL statement. For more information on JDBC 2.0, see Section 5.7, “JDBC 2.0” on page 246.

- **Data areas**

  Using the DataArea and associated classes allow a Java application to set, retrieve, and monitor AS/400 Data Area objects. It supports character, decimal, logical, and local data areas.

- **Message file**

  The V4R4 Toolbox allows programs to use AS/400 message files. The MessageFile class allows you to receive a message from an AS/400 message file. The MessageFile class returns an AS400Message object that contains the message.

- **Permission**

  The permissions classes can be used to request and set permission information associated with AS/400 IFS objects. For example, it is possible to list the user profiles that have explicit authority to a file or directory.

- **System status**

  Using the SystemStatus and SystemPool classes, you can dynamically retrieve high-level information about the AS/400 work management status. Using the SystemPool classes, you can even modify the pool sizes and other work management related values.

- **System values**

  You can use SystemValue and SystemValueList objects to retrieve and set system value objects. AS/400 security will always prevent an application from changing a value to which it is not authorized.

3.1.3.3 Additional New Functions

AS/400 Toolbox for Java Modification 2 also offers these new functions:

- **Graphical Toolbox and Panel Definition Markup Language (PDML)**

  The Graphical Toolbox is a tool and PDML is a powerful language that enables you to define panel layouts. PDML is an extension to XML. This way, you can avoid AWT and Swing programming. For more information on the PDML and the Graphical Toolbox, see Section 5.3, “PDML” on page 217.

- **Program Call Markup Language (PCML)**

  PCML is an alternative way to call AS/400 programs. It allows you to define an interface using PCML, which is another extension to XML. PCML simplifies calling AS/400 objects since it performs operations such as the parameter conversions that are required when using a ProgramCall method. For more information on PCML, see Section 5.6, “PCML Examples” on page 241.

- **JarMaker and AS400ToolboxJarMaker**

  These two classes are used to reduce the size of a JAR or ZIP file. The AS400ToolboxJarMaker is an extension to the JarMaker class. It knows about AS/400 components, such as ProgramCall, so that specific components can be extracted from an AS/400 Toolbox for Java archive. Using these tools, you can reduce the size of deployment Java archives. See Section 10.1, “Java Archive Files” on page 365, for more information about JarMaker.

3.1.3.4 Performance Enhancements

Many of the AS/400 classes have been coded to automatically detect if they are running directly on an AS/400 system. With the exception of the JDBC and the IFS classes, the toolbox classes can communicate more efficiently and make direct calls to existing AS/400 APIs. Additional performance improvements can be made if the AS/400 System Name property is set to localhost and the User ID and password is set to *current. It is possible to use the setMustUseSockets(boolean) method to prevent or allow the direct calling of AS/400 APIs.

See Chapter 5, “AS/400 Toolbox for Java Modification 2” on page 213, for more information on using the new AS/400 Toolbox for Java classes. For information on SSL and the JarMaker tools, refer to Chapter 10, “Deployment Considerations and Tools” on page 365.

3.1.4 Supported Platforms

The AS/400 Toolbox for Java is pure Java. It runs on any platform that fully supports the Java JVM 1.1 specification. For Java applications, the Toolbox is supported on the following platforms:

- IBM AS/400 V4R2 or later
- IBM OS/2 Warp 4.0
- IBM AIX 4.1.4
- Sun Solaris 2.5

- Microsoft Windows 95/98
- Microsoft Windows NT

For Java applets, a Web browser that fully supports JVM 1.1 or later is required. To use JDBC 2.0 support, Java 2 (JDK 1.2) is required.

The AS/400 Toolbox for Java requires both the TC1 LPP (TCP/IP Connectivity Utilities for AS/400) and the Host Server option of OS/400 to be installed and configured on the AS/400 system. The toolbox (modification 0 and modification 1) can connect to V3R2, V3R7, V4R1, and V4R2 or later releases of OS/400. Modification 2 can connect to V4R2 or later releases of OS/400.

3.1.5 Application Developer Usage

Any Java Integrated Development Environment (IDE) can be used with the AS/400 Toolbox for Java. Java source can be kept on your client workstation or in the AS/400 integrated file system and accessed using a network drive.

3.1.6 AS/400 Host Servers

The AS/400 host servers must be running. Use the AS/400 command `STRHOSTSVR *ALL` to start the host servers. When working with DDM, the TCP/IP server for DDM must be running. Use the AS/400 command `STRTCPSVR *DDM` to start the TCP/IP server for DDM.

Ensure that the QUSER user profile is enabled and that the password has not expired. QUSER is used by the servers at start-up time.

3.1.7 AS400 Object, Infrastructure, and Sign-On

An AS400 object manages the following elements:

- *A set of socket connections to the AS/400 system*

  Each AS400 object contains one set of socket connections (up to one connection for each service type). This allows the Java programmer to control the number of connections to the AS/400 system. To optimize communications performance, a Java program can create multiple AS400 objects for the same AS/400 system. This allows multiple socket connections to the AS/400 system. Java programs that want to conserve AS/400 resources create only one AS400 object. This reduces the number of connections and reduces the amount of resources used on the AS/400 system.

- *Sign-on behavior for the AS/400 system*

  This includes prompting the user for sign-on information, password caching, and default user management.

- *Prompting for sign-on information*

  Prompting for a user ID and password may occur when connecting to the AS/400 system. Java programs can turn off prompting and graphical message windows displayed by the AS400 object. An example is an application running on a gateway on behalf of many clients. If prompts and messages are displayed on the gateway machine, the user has no way of interacting with the prompts.

- *Password caching*

  To minimize the number of times a user has to type sign-on information, password caching can be used. The password cache applies to all AS400 objects that represent an AS/400 system within a Java virtual machine. This means a cached password in one Java virtual machine is not visible to another virtual machine. The cache is discarded when the last AS400 object is destroyed. The sign-on dialog has a check box that gives the user the option to not cache any given password. When an AS400 object is constructed, the Java program has the option to supply the user ID and password. Passwords supplied on constructors are not cached.

- *Default user management*

  To minimize the number of times a user has to sign on, a default user ID can be used. The default user ID is used when a user ID is not provided by the Java program. The default user ID can be set either by the Java program or through the user interface. If the default user ID is not established, the sign-on dialog allows the user to set the default user ID. Once the default user ID is established for a given AS/400 system, the sign-on dialog does not allow the default user ID to be changed.

The Java program must provide an AS400 object when using an instance of a class that accesses the AS/400 system. For example, the CommandCall object requires an AS400 object before it can send commands to the AS/400 system.

3.2 AS/400 Toolbox for Java and Host Servers

This set of interfaces provides the infrastructure needed to create and maintain socket connections to the AS/400 servers, send and receive data streams, and handle a sign on. This group of classes includes a private AS/400 security manager class that maintains a list of validated AS/400 systems and sign-on information for the system. These classes use the sign-on server and the central server to interact with the AS/400 system.

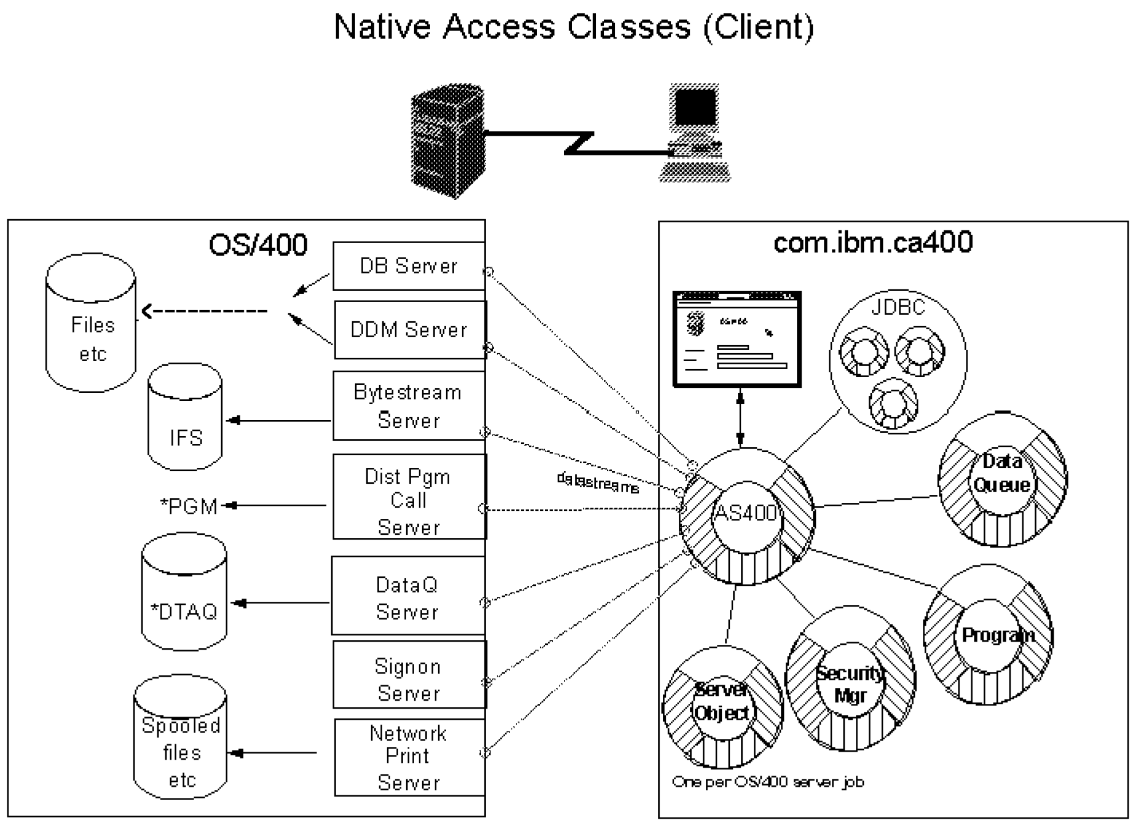


Figure 76. Java Host Server Overview

3.2.1 Data Descriptions and Conversions

The data conversion APIs provide the capability to convert numeric and character data between AS/400 and Java formats. Conversion may be needed when accessing AS/400 data from a Java program. The data conversion APIs support the conversion of various numeric formats and between various EBCDIC code pages and unicode.

Two levels of support are provided by the data conversion APIs:

- Data types convert data between the AS/400 and Java format.
- Record-level conversion builds on data types to support converting all fields in a record with a single method call. The RecordFormat class allows the program to describe data that makes up a DataQueueEntry, ProgramCall parameter, Record-level database access, or any buffer of AS/400 data. The Record class allows the program to convert the contents of the record and access the data by field name.

3.2.2 AS/400 Data Types

Table 5 on page 98 shows a set of classes, which represent AS/400 data as Java data types to simplify the handling of AS/400 data for Java programmers. Each class converts data between the AS/400 representation and the Java representation of the data.

Table 5. AS/400 Data Types

Class	Description
AS400Bin2	Provides a converter between a Short object and a signed two-byte binary number.
AS400Bin4	Provides a converter between an Integer object and a signed four-byte binary number.
AS400UnsignedBin2	Provides a converter between an Integer object and an unsigned two-byte binary number.
AS400UnsignedBin4	Provides a converter between a Long object and an unsigned four-byte binary number.
AS400Float4	Provides a converter between a Float object and a four-byte floating point number.
AS400Float8	Provides a converter between a Double object and an eight-byte floating point number.
AS400PackedDecimal	Provides a converter between a BigDecimal object and a packed decimal format floating point number.
AS400ZonedDecimal	Provides a converter between a BigDecimal object and a zoned decimal format floating point number.
AS400Text	Provides character set conversion between Java String objects and AS/400 native code pages.
AS400ByteArray	Provides a converter between a byte array and fixed-length byte array representing AS/400 data that cannot be converted.
AS400Array	Provides a composite data type representing an array of AS400DataType objects.
AS400Structure	Provides a composite data type representing a structure of AS400DataType objects.
AS400JDBCBlob	Provides access to binary large objects.
AS400JDBCBlobLocator	Provides access to binary large objects.
AS400JDBCClob	Provides access to character large objects.
AS400JDBCClobLocator	Provides access to character large objects.
AS400JDBCInputStream	Provides access to binary data using an input stream.

3.2.3 Record Level Conversions

The classes, which are shown in Table 6 on page 99, allow Java programs to define AS/400 data in a way that is similar to how data is defined on the AS/400 system. These classes provide a way to define field descriptions and record formats, which are used to describe AS/400 data. Data is accessed in a record object using field names defined by the field description object of the associated record format object.

Table 6. Record Level Conversions

Public Class	Description
FieldDescription	Abstract class that describes a field of data from an AS/400 system. Can be a DDS field or program described data. Contains an AS400DataType, length, and name. The following field description classes are provided: • BinaryFieldDescription • CharacterFieldDescription • DBCSEitherFieldDescription • DBCSGraphicFieldDescription • DBCSOnlyFieldDescription • DBCSOpenFieldDescription • DateFieldDescription • FloatFieldDescription • HexFieldDescription • PackedDecimalFieldDescription • TimeFieldDescription • TimestampFieldDescription • ZonedFieldDescription
RecordFormat	General purpose class that describes a structure of fields. It may represent a record from an AS/400 file, or a structure of data returned by a program, data queue, and so on. A RecordFormat object contains FieldDescriptions.
Record	Holds the actual data described by a RecordFormat object. Data is accessed in a Record object using field names defined by the FieldDescription objects of the associated RecordFormat object.

3.2.4 JDBC Specification

The public classes and methods for the JDBC driver are implementations of the interface defined by the Javasoft JDBC specification. The JDBC driver and supporting classes are completely written in Java, and do not require any other client code. As illustrated in Figure 77 on page 100, the JDBC driver provided by the Toolbox can be used instead of a JDBC/ODBC bridge driver.

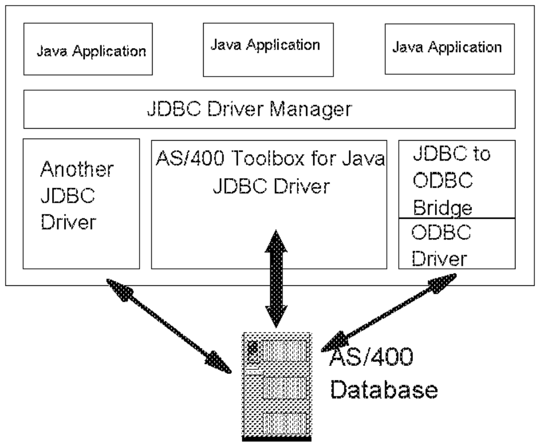


Figure 77. JDBC Interface to the AS/400 System

No unique public interfaces are defined by the AS/400 Toolbox for Java implementation of the interface. These classes use the database servers to access the AS/400 system (see Table 7).

Table 7. JDBC Classes

Public Class	Description
AS400JDBCCallableStatement	Used to execute SQL stored procedures.
AS400JDBCConnection	Represents a session with a specific DB2/400 database. Within the context of a Connection, SQL statements are executed and results are returned.
AS400JDBCDatabaseMetaData	Provides information about the database as a whole.
AS400JDBCDriver	Represents the AS/400 Toolbox for Java JDBC driver for interacting with the JDBC DriverManager.
AS400JDBCPreparedStatement	Class that stores and executes a pre-compiled SQL statement.
AS400JDBCResultSet	Provides access to the data generated when an SQL statement is executed.
AS400JDBCResultSetMetaData	Used to find out about the types and properties of the columns in a ResultSet.
AS400JDBCException	Provides information on database access errors, including a string that describes the error.
AS400JDBCStatement	Used to execute a static SQL statement and obtain the result set produced when the statement is executed.
AS400JDBCWarning	An extension of SQLWarning. Used to report JDBC warnings through the getWarnings() method.

3.2.5 Record-Level File Access

AS/400 physical files can be accessed one record at a time using the public interface of these classes. Files and members can be created, read, deleted, and updated. The record format can be defined by the programmer at application development time. Or, the format can be retrieved at runtime by the AS/400 Toolbox for Java support. These classes use the DDM server to access the AS/400 system.

3.2.5.1 Record Level Access — Pre-V4R2M0

When using Record Level Access over an IP network, be aware that pre-V4R2 versions of OS/400 do not offer this support using native TCP/IP. You need to perform additional setup before this option can work. On pre-V4R2 operating systems, you need to download additional PTFs and perform extra steps as described in the PTF cover letter. Table 8 contains a list of version-specific, required PTFs. Download the PTF for the specified version only. You must obtain the most recent supersede of a PTF if one is available.

Table 8. PTF List for Record-Level Access

OS/400 Versions	PROD ID / PTF
V3R1	5763SS1 / SF46301
V3R2	5763SS1 / SF46302
V3R6	5716SS1 / SF46306
V3R7	5716SS1 / SF46303
V4R1	5769SS1 / SF46313

The public classes in Table 9 are defined and implemented.

Table 9. Record-Level Access Public Classes

Public Class	Description
AS400File	Represents an AS/400 physical or logical file.
SequentialFile	Represents an AS/400 file that is to be accessed by relative record number.
KeyedFile	Represents an AS/400 file that contains key fields and is to be accessed by key. This class can be used to create a physical file, access the data in a file using a key, write data by key, and delete a file.

3.2.6 Integrated File System

The file system classes allow access to file objects that are in the AS/400 integrated file system (IFS). The original intent was to extend the file classes that are in the Java.io package. However, this was prohibited by the design and implementation of the Java.io classes. Instead, new classes were created to represent a file object in the integrated file system. A program can open an input or output stream on a file object, or read and write data from or to any specified location in the file. These classes use the bytestream server to access the AS/400 system (see Table 10 on page 102).

Table 10. Integrated File System Classes

Public Class	Description
IFSFile	Represents an object in the AS/400 integrated file system. Similar to java.io.File.
IFSFileDescriptor	A file handle that references an open file. Similar to java.io.FileDescriptor.
IFSFileInputStream	Used to read (open an input stream on) an object in the integrated file system. Similar to java.io.FileInputStream.
IFSFileOutputStream	Used to write to (open an output stream on) a file in the integrated file system. Similar to java.io.FileOutputStream.
IFSKey	Provides byte range locking on a file.
IFSRandomAccessFile	Allows reading and writing of data from or to any specified location of a file in the integrated file system. Similar to java.io.RandomAccessFile.

3.2.7 Print

Print support in the Java language does not make it possible to plug in as a print provider. The existing Java print classes use the client's native print provider. The toolbox print support provides a set of classes that are similar to the native Java classes, but use the AS/400 print services instead of the native print provider.

In addition, some print management classes are provided to enable management of printers, output queues, and spooled files. These classes are listed Table 11 on page 103. All classes use the network print server to access the AS/400 system.

The toolbox print support requires additional function in the network print server. This function is provided by PTFs. At the time this redbook was published, the PTFs were:

- For **V4R3**, 5769SS1—PTF SF48498
- For **V4R2**, 5769SS1—PTF SF46476
- For **V4R1**, 5769SS1—PTF SF41926, PTF SF42518, and PTF SF46470
- For **V3R7**, 5716SS1—PTF SF42316 and PTF SF42516
- For **V3R2**, 5763SS1—PTF SF42344 and PTF SF42515

Table 11. Printer Classes

Public Class	Description
AFPResource	Class for working with AS/400 AFP resources (page segments, overlays, fonts, and so on).
AFPResourceList	List of AFP resources
OutputQueue	Represents an AS/400 output queue
OutputQueueList	List of output queues
Printer	Represents an AS/400 printer device
PrinterFile	Represents an AS/400 printer file
PrinterFileList	List of printer files
PrinterList	List of AS/400 printer devices
SpooledFile	Represents an AS/400 spooled file
SpooledFileList	List of AS/400 spooled files
SpooledFileMessage	Represents a message that an AS/400 spooled file is waiting on
SpooledFileOutputStream	Used to write data to a new AS/400 spooled file
WriterJob	Represents an AS/400 printer writer
WriterJobList	List of spoolwriter jobs

3.2.8 Command

Any AS/400 batch command can be run using command support. A list of AS/400 messages that are generated when the command is run can be retrieved after the command completes. These classes use the distributed program call server to access the AS/400 system (see Table 12).

Table 12. Command Classes

Public Class	Description
CommandCall	Used to specify and run an AS/400 command string.
AS400Message	Used to retrieve messages that were generated when a command was run on an AS/400 system.

3.2.9 Program Call

Any AS/400 program can be called using program call support. Parameters may be passed to the AS/400 program, and data can be returned by the AS/400 program to the Java calling program. These classes use the Distributed Program Call server to access the AS/400 system (see Table 13 on page 104).

Table 13. Program Call Classes

Public Class	Description
ProgramCall	Used to call an AS/400 program, passing parameters to the program and receiving returned data from the program. The program runs under the DPC server job. When the program exits, data is returned to the calling Java program as byte data.
ProgramParameter	Represents a parameter that can be passed to an AS/400 program. The parameter can be an input parameter, an output parameter, or and input/output parameter.
AS400Message	Used to retrieve messages that were generated when a program call failed on an AS/400 system.

3.2.10 Data Queue

Both keyed and sequential data queues can be accessed using the public interfaces of the data queues classes. Entries can be placed on a data queue or removed. Data queues can be created or deleted on the AS/400 system. These classes use the Data Queues server to access the AS/400 system.

The public classes in Table 14 are defined and implemented.

Table 14. Data Queue Classes

Public Class	Description
DataQueue	Represents an AS/400 data queue that is accessed sequentially.
KeyedDataQueue	Represents an AS/400 data queue that is accessed using a key.
DataQueueEntry	Represents data that is read from a sequential data queue. The data can be accessed as byte data or as a string.
KeyedDataQueueEntry	Represents data that is read from a keyed data queue. The returned data and the key can be accessed as byte data or as a string.

3.3 How the AS/400 System Fits into This Picture

The AS/400 system can be a repository for data, programs, HTML documents, applets, and Java applications. An HTTP server running on the AS/400 system can be used to serve Web pages and applets. The class files for Java applications can reside in the integrated file system of the AS/400 system and accessed using a mapped drive.

When an HTML document containing an applet is served from the AS/400 system, the class files are loaded from that AS/400 system. The applet can access only that AS/400 system. For applications, the class files are located using the CLASSPATH environment variable. On a network station (or comparable hardware), the CLASSPATH variable can be set to include the toolbox class files. On a Windows (or other client operating system) workstation, there are two options. The workstation can have a mapped drive to the AS/400 system (this requires Client Access), or the class files can be copied to the client. In either case, the CLASSPATH environment variable must be appropriately set to locate the class files.

No new function is needed on the AS/400 system to use the AS/400 Toolbox for Java because the existing OS/400 servers are used. These servers are used:

- Database servers
- Distributed program call server
- Data queue server
- Network print server
- Bytestream server
- Sign-on server
- Central server
- DDM server

From the perspective of the AS/400 Toolbox for Java, the servers are a black-box interface to perform functions on the AS/400 system. All requests directed toward data or resources on the AS/400 system funnel through the servers.

3.3.1 Security

Each connection to the AS/400 system is validated for user ID and password. The Toolbox classes enable a single sign on for multiple connections to the AS/400 system. Password expiration warnings and changing a password are supported.

AS/400 security is enforced using the same security model as Client Access/400. The user must have a valid AS/400 sign on, and proper authority to the AS400 objects or resources. User ID and password are prompted if one is not provided by the user. The user ID and password are verified on the AS/400 system. All passwords are encrypted prior to sending them to the AS/400 system. To ensure additional security, passwords are not passed between classes except between the sign on GUI and the security class.

The servers run with the authorities of the user ID that is passed when the connection is made. No authorities are adopted.

3.3.2 National Language Support

The AS/400 Toolbox for Java uses the internationalization support available with JDK 1.1. Translatable information resides in property files at runtime. Resource bundles are used to retrieve the proper text depending on the locale. Java internationalization support is built on Java locale objects that are defined by a country code, a language code, and a variant. The country codes are the two-letter ISO-3166 standard. The language codes are the two-letter ISO-390 standard. JDK 1.1 supports 27 different locales including both single-byte and double-byte locales, but no right-to-left languages. The AS/400 Toolbox for Java does not attempt to add additional locales and is limited to the locales that Java supports.

3.3.3 Save and Restore Considerations

The Java classes and applets can reside in the IFS. There are no unique considerations for saving and restoring these entities.

3.3.4 Installation and Run-Time Considerations

The AS/400 Toolbox for Java class files are used at development time by the compiler and at application or applet runtime. The JDK set a precedent for how class files are accessed at runtime. It is based on a thin client/browser model. That is, the class files reside on a server and are brought to the client when an applet or application is loaded. There is no functional requirement to install the class files on the client.

For *applets and applications loaded locally,* the *class files* are located using the `codebase` applet tag. When the HTML document containing the applet is served from the AS/400 system (through the HTTP server), the class files are loaded from that AS/400 system. The applet can access only that particular AS/400 system.

3.3.5 Error Recovery Considerations

The Java model for error processing is to throw exceptions instead of returning return codes. The AS/400 Toolbox for Java follows this model. When an AS/400 Toolbox for Java class discovers an error, it throws an exception. Some exceptions contain a documented return code value. Some exceptions allow retrieving text that describes the error. The description for each AS/400 Toolbox for Java API includes a list of exceptions that can be thrown by the API. The application can catch these exceptions and handle them based on the API and exception returned.

The AS/400 Toolbox for Java handles errors so that the degree of success of an API is obvious to the application. This is a data integrity statement. The application knows the state of the data of an API call based on the exception (or lack of exception) generated.

In addition to throwing an exception, an error is logged to the AS/400 Toolbox for Java error log in some cases. An error is logged for unexpected conditions (for First Failure Support Technology (FFDC)), severe errors, and other places a message can help the user recover from the error. Errors are only logged if logging is turned on by the Java program.

3.3.6 Mapping AS/400 Data Types to Java Data Types

Table 15 shows you how AS/400 data types map to Java data types.

Table 15. AS/400 Types Mapped to Java Types

AS/400 Type	Java Type
binary (1 – 4 digits)	short
binary (5 – 9 digits)	int
character	String
date	String
float (single precision)	float
float (double precision)	double
hex	byte
packed decimal	BigDecimal
zoned decimal	BigDecimal
time	String
timestamp	String

3.4 Introduction to Application Examples

The remainder of this chapter covers application examples, including:

- AS/400 database access:
  - JDBC
  - JDBC stored procedures
  - DDM Record Level Access
  - Using distributed program call
  - Using data queues
- Network print
- Integrated file system

3.5 AS/400 Database Access

The database access example applications shown in the remainder of this chapter are for the most part functionally equivalent. They allow for the retrieval, update, add, and delete of a record from a PARTS file on the AS/400 system. All of these functions support the display of the entire PARTS file in a list box.

The PARTS file is defined as shown in Table 16 on page 108.

Table 16. AS/400 Parts File

Field Name	Field Description	Length	Decimals	Type
PARTNO	Part Number	5	0	Zoned
PARTDS	Description	25		Char
PARTQY	Quantity	5	0	Packed
PARTPR	Price	6	2	Packed
PARTDT	Part Shipment Date	10		Date

A start time and end time are updated on the window so that different AS/400 access methods can be compared for performance. The examples were built using IBM's VisualAge for Java development environment. The list box and the message box widgets used in all of the examples are from IBM. Both of them can be found in the project IBM Enterprise Data Access Libraries 2.0, package com.ibm.ivj.eab.dab.

Complete listings of all RPG and DDS source code can be found in Appendix B, “AS/400 Source Listings” on page 399.

Both the Java code and the AS/400 libraries are available for you to download from the Internet. See Section A.1, “Downloading the Files from the Internet” on page 396, for more information.

3.5.1 JDBC Interface

The AS/400 Toolbox for Java implements the standard JDBC interface for access to data. JDBC defines a consistent set of classes and interfaces for communication with a database server.

The advantages of using JDBC are that it is an industry standard and it is easy to use. Being an industry standard allows the AS/400 developer to use generic Java applets and applications and point them to the AS/400 system for data storage and retrieval. Additionally, the Java developer can use the AS/400 system as a server without worrying about AS/400 specific implementation issues. JDBC can be easier to use than the other classes in the AS/400 toolbox because the driver takes care of all data conversion issues. AS/400 data types are automatically mapped to Java data types. The Java developer does not need to be concerned with the actual data representation on the AS/400 system.

When using JDBC, you need to reference the following interfaces under the java.sql package:

- **Driver** — Creates the connection and returns information about the driver version.
- **Connection** — Represents a connection to a specific database.
- **Statement** — Runs SQL statements and obtains the results.
- **PreparedStatement** — Runs pre-compiled SQL statements.
- **CallableStatement** — Runs SQL stored procedures.

- **ResultSet** — Provides access to a table of data generated by running an SQL statement or databaseMetaData catalog method.
- **ResultSetMetaData** — Determines the types and properties of the columns in a ResultSet.
- **DatabaseMetaData** — Provides catalog methods, which provide information about the database.
- **Blob** — The representation (mapping) in the Java programming language of an SQL BLOB. An SQL BLOB is a built-in type that stores a Binary Large Object as a column value in a row of a database table.
- **Clob** — The representation (mapping) in the Java programming language of an SQL CLOB. An SQL CLOB is a built-in type that stores a Character Large Object as a column value in a row of a database table.

Accessing data on the AS/400 system using JDBC in your application involves the following steps:

1. Register the AS/400 JDBC driver.
2. Connect to the database.
3. Define and Prepare SQL statements.
4. Execute SQL statements.
5. Obtain and process results of the statements.
6. Close the statements.
7. Close the database connection.

3.5.2 JDBC Performance Tips

JDBC from a Java program communicates with the same server program on the AS/400 system as the Client Access/400 ODBC driver. Any server side tuning suggestions for ODBC apply to JDBC. For more information on ODBC performance related issues, please refer to redbook *AS/400 Client/Server Performance Using the Windows Clients*, SG24-4526.

JDBC allows SQL statements to be sent to the AS/400 system for execution. If an SQL statement is run more than one time, use a PreparedStatement object to execute the statement. A PreparedStatement compiles the SQL once, so that subsequent executions run quickly. If a plain Statement object is used, the SQL must be compiled and run every time it is executed. Use *Extended Dynamic* support. It caches the SQL statements in SQL packages on the AS/400 system. Also turn on **package cache**, and cache SQL statements in memory.

Do not use a PreparedStatement object if an SQL statement is run only one time. Compiling and running a statement at the same time has less overhead than compiling the statement and running it in two separate operations.

Consider using JDBC stored procedures. In a client/server environment, stored procedures can help reduce communication I/Os, and therefore, help improve response time.

Use a *just-in-time (JIT)* compiler for your Java execution environment if possible. The latest JIT technology allows Java programs to perform almost as well as native code written in C or C++.

There are many properties that can be specified in the JDBC URL or in the JDBC properties object. Several of these properties can significantly affect the performance of a JDBC client/server application and should be used where possible. The properties control record blocking, package caching, and extended dynamic support. Selected properties and their settings are listed in Table 17. Other non-performance properties can be found in the Toolbox documentation.

Note: When using the properties for JDBC in your connection to the AS/400 system, all property keywords and values have to be coded in lowercase letters (see the example in Figure 81 on page 117).

Table 17. General Properties

Property	Description	Choices	Default
"user"	Specifies the user name for connecting to the AS/400 server. If none is specified, the user is prompted, unless the "prompt" property is set to "false", in which case an attempt to connect will fail.	AS/400 user	User is prompted
"password"	Specifies the password for connecting to the AS/400 server. If none is specified, the user is prompted, unless the "prompt" property is set to "false", in which case an attempt to connect will fail.	AS/400 password	User is prompted
"prompt"	Specifies whether the user should be prompted if a user name or password is needed to connect to the AS/400 server. If a connection cannot be made without prompting the user, and this property is set to "false", an attempt to connect will fail.	"true" or "false"	"true"
"secure"	Specifies whether SSL (Secure Sockets Layer) should be used.	"true" or false"	"false"

See Table 18 through Table 22 on page 114 for a complete list of the JDBC properties (none of them are required).

Table 18. Server Properties

Property	Description	Choices	Default
"libraries"	Specifies the AS/400 libraries to add to the server job's library list. The libraries are delimited by commas or spaces, and "*LIBL" may be used as a place holder for the server job's current library list. The library list is used for resolving unqualified stored procedure calls and finding schemas in databaseMetaData catalog methods. If "*LIBL" is not specified, the specified libraries replace the server job's current library list. In addition, if no default schema is specified in the URL, the first library listed in this property is also the default schema, which is used to resolve unqualified names in SQL statements.	AS/400 libraries	"*LIBL"
"trans-action isolation"	Specifies the default transaction isolation.	"none", "read committed", "read uncommitted", "repeatable read", or "serializable"	"none"

Table 19. Format Properties

Property	Description	Choices	Default
"date format"	Specifies the date format used in date literals within SQL statements.	"mdy", "dmy", "ymd", "usa", "iso", "eur", or "jis"	(server job)
"date separator"	Specifies the date separator used in date literals within SQL statements. This property has no effect unless the "date format" property is set to "julian", "mdy", "dmy", or "ymd".	"/" (slash), "-" (dash), "." (period), "," (comma), or "b" (space)	(server job)
"decimal separator"	Specifies the decimal separator used in numeric literals within SQL statements.	"." (period) or "," (comma)	(server job)
"naming"	Specifies the naming convention used when referring to tables.	"sql" (for example, schema.table), "system" (for example, schema/table)	"sql"
"time format"	Specifies the time format used in time literals within SQL statements.	"hms", "usa", "iso", "eur", or "jis"	(server job)
"time separator"	Specifies the time separator used in time literals within SQL statements. This property has no effect unless the "date format" property is set to "hms".	":" (colon), "." (period), "," (comma), or "b" (space)	(server job)
lob threshold	Specifies the maximum LOB (large object) size (in kilobytes) that can be retrieved as part of a result set. LOBs that are larger than this threshold will be retrieved in pieces using extra communication to the server. Larger LOB thresholds will reduce the frequency of communication to the server, but will download more LOB data, even if it is not used. Smaller LOB thresholds may increase the frequency of communication to the server, but will only download LOB data as it is needed.	"0" - "4194304"	"0"

Table 20. Performance Properties

Property	Description	Choices	Default
"block criteria"	Specifies the criteria for retrieving data from the AS/400 server in blocks of records. Specifying a non-zero value for this property reduces the frequency of communication to the server, and therefore, increases performance. Ensure that record blocking is off if the cursor is going to be used for subsequent UPDATEs. Otherwise, the row that is updated may not necessarily be the current row.	"0" (no record blocking) "1" (block if FOR FETCH ONLY is specified) "2" (block unless FOR UPDATE is specified)	"2"
"block size"	Specifies the block size (in kilobytes) to retrieve from the AS/400 server and cache on the client. This property has no effect unless the "block criteria" property is non-zero. Larger block sizes reduce the frequency of communication to the server, and therefore, increase performance.	"8", "16", "32", "64", "128", "256", or "512"	"32"

Property	Description	Choices	Default
"prefetch"	Specifies whether to prefetch data upon executing a SELECT statement. This increases performance when accessing the initial rows in the ResultSet.	"true" or "false"	"true"
"extended dynamic"	Specifies whether to use extended dynamic support. Extended dynamic support provides a mechanism for caching dynamic SQL statements on the server. The first time a particular SQL statement is run, it is stored in an SQL package on the server. On subsequent runs of the same SQL statement, the server can skip a significant part of the processing by using information stored in the SQL package. If this is set to "true", a package name must be set using the "package" property.	"true" or "false"	"false"
"package"	Specifies the base name of the SQL package. Extended dynamic support works best when this is derived from the application name. Note that only the first seven characters are significant. This property has no effect unless the "extended dynamic" property is set to "true". In addition, this property must be set if the "extended dynamic" property is set to "true".	SQL package	""
"package criteria"	Specifies the type of SQL statements to be stored in the SQL package. This can be useful to improve the performance of complex join conditions. This property has no effect unless the "extended dynamic" property is set to "true".	"default" (only store SQL statements with parameter markers in the package) "select" (store all SQL SELECT statements to be stored in the package)	"default"
"package library"	Specifies the library for the SQL package. This property has no effect unless the "extended dynamic" property is set to "true".	Library for SQL package	"QGPL"
"package cache"	Specifies whether to cache SQL packages in memory. Caching SQL packages locally reduces the amount of communication to the server in some cases. This property has no effect unless the "extended dynamic" property is set to "true".	"true" or "false"	"false"
"package clear"	Specifies whether to clear SQL packages when they become full. Clearing an SQL package results in removing all SQL statements that have been stored in the SQL package. This property has no effect unless the "extended dynamic" property is set to "true".	"true" or "false"	"false"
"package add"	Specifies whether to add statements to an existing SQL package. This property has no effect unless the "extended dynamic" property is set to "true".	"true" or "false"	"true", "false"
"package error"	Specifies the action to take when SQL package errors occur. When an SQL package error occurs, the driver optionally throws an SQLException or post a warning to the Connection, based on the value of this property. This property has no effect unless the "extended dynamic" property is set to "true".	"exception", "warning", or "none"	"warning"

Table 21. Sort Properties

Property	Description	Choices	Default
"sort"	Specifies how the server sorts records before sending them to the client.	• "hex" (base the sort on hexadecimal values) • "job" (base the sort on the setting for the server job) • "language" (base the sort on the language set in the "sort language" property) • "table" (base the sort on the sort sequence table set in the "sort table" property)	"job"
"sort language"	Specifies a three-character language ID to use for selection of a sort sequence. This property has no effect unless the "sort" property is set to "language".	Language ID	(locale)
"sort table"	Specifies the library and file name of a sort sequence table stored on the AS/400 server. This property has no effect unless the "sort" property is set to "table".	Qualified sort table name	""
"sort weight"	Specifies how the server treats case while sorting records. This property has no effect unless the "sort" property is set to "language".	•"shared" (upper- and lower-case characters are sorted as the same character) • "unique" (upper- and lower-case characters are sorted as different characters)	"shared"

Table 22. Other Properties

Property	Description	Choices	Default
"access"	Specifies the level of database access for the connection.	• "all" (all SQL statements allowed) • "read call" (SELECT and CALL statements allowed) • "read only" (SELECT statements only)	"all"
"errors"	Specifies the amount of detail to be returned in the message for errors that occur on the AS/400 server.	"basic" or "full"	"basic"
"remarks"	Specifies the source of the text for REMARKS columns in ResultSets returned by databaseMetaData methods.	"sql" (SQL object comment) or "system" (OS/400 object description)	"system"
"translate binary"	Specifies whether binary data is translated. If this property is set to "true", the BINARY and VARBINARY fields are treated as CHAR and VARCHAR fields.	"true" or "false"	"false"
"trace"	Specifies whether trace messages should be logged. Trace messages are useful for debugging programs that call JDBC. However, there is a performance penalty associated with logging trace messages, so this property should only be set to "true" for debugging. Trace messages are logged to System.out.	"true" or "false"	"false"
"data truncation"	Specifies whether data truncation exceptions are thrown. If this property is set to "true", then data truncation exceptions are thrown if data needs to be truncated when writing to the database. If this property is set to "false", then no such data truncation exceptions are thrown. Either way, data truncation warnings are posted if data needs to be truncated when reading from the database.	"true" or "false"	"false"

3.5.3 JDBC Application Example

This example uses JDBC to access records in an AS/400 database. The client program requests data from the AS/400 database by sending SQL statements to the OS/400 host database server. The host server executes the SQL statement and returns the results to the client program in an SQL result set. The JDBC support handles all data conversions (see Figure 78 on page 115).

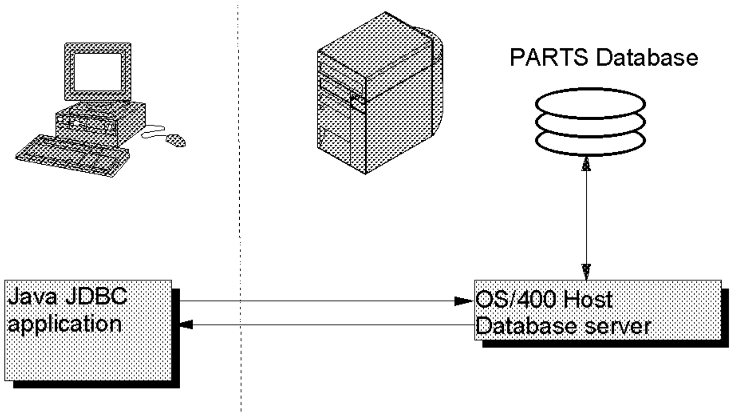
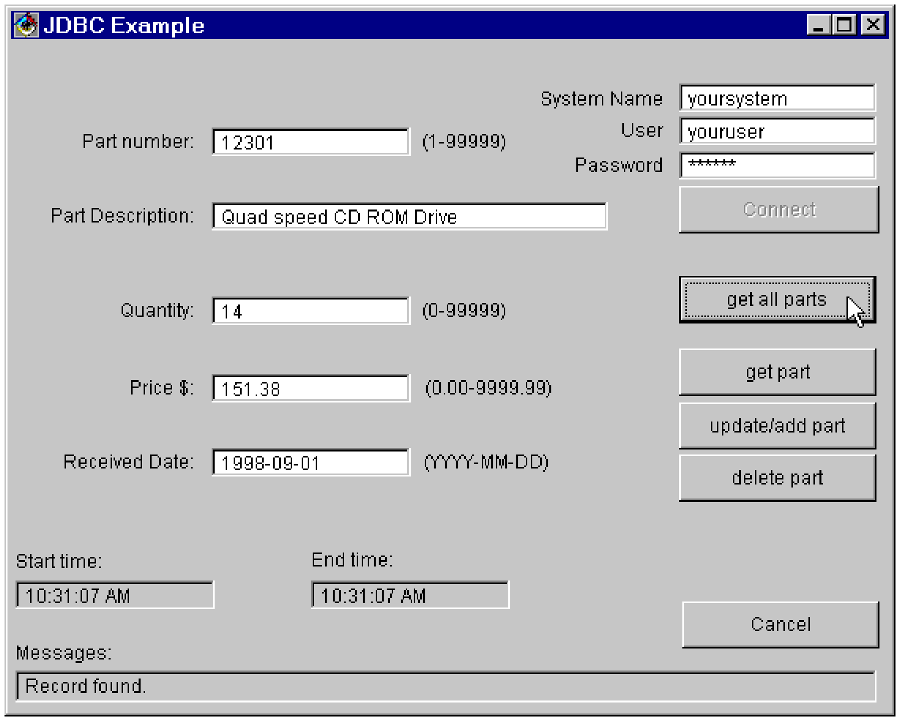


Figure 78. JDBC Application

A single part record can be retrieved, updated, added, and deleted or all part records can be displayed in a list box (see Figure 79). Two classes drive the application: JDBCExample and JDBCExampleDisplayAll.



JDBC Example

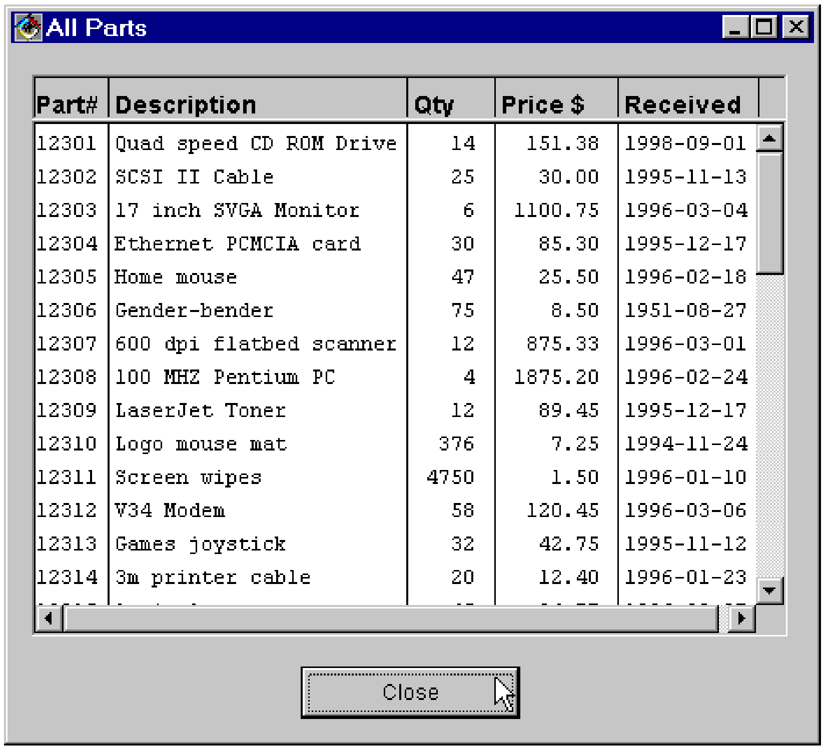
			System Name	yoursystem
Part number:	12301	(1-99999)	User	youruser
			Password	******
Part Description:	Quad speed CD ROM Drive			
Quantity:	14	(0-99999)		
Price $:	151.38	(0.00-9999.99)		
Received Date:	1998-09-01	(YYYY-MM-DD)		

Start time: 10:31:07 AM

End time: 10:31:07 AM

Messages: Record found.

Figure 79. JDBC Example One Part



Part#	Description	Qty	Price $	Received
12301	Quad speed CD ROM Drive	14	151.38	1998-09-01
12302	SCSI II Cable	25	30.00	1995-11-13
12303	17 inch SVGA Monitor	6	1100.75	1996-03-04
12304	Ethernet PCMCIA card	30	85.30	1995-12-17
12305	Home mouse	47	25.50	1996-02-18
12306	Gender-bender	75	8.50	1951-08-27
12307	600 dpi flatbed scanner	12	875.33	1996-03-01
12308	100 MHZ Pentium PC	4	1875.20	1996-02-24
12309	LaserJet Toner	12	89.45	1995-12-17
12310	Logo mouse mat	376	7.25	1994-11-24
12311	Screen wipes	4750	1.50	1996-01-10
12312	V34 Modem	58	120.45	1996-03-06
12313	Games joystick	32	42.75	1995-11-12
12314	3m printer cable	20	12.40	1996-01-23

Figure 80. JDBC Example All Parts

The JBCDExample class creates the main application window, connects and disconnects the database, prepares the SQL statements, provides a GUI to display, and enters and manipulates the values for a single part record. All of the four basic database operations (Create, Read, Update, Delete) are implemented in this class. It instantiates a JDBCExampleDisplayAll object when the Get All Parts button is pressed.

3.5.4 JDBCExample Class

In this section, we investigate the key methods of the JDBCExample class.

3.5.4.1 Instance Variables

The following instance variables for accessing the database are declared for the class:

```
private java.sql.Connection dbConnect;
private java.sql.PreparedStatement psAllRecord;
private java.sql.PreparedStatement psSingleRecord;
private java.sql.PreparedStatement psUpdateRecord;
private java.sql.PreparedStatement psAddRecord;
private java.sql.PreparedStatement psDeleteRecord;
```

3.5.4.2 The connectToDB Method

The connectToDB method is called when the Connect button is pressed. String parameters representing the AS/400 system name, user ID, and password are passed to the method.

```
public String connectToDB(String systemName, String userid, String password)
{
     try
     {
            setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.WAIT_CURSOR));
            java.sql.DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
            dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName +
                   "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
                   "package=JDBCExa;package library=apilib", userid, password);
            psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?");
            psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS ORDER BY PARTNO");
            psUpdateRecord = dbConnect.prepareStatement("UPDATE PARTS SET PARTDS = ?," +
                   " PARTQY = ?, PARTPR = ?, PARTDT = ? WHERE PARTNO = ?");
            psAddRecord = dbConnect.prepareStatement("INSERT INTO PARTS (PARTDS, PARTQY," +
                   " PARTPR, PARTDT, PARTNO) VALUES(?, ?, ?, ?, ?)");
            psDeleteRecord = dbConnect.prepareStatement("DELETE FROM PARTS WHERE PARTNO = ?");
     }
     catch (Exception e)
     {
            showException(e);
            setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
            return "Connect Failed.";
     }
     setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
     return "Connected to AS/400.";
}
```

Figure 81. The connectToDB Method Example

Class: JDBCExample Method: connectToDB

Let us examine the method:

```
java.sql.DriverManager.registerDriver
   (new com.ibmas400.access.AS400JDBCDriver());
```

- This statement loads the JDBC driver into the Java virtual machine. The fully-qualified name of the AS/400 JDBC driver class is passed as a parameter.

```
dbConnect = java.sql.DriverManager.getConnection
   ("jdbc:as400://" + systemName + "/apilib;naming=sql;errors=full;date
   format=iso;extended dynamic=true;package=JDBCEx;package library=apilib",
   userid, password);
```

- This statement creates a java.sql.Connection object called dbConnect. The form of the DriverManager's getConnection method used here takes a URL, user ID, and password parameters. The URL is formatted.

```
jdbc:as400://systemName/defaultLibraryName;parameter1=value1;
parameter2=value2;...
```

- The default library name is optional, as are the properties. We are using APILIB as the default library, and specifying the use of the ISO format for date fields. Error messages must contain all available information. We further specify some parameters for the performance properties to use extended dynamic.

```
psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO =
   ?");
psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS ORDER BY
   PARTNO");
psUpdateRecord = dbConnect.prepareStatement("UPDATE PARTS SET PARTDS = ?," +
   " PARTQY = ?, PARTPR = ?, PARTDT = ? WHERE PARTNO = ?");
psAddRecord = dbConnect.prepareStatement("INSERT INTO PARTS (PARTDS, PARTQY," +
   " PARTPR, PARTDT, PARTNO) VALUES(?, ?, ?, ?, ?)");
psDeleteRecord = dbConnect.prepareStatement("DELETE FROM PARTS WHERE PARTNO =
   ?");
```

• These statements create five preparedStatement objects. PreparedStatements are precompiled SQL statements that are more efficient to execute than plain Statements when run repeatedly. The "?" is used as a parameter marker, with the value set prior to running the PreparedStatement. The first statement creates an object that selects a record from the parts file that has a PARTNO field equal to a value defined later. The second statement creates an object that selects all records from the parts file and orders the result set by the PARTNO. The third statement creates an object that selects a record from the parts file that has a PARTNO field equal to a value defined later. If that record is found, it is updated with the four other parameters yet to define. The fourth statement creates a new record in the parts file using five parameters to define the values of the fields that the new record contains. The fifth statement creates an object that selects a record from the parts file that has a PARTNO field equal to a value defined later. If that record is found, it is deleted from the parts file.

3.5.4.3 The getRecord Method

The getRecord method is called when the get part button is pressed. A string parameter containing the part number is passed, along with the four text field objects that are used to display values of other fields in the part record.

```
public String getRecord(String partNo, java.awt.TextField partDesc, java.awt.TextField partQty,
                       java.awt.TextField partPrice, java.awt.TextField partDate)
{
     java.sql.ResultSet rs = null;
     try
     {
            psSingleRecord.setInt(1, Integer.parseInt(partNo.trim()));
            rs = psSingleRecord.executeQuery();
            if (rs.next())
            {
                   partDesc.setText(rs.getString("PARTDS").trim());
                   partQty.setText(Integer.toString(rs.getInt("PARTQY")));
                   partPrice.setText(rs.getBigDecimal("PARTPR", 2).toString());
                   partDate.setText(rs.getDate("PARTDT").toString());
            }
            else
            {
                   partDesc.setText("");
                   partQty.setText("0");
                   partPrice.setText("0.00");
                   partDate.setText("");
                   return "Record not found.";
            }
     }
     catch (Exception e)
     {
            e.printStackTrace();
            showException(e);
            return "Error during SQL-Execution at SELECT.";
     }
     return "Record found.";
}
```

Figure 82. The getRecord Method Example

Class: JDBCExample Method: getRecord

Let us explore this method:

```
java.sql.ResultSet rs = null;
```

- This line declares and initializes a variable, rs, to reference a ResultSet object.

```
psSingleRecord.setInt(1, Integer.parseInt(partNo.trim()));
```

- This line uses the preparedStatement method, setInt, to set the value of parameter 1 to the integer value of the part number passed on the parameter list. Note the trim() method, which cuts off leading and trailing blanks from the parameter partNo. Like this, the statement does not throw an exception in case the user entered a part number with leading or trailing blanks.

```
rs = psSingleRecord.executeQuery();
```

- This line executes the SQL defined by the psSingleRecord PreparedStatement object and places the table of resulting records in a

ResultSet object referenced by rs. The method executeQuery() always returns a result set.

```
if (rs.next()) {
```

- The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. Because this is the first one read from the result set, the method positions to the first record from the result set and returns `true`. If there are no records to retrieve, the method returns a `false` value.

```
partDesc.setText(rs.getString("PARTDS").trim());
partQty.setText(Integer.toString(rs.getInt("PARTQY")));
partPrice.setText(rs.getBigDecimal("PARTPR", 2).toString());
partDate.setText(rs.getDate("PARTDT").toString());
```

- These lines retrieve values of database fields and place them in their corresponding screen fields. The ResultSet object has *getter* methods for many Java data types. Here we use the following methods:

getString	Returns the value of the column PARTDS as a String object.
getInt	Returns the value of the column PARTQY as an integer.
getBigDecimal	Returns the value of the PARTPR field as a BigDecimal object.
getDate	Returns the value of column PARTDT as a Date.

Note: In DB2 databases, it is likely that trailing blank characters are stored in the fields. In a GUI, we do not want these characters to be displayed, so the trim() method cuts them off from the database field PARTDS.

3.5.4.4 The updateRecord Method

This method is called first when the Update/Add part button is pressed. String parameters containing the values of all entry fields on the screen are passed in. These values are used to update the part record designated by the value of the parameter part number. If the record does not exist yet, the addRecord method is executed. This method does not have to return a result set. We are only interested in the fact of whether the database operation was successful. Instead of executeQuery(), the executeUpdate() method is used, which returns the number of database records affected by the SQL statement.

```
public String updateRecord(String partNo, java.awt.TextField partDesc, java.awt.TextField partQty,
                       java.awt.TextField partPrice, java.awt.TextField partDate)
{
     try
     {
            psUpdateRecord.setString(1, partDesc.getText().trim());
            psUpdateRecord.setInt(2, Integer.parseInt(partQty.getText().trim()));
            psUpdateRecord.setFloat(3, new
                       java.lang.Float(partPrice.getText().trim()).floatValue());
            psUpdateRecord.setString(4, partDate.getText().trim());
            psUpdateRecord.setInt(5, Integer.parseInt(partNo.trim()));
            int rowsUpdated = psUpdateRecord.executeUpdate();
            if (rowsUpdated > 0)
            {
                   return java.lang.String.valueOf(rowsUpdated) + " Record updated.";
            }
            else
            {
                   return addRecord(partNo, partDesc, partQty, partPrice, partDate);
            }
     }
     catch (Exception e)
     {
            e.printStackTrace();
            showException(e);
            return "Error during SQL-Execution at UPDATE.";
     }
}
```

Figure 83. The updateRecord Method Example

Class: JDBCExample Method: updateRecord

Let us break apart this method:

```
psUpdateRecord.setString(1, partDesc.getText().trim());
```

- This line uses the preparedStatement method, setString, to set the value of parameter 1 to the string value of the part description passed on the parameter list. As before, the trim() method cuts off leading and trailing blanks.

```
int rowsUpdated = psUpdateRecord.executeUpdate();
```

- This line runs the SQL update statement and returns the number of rows affected, which can be used for validation whether the update was successful. Unlike the method executeQuery(), executeUpdate does not return a result set.

```
return addRecord(partNo, partDesc, partQty, partPrice, partDate);
```

- In case that the update was not successful (rows Updated = zero), the method addRecord is called and returns its result instead of the update method.

Instead of using a prepared statement, you can create a dynamic SQL statement object to perform the same task. Performance improves if the PreparedStatement is used, assuming the update occurs more than one time during the user session. The ad hoc SQL statement appears as shown in Figure 84.

```
java.sql.Statement sUpdateRecord = dbConnect.createStatement();
String updatestring = "UPDATE PARTS SET PARTDS = '" + partDesc.getText() + "', PARTQY = " +
                      partQty.getText() + ", PARTPR = " + partPrice.getText() + ", PARTDT = '" +
                      partDate.getText() + "' WHERE PARTNO = " + partNo;
int rowsUpdated = 0;
try
{
     rowsUpdated = sUpdateRecord.executeUpdate(updatestring);
}
catch (java.sql.SQLException SQLe) {}
if (rowsUpdated > 0)
{
     return java.lang.String.valueOf(rowsUpdated) + " Record updated.";
}
```

Figure 84. The updateRecord Method Dynamic SQL Example

```
java.sql.Statement sUpdateRecord = dbConnect.createStatement();
```

- This line creates a dynamic SQL statement object called sUpdateRecord.

```
String updatestring = "UPDATE PARTS SET PARTDS = '" + partDesc.getText() + "',
PARTQY = " + partQty.getText() + ", PARTPR = " + partPrice.getText() + ",
PARTDT = '" + partDate.getText() + "' WHERE PARTNO = " + partNo;
```

- These lines build a String value for the update SQL statement. Standard SQL syntax is used to update part fields with values passed on the parameter list for the part number requested.

```
rowsUpdated = sUpdateRecord.executeUpdate(updatestring);
```

- This line runs the SQL update statement and returns the number of rows affected, which can be used for validation whether the update was successful.

3.5.4.5 The addRecord Method

This method works basically the same way as the updateRecord method, except that it uses a different preparedStatement (INSERT).

3.5.4.6 The deleteRecord Method

This method works basically the same way as the updateRecord method, except that it uses a different preparedStatement (DELETE).

3.5.4.7 The dispose Method

This method is called when the application window is closed.

```
public void dispose()
{
     try
     {
            if (psSingleRecord != null)
            {
                   psSingleRecord.close();
            }
            if (psAllRecord != null)
            {
                   psAllRecord.close();
            }
            if (psUpdateRecord != null)
            {
                   psUpdateRecord.close();
            }
            if (psAddRecord != null)
            {
                   psAddRecord.close();
            }
            if (psDeleteRecord != null)
            {
                   psDeleteRecord.close();
            }
            if (dbConnect != null)
            {
                   dbConnect.close();
            }
     }
     catch (Exception e)
     {
            System.out.println("Dispose Exception" + e);
     }
     this.getComponents();
     if (ivjJDBCExampleDisplayAll1 != null)
            ivjJDBCExampleDisplayAll1.dispose();
     super.dispose();
     System.exit(0);
     return;
}
```

Figure 85. The dispose Method Example

Class: JDBCExample Method: dispose

Let us examine this method:

```
if (psSingleRecord != null)
{
   psSingleRecord.close();
}
```

- These lines release the psSingleRecord PreparedStatement database and JDBC resources immediately. The current ResultSet is closed as well. The PreparedStatements should be tested for null, since we do not know whether the user is connected to the database (closing a PreparedStatement, which is null, throws an exception).

```
dbConnect.close();
```

- This line disconnects from the AS/400 system.

```
if (ivjDisplayAll != null) ivjDisplayAll.dispose();
```

- If the JDBCExampleDisplayAll class is still instantiated, this line calls its dispose method to shut it down.

```
super.dispose();
```

- This line calls the super class dispose method to make sure any resources used by the Frame are properly freed.

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

- This line ensures that your program shuts down properly. AS/400 Toolbox for Java connects to the AS/400 system with user threads. Because of this, a failure to issue System.exit(0) may keep your Java program from properly shutting down.

3.5.5 JDBCExampleDisplayAll Class

In this section, we investigate the key methods of the JDBCExampleDisplayAll Class.

3.5.5.1 Instance Variables

The following instance variables for accessing the database are declared for the class:

```
private java.sql.Connection dbConnect;
private java.sql.PreparedStatement psAllRecord;
```

3.5.5.2 The constructor Method

A non-default constructor is created for the class that takes parameters of a Connection object and a PreparedStatement object. This is done so that the main class (JDBCExample) can instantiate this class by passing the database objects already created (see Figure 86).

```
public JDBCExampleDisplayAll(java.sql.Connection dbc, java.sql.PreparedStatement psAll)
{
     this();
     dbConnect = dbc;
     psAllRecord = psAll;
     this.populateListBox();
     this.setVisible(true);
}
```

Figure 86. Non-Default Constructor Example

Class: JDBCExampleDisplayAll Constructor

Let us look more closely at the constructor:

```
this();
```

- This line executes the default constructor to take care of the window set up and initialization.

```
dbConnect = dbc;
psAllRecord = psAll;
```

- These lines set the instance variables for the database Connection and PreparedStatement to reference the objects passed from JDBCExample.

```
this.populateListBox();
```

- This line executes the database query and loads the records to the list box. See the method details in Figure 87.

```
this.setVisible(true);
```

- This line displays the Frame.

3.5.5.3 The populateListBox Method

This method is called from the non-default constructor. It runs the SQL statement to select all records from the parts file and converts the data in a suitable form for the GUI.

```
public void populateListBox()
{
     java.sql.ResultSet rs = null;
     try
     {
            rs = psAllRecord.executeQuery();
            while (rs.next())
            {
                   String[] array = new String[5];
                   array[0] = rs.getString("PARTNO");
                   array[1] = rs.getString("PARTDS");
                   array[2] = insertSpaces(Integer.toString(rs.getInt("PARTQY")), 5);
                   array[3] = insertSpaces(rs.getBigDecimal("PARTPR", 2).toString(), 8);
                   array[4] = rs.getDate("PARTDT").toString();
                   ivjIMulticolumnListbox1.addRow(array, array[0]);
            }
     }
     catch (Exception e)
     {
            showException(e);
     }
     return;
}
```

Figure 87. The populateListbox Method Example

Class: JDBCExampleDisplayAll Method: populateListBox

Let us examine this method:

```
rs = psAllRecord.executeQuery();
```

- This line executes the SQL defined by the psAllRecord PreparedStatement object. It places the table of resulting records in a ResultSet object referenced by rs.

```
while (rs.next()) {
```

- The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. The first time this happens, the cursor is pointed to the first record in the ResultSet and returns `true`. If there are no records to retrieve, the method returns a `false` value. The method loops until the next() method returns a `false` value.

```
String[] array = new String[5];
```

- This line creates a new string array object, which receives the data from the current record of the ResultSet.

```
array[0] = rs.getString("PARTNO");
array[1] = rs.getString("PARTDS");
array[2] = insertSpaces(Integer.toString(rs.getInt("PARTQY")), 5);
array[3] = insertSpaces(rs.getBigDecimal("PARTPR", 2).toString(), 8);
array[4] = rs.getDate("PARTDT").toString();
```

- These lines retrieve each of the field values from the current record in the ResultSet. The values are converted to strings and placed into the string array. Note the insertSpaces(String, int) method, which is responsible for lining up numeric columns correctly. For the same purpose, you also have to choose a monospace font for the dataFont property of the list box.

```
ivjIMulticolumnListbox1.addRow(array, array[0]);
```

- This line adds the string values of the current parts record as a new row at the end of the list box. The list box places each element of the array in its corresponding column and keeps a key for the row, which is part number in this case.

3.5.6 Reusable GUI Part

For the remainder of the database examples in this chapter, a reusable class is used to handle the user interface. The advantage is that the user interface is designed, programmed, and tested once. Then, it is re-used in multiple applications that demonstrate different methods of accessing resources on the AS/400 system.

The class is called *ToolboxGUI*. It is a subclass of java.awt.Panel and can be dropped onto a Frame. ToolboxGUI communicates with its parent container through a PartsContainer interface. This interface allows specific methods of the parent to be invoked by the ToolboxGUI class to handle functions such as connecting to the database, retrieving, updating or deleting a part record, or adding part records to a list box.

To use the ToolboxGUI, create a new class that is a sub-class of java.awt.Frame, and implement the PartsContainer interface. In the Visual Composition Editor, choose the bean ToolboxGUI, drop it on the empty application frame, re-size and re-position to fit.

The PartsContainer interface designates methods to be implemented in the main application class so that ToolboxGUI can make requests for database access. The interface methods are:

connectToDB	Connects to the database server and returns a String result.
getRecord	Retrieves a single record from the database and places the resulting record field values in the TextFields passed.
populateListBox	Retrieves all records from the database and adds values for each record in the list box widget passed.
updateRecord	Updates or adds the database record with the values passed; returns a String result.
deleteRecord	Deletes a single record from the database; returns a String result.

ToolboxGUI calls out to the parent method using the following format:

```
((PartsContainer)getParent()).connectToDB(systemName, userid, password);
```

The getParent() method returns the ToolboxGUIs container object. This object is cast as an object that conforms to the PartsContainer interface. The connectToDB method is invoked on the parent object. Similar code is used for the other interface methods.

The ToolboxGUI also has a helper class, DisplayAllParts, to display and populate the IMulticolumnListbox. This class is instantiated when the Get All Parts button is pressed. It uses the same mechanism defined previously to call out to the parents populateListBox method.

3.5.7 Stored Procedures

Using stored procedures with the Toolbox is an extension of the JDBC access technique. Instead of using PreparedStatement and Statement objects to execute SQL statements, a CallableStatement object is defined and executed.

The prepareCall method on the Connection object is used to create a CallableStatement object, for example:

```
CallableStatement aCS = aConnection.prepareCall(
         "CALL LibraryName.PocedureName(?, ?, ?)");
```

These lines define a CallableStatement object, aCS. When executed, aCS calls the procedure in the specified library, passing three parameters. These parameters can be input, output, or both. Output parameters must be registered using the registerOutParameter method, for example:

```
aCS.registerOutParameter (3, java.sql.Types.INTEGER);
```

This line registers the third parameter (the third question mark) as an output parameter for the stored procedure of an SQL type integer. After the procedure is executed, the value of the parameter can be retrieved using aCS.getInt(3). Other *getters* (= methodname starting with get) exist for each registered data type.

Input parameters must be set using the set method associated with the data type, for example:

```
aCS.setInt(1, 500);
```

This line sets the value of the first parameter to an integer value of 500.

Stored procedures can be executed using the execute, executeQuery, or executeUpdate methods. The execute method is used when zero or more result sets are expected to be returned. The executeQuery method can be used when exactly one result set is returned. The executeUpdate can be used when no result set but a number of rows affected is to be returned for database operations such as UPDATE, INSERT, or DELETE.

Stored procedures are generally used for two reasons. First, native programs written in RPG, COBOL, and others can be used by the Java application through a standard interface. Second, stored procedures can greatly boost performance of the application when compared with straight SQL.

3.5.8 JDBC Stored Procedure Application Example

In the following example, we use JDBC stored procedures to access records in an AS/400 database (see Figure 88).

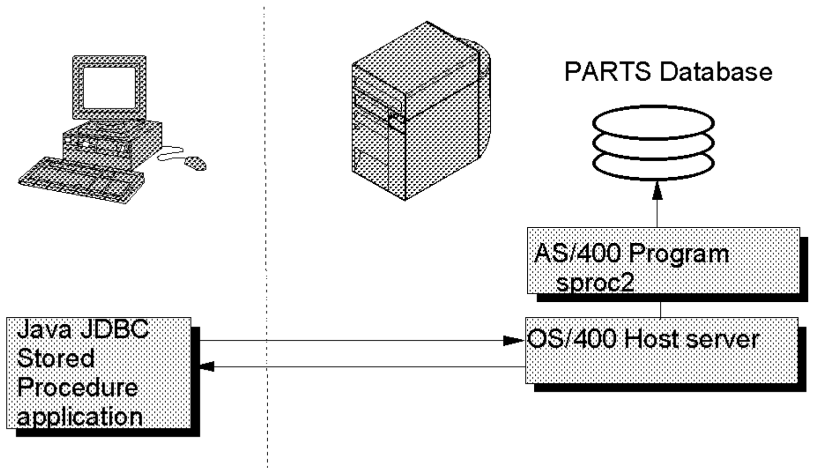
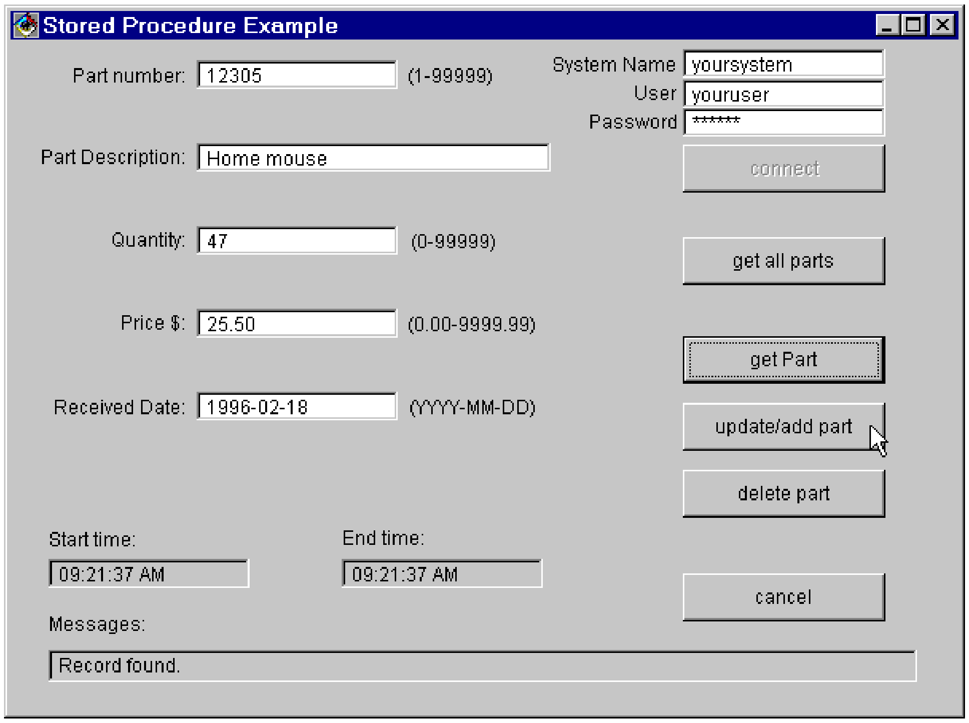


Figure 88. JDBC Application Stored Procedures

The client program requests data from the AS/400 database by calling an AS/400 stored procedure program. The host server passes the call to the AS/400 program and returns the results to the client program in an SQL result set. The JDBC support handles all data conversions. Figure 89 and Figure 90 on page 129 show the user interface to the Stored Procedure.


Stored Procedure Example

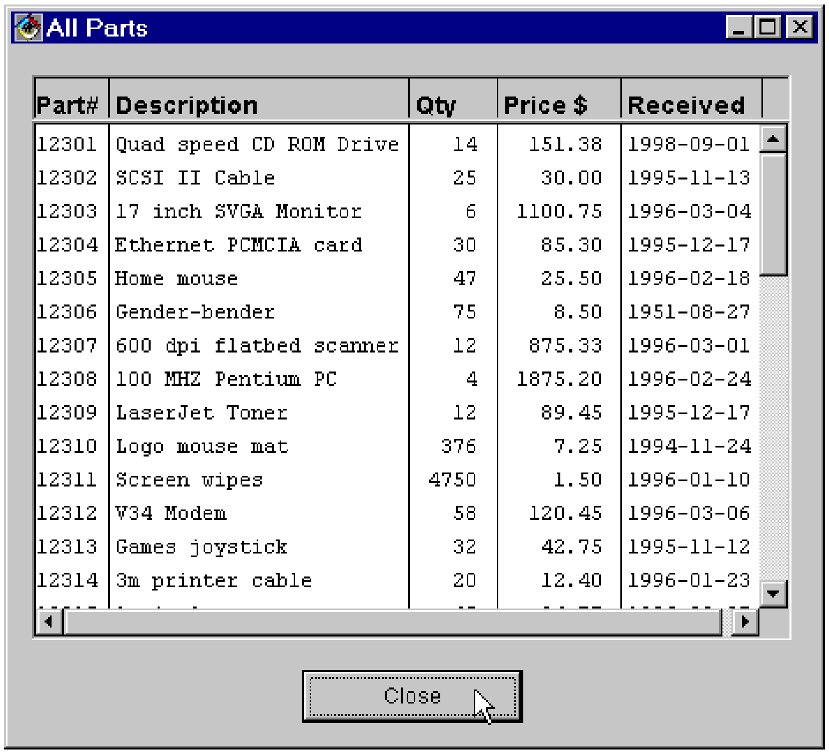
Part number:	12305	(1-99999)
Part Description:	Home mouse	
Quantity:	47	(0-99999)
Price $:	25.50	(0.00-9999.99)
Received Date:	1996-02-18	(YYYY-MM-DD)

System Name yoursystem
User youruser
Password ******

Start time: 09:21:37 AM
End time: 09:21:37 AM

Messages: Record found.

Figure 89. Stored Procedure Example One Part


All Parts

Part#	Description	Qty	Price $	Received
12301	Quad speed CD ROM Drive	14	151.38	1998-09-01
12302	SCSI II Cable	25	30.00	1995-11-13
12303	17 inch SVGA Monitor	6	1100.75	1996-03-04
12304	Ethernet PCMCIA card	30	85.30	1995-12-17
12305	Home mouse	47	25.50	1996-02-18
12306	Gender-bender	75	8.50	1951-08-27
12307	600 dpi flatbed scanner	12	875.33	1996-03-01
12308	100 MHZ Pentium PC	4	1875.20	1996-02-24
12309	LaserJet Toner	12	89.45	1995-12-17
12310	Logo mouse mat	376	7.25	1994-11-24
12311	Screen wipes	4750	1.50	1996-01-10
12312	V34 Modem	58	120.45	1996-03-06
12313	Games joystick	32	42.75	1995-11-12
12314	3m printer cable	20	12.40	1996-01-23

Figure 90. Stored Procedure Example All Parts

Class StoredProcedureExample is the main class in this application. It is functionally equivalent to the JDBCExample application, but is implemented using different techniques. The ToolboxGUI class is used to handle all user interaction. A stored procedure is used instead of SQL statements. Record read, update, add, and delete are implemented in this example. This was done by creating two stored procedures that perform the corresponding database operations. The reason for creating two stored procedures was due to the different number of parameters needed for the different database operations.

The programs we use as stored procedures are written in RPG and named SPROC2 and SPROC3 in library APILIB. The first program SPROC2 takes two integer input parameters. Parameter 1 is an action code. A value of 1 returns a single record in the result set with the part number field matching the part number supplied in the second parameter. A value of 2 in the first parameter returns all records from the parts database in a result set. The second parameter is ignored in this case. A value of 3 deletes a single record in the database file with the part number field matching the part number supplied in the second parameter. The second program SPROC3 takes three integers, one string, one float, and one date input parameter. Parameter 1 is an action code. A value of 1 causes a single record defined by the part number supplied in the second parameter to be updated. Parameters three through six supply the values for the corresponding database fields. A value of 2 causes a single record defined by the part number supplied in the second parameter to be written into the database file. Parameters three through six supply the values for the corresponding database fields.

3.5.9 StoredProcedureExample Class

In this section, we investigate the key methods of the StoredProcedureExample Class. SQL statements are written in capital letters for better readability. They can also be written in lowercase.

3.5.9.1 Instance Variables

The following instance variables for accessing the database are declared for the class:

```
private java.sql.CallableStatement callableStmt;
private java.sql.CallableStatement callableStmt1;
private java.sql.Connection dbConnect;
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
```

3.5.9.2 The connectToDB Method

The connectToDB method is called by the ToolboxGUI class when the Connect button is pressed. String parameters representing the AS/400 system name, user ID, and password are passed to the method.

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
     java.sql.DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
     dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName +
             "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
             "package=TeamLab;package library=apilib", userid, password);
     try
     {
             dbConnect.createStatement().execute("DROP PROCEDURE APILIB.PARTQRY2");
     }
     catch (Exception e){}
     try
     {
             dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY2(INOUT P1" +
                     " INT, INOUT P2 INT) RESULT SETS 1 LANGUAGE RPG DETERMINISTIC CONTAINS SQL" +
                     " EXTERNAL NAME APILIB.SPROC2 PARAMETER STYLE GENERAL");
     }
     catch (Exception e){}
     callableStmt = dbConnect.prepareCall("CALL APILIB.PARTQRY2(?, ?)");
     try
     {
             dbConnect.createStatement().execute("DROP PROCEDURE APILIB.PARTQRY3");
     }
     catch (Exception e){}
     try
     {
             dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY3(INOUT P1" +
                      " INT, IN P2 INT , IN P3 CHAR (25), IN P4 INT , IN P5 DEC (6, 2), IN P6 DATE)" +
                      " LANGUAGE RPG NOT DETERMINISTIC CONTAINS SQL EXTERNAL NAME APILIB.SPROC3" +
                      " PARAMETER STYLE GENERAL");
     }
     catch (Exception e){}
     callableStmt1 = dbConnect.prepareCall("CALL APILIB.PARTQRY3(?, ?, ?, ?, ?, ?)");
     return;
}
```

Figure 91. Stored Procedure Example connectToDB Method

Let us dissect this method:

```
java.sql.DriverManager.registerDriver(new
   com.ibm.as400.access.AS400JDBCDriver());
dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName
   + "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
   "package=TeamLab;package library=apilib", userid, password);
```

- Loads the JDBC driver and connects to the AS/400 system the same way as in the preceding JDBCExample class.

```
dbConnect.createStatement().execute("DROP PROCEDURE APILIB.PARTQRY2");
```

- Attempts to remove the stored procedure PARTQRY2 from the system catalog, if it exists. If the procedure does not already exist in the catalog, an error is thrown, so we catch it and do nothing.

```
dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY2(INOUT
   P1" + " INT, INOUT P2 INT) RESULT SETS 1 LANGUAGE RPG DETERMINISTIC CONTAINS
   SQL" +" EXTERNAL NAME APILIB.SPROC2 PARAMETER STYLE GENERAL");
```

- This command executes an SQL statement to add the PARTQRY2 procedure to the system catalog. A new statement object is created by the Connection object. The execute method is used to run an ad hoc SQL statement to declare the RPG program SPROC2 to the catalog. In a production environment, the procedure is added to the catalog once by a system administrator and not added on the fly by an application each time it connects to the database.

Attention

We show how to drop and create an AS/400 stored procedure from a Java client here. In most cases, it is better to do this directly on the AS/400 system. You can do this on the AS/400 system by using interactive SQL or through an application program. Creating the stored procedure needs to be done only once. It is added to the system catalog, so it can be found and reused. Creating a stored procedure from the client, as shown here, adds extra overhead to a Java application.

```
callableStmt = dbConnect.prepareCall("CALL APILIB.PARTQRY2(?, ?)");
```

- This command creates a new CallableStatement object from the Connection object. The statement declares the stored procedure and has markers for two parameters. The parameters are input only, because no output parameters are registered.

3.5.9.3 The getRecord Method

The GetRecord method is called by the ToolboxGUI class when the Get Part button is pressed.

```
public String getRecord(String partNo, java.awt.TextField partDesc, java.awt.TextField partQty,
java.awt.TextField partPrice, java.awt.TextField partDate) throws Exception
{
     java.sql.ResultSet rs = null;
     callableStmt.setInt(1, 1);
     callableStmt.setInt(2, Integer.parseInt(partNo));
     rs = callableStmt.executeQuery();
     if (rs.next())
     {
            partDesc.setText(rs.getString(2).trim());
            partQty.setText(Integer.toString(rs.getInt(3)));
            partPrice.setText(rs.getBigDecimal(4, 2).toString());
            partDate.setText(rs.getDate(5).toString());
     }
     else
     {
            partDesc.setText("");
            partQty.setText("0");
            partPrice.setText("0.00");
            partDate.setText("");
            return "Record not found.";
     }
     return "Record found.";
}
```

Figure 92. Stored Procedure Example getRecord Method

Class: StoredProcedureExample Method: getRecord

Method highlights:

```
java.sql.ResultSet rs = null;
```

- Declares a variable, rs, to reference a ResultSet object.

```
callableStmt.setInt(1, 1);
callableStmt.setInt(2, Integer.parseInt(partNo.trim()));
```

- Uses the setInt method to set the value of parameter 1 to the integer value 1 to tell the program to get a single part record. Then, the setInt method is used to set the value of parameter 2 to the integer value of the part number passed on the parameter list.

```
rs = callableStmt.executeQuery();
```

- Executes the stored procedure defined by the callableStatement object and places the table of resulting records in a ResultSet object referenced by rs.

```
if (rs.next()) {
```

- The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. Because this is the first read from the result set, the method positions to the first record from the result set and returns a `true` value. If there are no records to retrieve, the method returns a `false` value.

```
partDesc.setText(rs.getString(2).trim());
partQty.setText(Integer.toString(rs.getInt(3)));
partPrice.setText(rs.getBigDecimal(4, 2).toString());
partDate.setText(rs.getDate(5).toString());
```

- These lines retrieve values of database fields and place them in their corresponding screen fields. The ResultSet object has getter methods for many Java data types. We use column indices instead of column names to reference the values requested from the result set.

3.5.9.4 The populateListBox Method

This method is called from the DisplayAllParts non-default constructor through the PartsContainer Interface. It runs the SQL statement to select all records from the parts file.

```
public void populateListBox(com.ibm.ivj.eab.dab.IMulticolumnListbox aListBox) throws Exception
{
     java.sql.ResultSet rs = null;
     callableStmt.setInt(1, 2);
     callableStmt.setInt(2, 0);
     rs = callableStmt.executeQuery();
     while (rs.next())
     {
            String[] array = new String[5];
            array[0] = rs.getString(1);
            array[1] = rs.getString(2);
            array[2] = WorkShop.DisplayAllParts.insertSpaces(Integer.toString(rs.getInt(3)), 5);
            array[3] = WorkShop.DisplayAllParts.insertSpaces((rs.getBigDecimal(4,
                      2).toString()),8);
            array[4] = rs.getDate(5).toString();
            aListBox.addRow(array, array[0]);
     }
     return;
}
```

Figure 93. Stored Procedure Example populateListBox Method

Class: StoredProcedureExample Method: populateListBox

The populateListBox method highlights:

```
callableStmt.setInt(1, 2);
callableStmt.setInt(2, 0);
```

- Uses the setInt method to set the value of parameter 1 to the integer value 2 to tell the program to get all part records. Then, the setInt method is used to set the value of parameter 2 to the integer value of 0 so that a null value is not passed to the procedure.

```
rs = callableStmt.executeQuery();
```

- Executes the stored procedure defined by the CallableStatement object, and places the table of resulting records in a ResultSet object referenced by rs.

```
while (rs.next()) {
```

- The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. The first time the cursor is pointed to the first record in the ResultSet and returns a `true` value. If there are no records to retrieve, the method returns a `false` value. The method loops until next() returns a false value.

```
array[0] = rs.getString(1);
array[1] = rs.getString(2);
array[2] =
   WorkShop.DisplayAllParts.insertSpaces(Integer.toString(rs.getInt(3)),5);
array[3] = WorkShop.DisplayAllParts.insertSpaces((rs.getBigDecimal(4,
   2).toString()),8);
array[4] = rs.getDate(5).toString();
```

- Retrieves the field values from the current record in the ResultSet. These values are converted to strings and placed into a string array for adding to the multi-column list box. The list box places each element of the array in a different column of the list box. Column indices are used here instead of column names.

```
aListBox.addRow(array, array[0]);
```

- Adds the String values of the current parts record as a new row at the end of the list box. The list box places each element of the array in its corresponding column and keeps a key for the row, which is part number in this case.

3.5.9.5 The dispose Method

The dispose method is called when the application window is closed.

```
public void dispose()
{
     try
     {
            if (callableStmt != null)
            {
                   callableStmt.close();
            }
            if (callableStmt1 != null)
            {
                   callableStmt1.close();
            }
            if (dbConnect != null)
            {
                   dbConnect.close();
            }
     }
     catch (Exception e)
     {
            System.out.println("Exception while disconnecting from AS/400." + e.toString());
     }
     super.dispose();
     System.exit(0);
     return;
}
```

Figure 94. Stored Procedure Example dispose Method

Class: StoredProcedureExample Method: dispose

The dispose method highlights include:

`callableStmt.close();`

- Releases the CallableStatements database and JDBC resources immediately. This also closes the current ResultSet. The CallableStatements should be tested for null, since we do not know whether the user has connected to the database. Closing a CallableStatement, which is null, throws an exception.

`dbConnect.close();`

- Disconnects from the AS/400 system.

`super.dispose();`

- Calls the super class dispose method to make sure any resources used by the frame are properly freed.

`System.exit(0);`

- This line ensures that your program shuts down properly. AS/400 Toolbox for Java connects to the AS/400 with user threads. Because of this, a failure to issue System.exit(0) may keep your Java program from properly shutting down.

3.5.10 DDM Record-Level Access Application Example

In the example in Figure 95, we use Distributed Data Management (DDM) Record Level Access to access records in an AS/400 database named Parts.

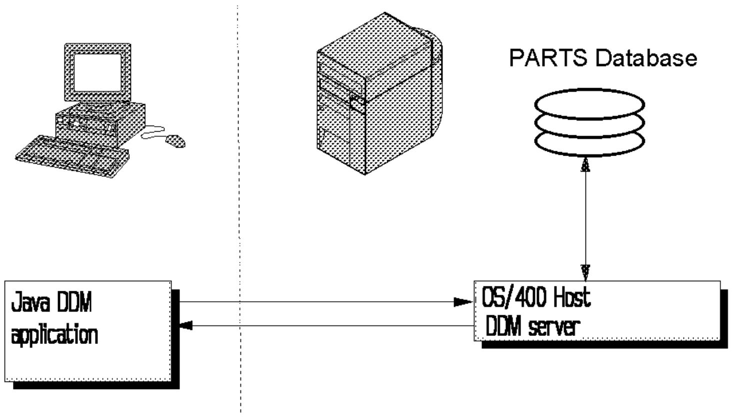
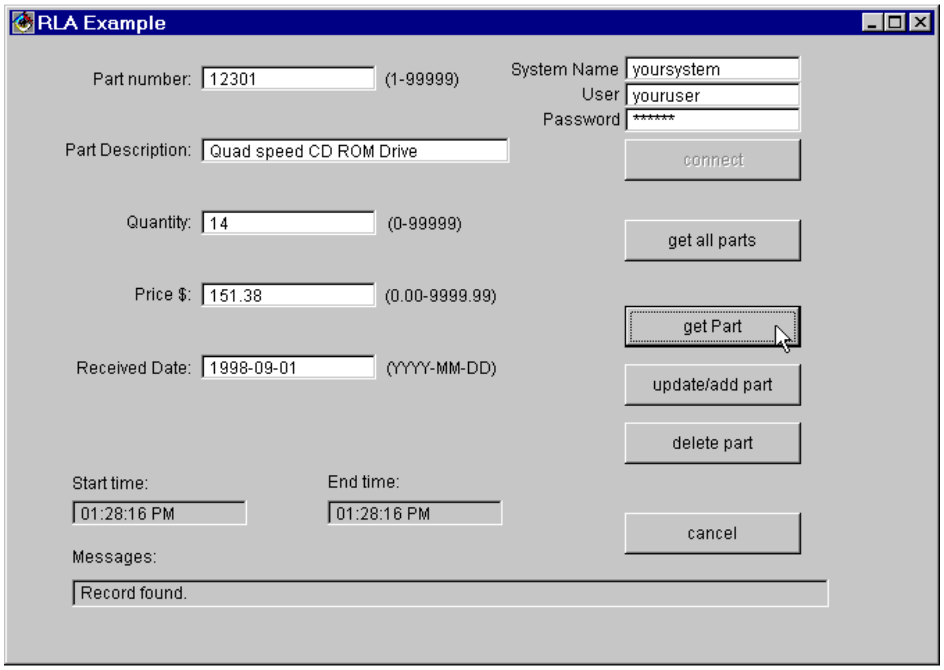


Figure 95. DDM Record Level Access

The client program requests data from the AS/400 database by interfacing with the host DDM server. The DDM server accesses the database and returns the results to the client program. We demonstrate using the DDM server to retrieve the format of the Parts file from the AS/400 system. This makes it easy to work with the file using field names.



Part number:	12301	(1-99999)
Part Description:	Quad speed CD ROM Drive	
Quantity:	14	(0-99999)
Price $:	151.38	(0.00-9999.99)
Received Date:	1998-09-01	(YYYY-MM-DD)
System Name	yoursystem	
User	youruser	
Password	******	
Start time:	01:28:16 PM	
End time:	01:28:16 PM	
Messages:	Record found.	

Figure 96. Distributed Data Management Record Level Access Example

Class RLAExample is the main class in this application. We use the same classes as in the other examples, and ToolboxGUI and DisplayAll Parts to handle all user interaction.

3.5.11 RLAExample Class

In this section, we investigate the key methods of the RLAExample Class.

3.5.11.1 Instance Variables

The following instance variables for accessing the database are declared for the class:

```
private AS400 as400;
private KeyedFile myKeyedFile;
private SequentialFile mySeqFile;
private RecordFormat partsFormat = null;
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
```

3.5.11.2 The connectToDB Method

The ConnectToDB method is called by the ToolboxGUI class when the Connect button is pressed.

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
     as400 = new AS400(systemName, userid, password);
     QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB", "PARTS", "*FILE", "MBR");
     myKeyedFile = new KeyedFile(as400, fileName.getPath());
     mySeqFile = new SequentialFile(as400, fileName.getPath());
     try
     {
             as400.connectService(AS400.RECORDACCESS);
     }
     catch (Exception e)
     {
             System.out.println("Unable to connect");
             System.exit(0);
     }
     try
     {
             AS400FileRecordDescription recordDescription = new AS400FileRecordDescription(as400,
                        "/QSYS.LIB/APILIB.LIB/PARTS.FILE");
             partsFormat = recordDescription.retrieveRecordFormat()[0];
             partsFormat.addKeyFieldDescription("PARTNO");
     }
     catch (Exception e)
     {
             System.out.println("Unable to retrieve record format from APILIB/PARTS");
             System.exit(0);
     }
     try
     {
             myKeyedFile.setRecordFormat(partsFormat);
             mySeqFile.setRecordFormat(partsFormat);
             myKeyedFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
             mySeqFile.open(AS400File.READ_ONLY, 100, AS400File.COMMIT_LOCK_LEVEL_NONE);
     }
     catch (Exception e)
     {
             System.out.println("Unable to open file");
             System.exit(0);
     }
     return;
}
```

Figure 97. Record Level Access Example connectToDB Method

Let us dissect the method:

```
as400 = new AS400(systemName, userid, password);
```

- Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

```
QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB", "PARTS",
   "*FILE", "MBR");
```

- Creates a new path name for the file to be accessed.

```
myKeyedFile = new KeyedFile(as400, fileName.getPath());
mySeqFile = new SequentialFile(as400, fileName.getPath());
```

- Creates a keyed file object and a sequential file object that represents the file we access on the AS/400 system. We use the QSYSObectPathName object to get the path and name of the file into the correct format. The keyed file is used for all single-record operations such as add, update, delete, and read. The sequential file is used for the multiple-record operation in the populateListBox method.

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

- Connects to the AS/400 DDM server. This is not required. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method as opposed to connecting the first time the user requests a part record.

```
AS400FileRecordDescription recordDescription = new
   AS400FileRecordDescription(as400, "/QSYS.LIB/APILIB.LIB/PARTS.FILE");
```

- Creates the Record Description object for accessing the file. The record description is the same as the record format for file APILIB/PARTS.

```
partsFormat = recordDescription.retrieveRecordFormat()[0];
```

- We retrieve the record format. There is only one record format for the file, so we use the first (and only) element of the RecordFormat array returned as the record format for the file.

```
partsFormat.addKeyFieldDescription("PARTNO");
```

- We make the PARTNO field the key field.

```
myKeyedFile.setRecordFormat(partsFormat);
mySeqFile.setRecordFormat(partsFormat);
```

- We set the record format with the format description that we retrieved from the AS/400 system.

```
myKeyedFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
mySeqFile.open(AS400File.READ_ONLY, 100, AS400File.COMMIT_LOCK_LEVEL_NONE);
```

- We open the keyed file for both read and write. Since we are not opening for read only, a blocking factor (0) is ignored and no blocking is done. If we are reading records only, as in the sequential file, we can specify a blocking factor (100) on the open to help achieve better performance. We are not using commitment control.

3.5.11.3 The getRecord Method

The getRecord method is called by the ToolboxGUI class when the Get Part button is pressed.

```
public String getRecord(String partNo, java.awt.TextField partDesc, java.awt.TextField partQty,
                     java.awt.TextField partPrice, java.awt.TextField partDate) throws Exception
{
     Object[] theKey = new Object[1];
     theKey[0] = new java.math.BigDecimal(partNo.trim());
     Record data = myKeyedFile.read(theKey);
     if (data != null)
     {
            partDesc.setText(((String) data.getField("PARTDS")).trim());
            partQty.setText(((java.math.BigDecimal) data.getField("PARTQY")).toString());
            partPrice.setText(((java.math.BigDecimal) data.getField("PARTPR")).toString());
            partDate.setText((String) data.getField("PARTDT"));
            return "Record found.";
     }
     else
     {
            partDesc.setText("");
            partQty.setText("0");
            partPrice.setText("0.00");
            partDate.setText("");
            return "Record not found.";
     }
}
```

Figure 98. Record Level Access Example getRecord Method

The getRecord method highlights include:

```
Object[] theKey = new Object[1];
theKey[0] = new java.math.BigDecimal(partNo.trim());
```

- Creates the key for reading the records. The key for a keyed file is specified as an object array.

```
Record data = myKeyedFile.read(theKey);
```

- Reads the first record matching the key. Null is returned if the record is not found.

```
partDesc.setText(((String) data.getField("PARTDS")).trim());
partQty.setText(((java.math.BigDecimal) data.getField("PARTQY")).toString());
partPrice.setText(((java.math.BigDecimal)
   data.getField("PARTPR")).toString());
partDate.setText((String) data.getField("PARTDT"));
return "Record found.";
```

- If the record is found, we use the field names to retrieve the data and set the text property of the objects shown on the ToolboxGUI. This causes the text fields to display the values received in this method.

3.5.11.4 The populateListBox Method

The populateListBox method is called from the DisplayAllParts non-default constructor through the PartsContainer Interface. It uses the sequential file and requests all records from the parts file.

```
public void populateListBox(com.ibm.ivj.eab.dab.IMulticolumnListbox aListBox) throws Exception
{
     try
     {
            Record record = mySeqFile.readFirst();
            while (record != null)
            {
                   String[] array = new String[5];
                   array[0] = ((java.math.BigDecimal) record.getField("PARTNO")).toString();
                   array[1] = (String) record.getField("PARTDS");
                   array[2] = WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                      record.getField("PARTQY")).toString(), 5);
                   array[3] = WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                      record.getField("PARTPR")).toString(), 8);
                   array[4] = (String) record.getField("PARTDT");
                   aListBox.addRow(array, array[0]);
                   record = mySeqFile.readNext();
            }
     }
     catch (Exception e)
     {
            System.out.println("unable to get all");
            System.exit(0);
     }
     return;
}
```

Figure 99. Record Level Access Example populateListBox Method

The populateListBox method highlights include:

```
Record record = mySeqFile.readFirst();
```

- We use the readFirst method to read the first record.

```
array[0] = ((java.math.BigDecimal) record.getField("PARTNO")).toString();
array[1] = (String) record.getField("PARTDS");
array[2] = WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal)
   record.getField("PARTQY")).toString(), 5);
array[3] = WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal)
   record.getField("PARTPR")).toString(), 8);
array[4] = (String) record.getField("PARTDT");
```

- We use the field names to retrieve the data fields and move them to the array elements.

```
aListBox.addRow(array);
```

- We use the addRow method to add a new row to the list box.

```
record = mySeqFile.readNext();
```

- We read the next record in the file.

Note: When using a sequential file, the records are not sorted by the key defined in the database file. Use a keyed file if you want to obtain sorted records.

3.5.11.5 The dispose Method

The dispose method is called when the application window is closed.

```
public void dispose()
{
     try
     {
            if (myKeyedFile != null)
            {
                   myKeyedFile.close();
            }
            if (mySeqFile != null)
            {
                   mySeqFile.close();
            }
            if (as400 != null)
            {
                   as400.disconnectAllServices();
            }
     }
     catch (Exception e)
     {
            System.out.println("Exception while disconnecting from AS/400." + e.toString());
     }
     super.dispose();
     System.exit(0);
     return;
}
```

Figure 100. Record Level Access Example dispose Method

Class: RLAExample Method: dispose

The dispose method supports these features:

`myKeyedFile.close();`

- Closes the open database file.

`as400.disconnectAllServices();`

- Releases all connections to the AS/400 system and releases resources associated with server jobs processing requests for the client.

`super.dispose();`

- Calls the super class dispose method to make sure any resources used by the frame are properly freed.

All other methods used in the dispose method are the same as in the JDBC and StoredProcedure examples.

3.5.12 Distributed Program Call Feature

The Program Call feature of the AS/400 Toolbox allows a Java program to directly execute any non-interactive program object (*PGM) on the AS/400 system. It passes input data as parameters and returns results through parameters.

The Java developer must use the data conversion classes from the Toolbox to convert input parameters from the Java format to an AS/400 data type and convert output parameters from AS/400 format to a Java format.

The advantage of using the Distributed Program Call class is that native AS/400 non-interactive programs can be executed from a Java application unchanged. Native program calls can also result in better performance of a Java application when compared with JDBC. In addition, this interface can call programs on the AS/400 system that do more than just database access. For example, a Java application can call a program that starts nightly job processing, saves libraries to tape, or sends or receives data through communication lines.

Calling a native AS/400 program involves the following steps:

1. Connect to the AS/400 system by creating an AS400 object.
2. Create a ProgramCall object.
3. Define and initialize a ProgramParameter array for passing parameters to and from the called program.
4. Use the Data Conversion classes to convert input parameter values from the Java format to the AS/400 format.
5. Use the setProgram method to specify the qualified name of the program to call and the parameters to use, if not declared on the ProgramCall constructor.
6. Execute the program using the run method.
7. If the run method fails, obtain detailed error information through AS400Message objects.
8. Retrieve output parameters using the getOutputData method of the ProgramParameter object.
9. Convert output parameter values using the data conversion classes.

3.5.13 Distributed Program Call (DPC) Application Example

In this example, we use the Distributed Program Call (DPC) interface to allow a client program to call an AS/400 program (see Figure 101 on page 145). We also develop this application using the Enterprise Toolkit/400 Program Call SmartGuide. Please see Section 9.4, “Creating a Program Call JavaBean” on page 345, for details.

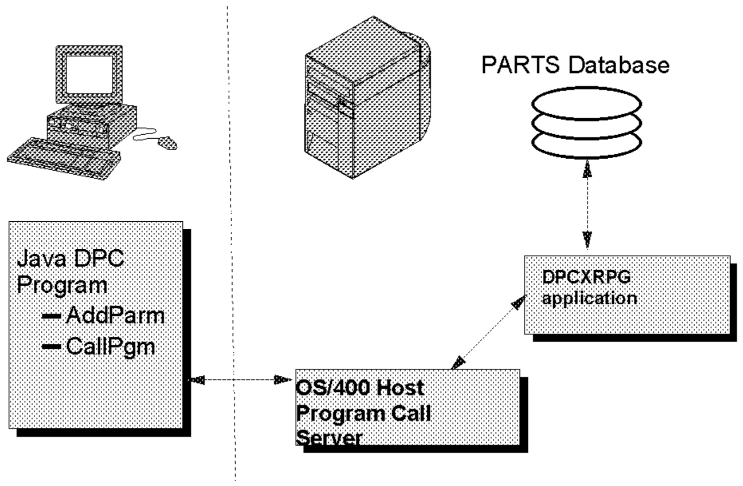
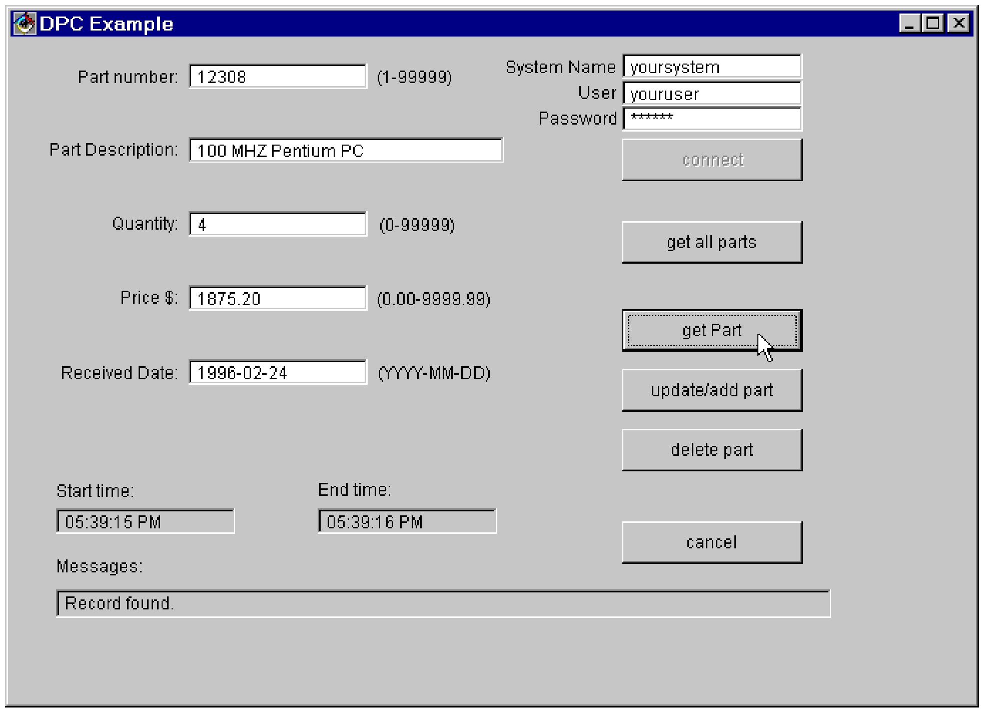


Figure 101. Distributed Program Call Example

The client program requests data from the AS/400 database by calling an AS/400 program. Information is passed between the programs using parameters. It is up to the application implementer to handle data conversions.



Part number: 12308 (1-99999)
System Name yoursystem
User youruser
Password ******
Part Description: 100 MHZ Pentium PC
Quantity: 4 (0-99999)
Price $: 1875.20 (0.00-9999.99)
Received Date: 1996-02-24 (YYYY-MM-DD)
Start time: 05:39:15 PM
End time: 05:39:16 PM
Messages: Record found.

Figure 102. Distributed Program Call Example

The client program requests data from the server program by calling it and passing it parameters. The input parameters are a flag and a part number and all of the attributes of a part. For example, S12301 is a request for a single record (Flag = S) of part number 12301. If requesting all parts (Flag=A), the part number is not necessary. The server program, DPCXRPG, searches the database for the requested information. The result is passed back in the output parameters.

Class DPCExample is the main class in this application. It is functionally equivalent to the StoredProcedureExample application, but is implemented using different techniques. The ToolboxGUI and DisplayAllParts class are used to handle all user interaction. A native RPG program is called on the AS/400 system to access and return data. All of the four basic database operations are implemented in this example.

3.5.13.1 RPG Program Background

Library: APILIB

Program Name: DPCXRPG

Parameters (all are used as Input/Output):

Table 23. Parameter List

Sequence / Field	Description	Length/Type
1 / OPTION	In: Operation Code / Out: Return Code	1 character
2 / PARTNO	Part Number	5.0 packed
3 / PARTDS	Part Description	25 character
4 / PARTQY	Part Quantity	5.0 packed
5 / PARTPR	Part Price	6.2 packed
6 / PARTDT	Part Date Received	10 date

Values of the Operation Code (Input OPTION):

Table 24. Flag Operation Codes

Operation Code	Database Operation to Execute
S	Retrieve a single record for the supplied key.
A	Retrieve all records.
F	Fetch next record based on the current position.
E	End the program.
D	Delete a single record for the supplied key.
U	Update a single record for the supplied key with the attribute data; Write a single record for the supplied key with the attribute data if it does not yet exist.

Values of the Return Code (Output OPTION):

Table 25. Flag Operation Codes

Return Code	Result Description
Y	Normal: Operation has succeeded / When operation code was U: Record updated
X	Normal: Operation has failed / When operation code was U: Record added
U	Unknown operation code has been supplied

3.5.14 DPCExample Class

In this section, we investigate the key methods of the DPCExample class.

3.5.14.1 Instance Variables

The following instance variables are declared for the class:

```
private AS400 as400;
private ProgramCall pgm;
private String progName = "/QSYS.LIB/apilib.LIB/DPCXRPG.PGM";
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
```

3.5.14.2 The connectToDB Method

The connectToDB method is called by the ToolboxGUI class when the Connect button is pressed. String parameters representing the AS/400 system name, user ID, and password are passed to the method.

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
     as400 = new AS400(systemName, userid, password);
     as400.connectService(AS400.COMMAND);
     pgm = new ProgramCall(as400);
     return;
}
```

Figure 103. Distributed Program Call Example connectToDB Method

Class: DPCExample Method: connectToDB

The connectToDB method offers these functions:

```
as400 = new AS400(systemName, userid, password);
```

- Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

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

- Connects to the AS/400 program call and command call server. This is not required. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method as opposed to connecting the first time the user requests a part record.

```
pgm = new ProgramCall(as400);
```

- Creates a new ProgramCall object for the AS/400 system defined in the AS400 object. The program and parameter information are supplied later.

3.5.14.3 The getRecord Method

The getRecord method is called by the ToolboxGUI class when the Get Part button is pressed.

```
public String getRecord(String partNo, java.awt.TextField partDesc, java.awt.TextField partQty,
java.awt.TextField partPrice, java.awt.TextField partDate) throws Exception
{
     ProgramParameter[] parmlist = new ProgramParameter[6];
     AS400Text asFlag = new AS400Text(1);
     parmlist[0] = new ProgramParameter(asFlag.toBytes("S"), 1);
     AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
     parmlist[1] = new ProgramParameter(asPartNo.toBytes(new java.math.BigDecimal(partNo.trim())),
                      3);
     AS400Text asDesc = new AS400Text(25);
     parmlist[2] = new ProgramParameter(asDesc.toBytes(""), 25);
     AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
     parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(0)), 3);
     AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
     parmlist[4] = new ProgramParameter(asPrice.toBytes(new java.math.BigDecimal(0)), 4);
     AS400Text asDate = new AS400Text(10);
     parmlist[5] = new ProgramParameter(asDate.toBytes("0001-01-01"), 10);
     pgm.setProgram(progName, parmlist);
     if (pgm.run() != true)
     {
             System.out.println("program failed:" + progName);
             AS400Message[] messagelist = pgm.getMessageList();
             for (int i = 0; i < messagelist.length; i++)
             {
                     System.out.println(messagelist[i]);
             }
             return "Program call failed!";
     }
     else
     {
             if (((String) (asFlag.toObject(parmlist[0].getOutputData(), 0))).equals("Y"))
             {
                     partDesc.setText(((String) (new
                        AS400Text(25)).toObject(parmlist[2].getOutputData(), 0)).trim());
                     partQty.setText(((java.math.BigDecimal) (new AS400PackedDecimal(5,
                        0)).toObject(parmlist[3].getOutputData(), 0)).toString());
                     partPrice.setText(((java.math.BigDecimal) (new AS400PackedDecimal(6,
                        2)).toObject(parmlist[4].getOutputData(), 0)).toString());
                     partDate.setText((String) (new
                        AS400Text(10)).toObject(parmlist[5].getOutputData(), 0));
                     return "Record found.";
             }
             else
             {
                     partDesc.setText("");
                     partQty.setText("0");
                     partPrice.setText("0.00");
                     partDate.setText("");
                     return "Record not found.";
             }
     }
}
```

Figure 104. Distributed Program Call Example getRecord Method

Class: DPCExample Method: getRecord

The getRecord method supports these features:

```
ProgramParameter[] parmlist = new ProgramParameter[6];
```

- Declares a ProgramParameter array for six parameters.

```
AS400Text asFlag = new AS400Text(1);
parmlist[0] = new ProgramParameter(asFlag.toBytes("S"), 1);
```

- The first parameter is an action code of one character. An object of type AS400Text with a length of one is created and called asFlag. The asFlag object is used to convert a Java String object with a value of `S` to its AS/400 equivalent and returned as an array of bytes. This byte array is used as the input for a program parameter. The second parameter of the ProgramParameter constructor is an integer declaring the number of bytes expected to be returned by the program after execution.

```
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
   java.math.BigDecimal(partNo.trim())), 3);
```

- The second parameter is a part number and is both input and output. An AS400PackedDecimal conversion object is created to convert the part number to its AS/400 format. An output buffer of three bytes is reserved for the value returned by the called program.

```
AS400Text asDesc = new AS400Text(25);
parmlist[2] = new ProgramParameter(asDesc.toBytes(""), 25);
AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(0)),
   3);
AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
parmlist[4] = new ProgramParameter(asPrice.toBytes(new
   java.math.BigDecimal(0)), 4);
AS400Text asDate = new AS400Text(10);
parmlist[5] = new ProgramParameter(asDate.toBytes("0001-01-01"), 10);
```

- The next four parameters are used for output only, but require a correct value set. This is avoids errors in the server program, such as decimal data errors.

```
pgm.setProgram(progName, parmlist);
```

- Associates a program name and parameter list with the ProgramCall object.

```
if (pgm.run() != true)
```

- Uses the run method of the ProgramCall object to execute the program on the AS/400 system. The method returns a `true` value if successful and a `false` value if a problem occurred.

```
AS400Message[] messagelist = pgm.getMessageList();
for (int i = 0; i < messagelist.length; i++)
   {System.out.println(messagelist[i]);}
```

- If an error occurred on the run(), obtain the error messages from the ProgramCall object and print each message on the console.

```
if (((String) (asFlag.toObject(parmlist[0].getOutputData(), 0))).equals("Y"))
```

- This statement checks the value of the action parameter returned by the program to see if the part record was retrieved successfully.
  `parmlist[0].getOutputData()` returns an array of bytes for the first parameter in the AS/400 format. The toObject method is used on the AS400Text object, asFlag, to convert the byte array to a Java object. Since toObject returns an object of type Object, it must be typecast as a string object to use string methods.

```
partDesc.setText(((String) (new
   AS400Text(25)).toObject(parmlist[2].getOutputData(), 0)).trim());
partQty.setText(((java.math.BigDecimal) (new AS400PackedDecimal(5,
   0)).toObject(parmlist[3].getOutputData(), 0)).toString());
partPrice.setText(((java.math.BigDecimal) (new AS400PackedDecimal(6,
   2)).toObject(parmlist[4].getOutputData(), 0)).toString());
partDate.setText((String) (new
   AS400Text(10)).toObject(parmlist[5].getOutputData(), 0));
```

- The same technique is used to retrieve and convert parameter values from the AS/400 format to Java objects. The string representation of each output parameter is used to set the text property of the associated TextFields on the window.

3.5.14.4 The populateListBox Method

The populateListBox method is called from the DisplayAllParts non-default constructor through the PartsContainer Interface. It runs the RPG program multiple times to retrieve all records from the PARTS file.

```
public void populateListBox(com.ibm.ivj.eab.dab.IMulticolumnListbox aListBox) throws Exception {
ProgramParameter[] parmlist = new ProgramParameter[6];
AS400Text asFlag = new AS400Text(1);
parmlist[0] = new ProgramParameter(asFlag.toBytes("A"), 1);
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmlist[1] = new ProgramParameter(asPartNo.toBytes(new java.math.BigDecimal(0)), 3);
AS400Text asDesc = new AS400Text(25);
parmlist[2] = new ProgramParameter(asDesc.toBytes(""), 25);
AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(0)), 3);
AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
parmlist[4] = new ProgramParameter(asPrice.toBytes(new java.math.BigDecimal(0)), 4);
AS400Text asDate = new AS400Text(10);
parmlist[5] = new ProgramParameter(asDate.toBytes("0001-01-01"), 10);
pgm.setProgram(progName, parmlist);
String flag = null;
```

Figure 105. Distributed Program Call Example populateListBox Method (Part 1 of 2)

```
if (pgm.run() != true) {
System.out.println("program failed:" + progName);
AS400Message[] messagelist = pgm.getMessageList();
for (int i = 0; i < messagelist.length; i++) {
System.out.println(messagelist[i]);
}
return;
} else {
flag = (String) (asFlag.toObject(parmlist[0].getOutputData(), 0));
if (flag.equals("Y")) {
parmlist[0] = new ProgramParameter(asFlag.toBytes("F"), 1);
pgm.setProgram(progName, parmlist);
do {
if (pgm.run() != true) {
System.out.println("program failed:" + progName);
AS400Message[] messagelist = pgm.getMessageList();
for (int i = 0; i < messagelist.length; i++) {
System.out.println(messagelist[i]);
}
return;
} else {
flag = (String) (asFlag.toObject(parmlist[0].getOutputData(), 0));
if (flag.equals("Y")) {
String[] array = new String[5];
array[0] = (((java.math.BigDecimal) (new AS400PackedDecimal(5, 0)).toObject
    (parmlist[1].getOutputData(), 0)).toString());
array[1] = (String) (new AS400Text(25)).toObject(parmlist[2].getOutputData(), 0);
array[2] = WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal) (new
    AS400PackedDecimal(5, 0)).toObject(parmlist[3].getOutputData(), 0)).toString(), 5);
array[3] = WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal) (new
    AS400PackedDecimal(6, 2)).toObject(parmlist[4].getOutputData(), 0)).toString(), 8);
array[4] = (String) (new AS400Text(10)).toObject(parmlist[5].getOutputData(), 0);
aListBox.addRow(array, array[0]);
}}
} while (flag.equals("Y"));
}}
return;
}
```

Figure 106. Distributed Program Call Example populateListBox Method (Part 2 of 2)

Class: DPCExample Method: populateListBox

Method highlights:

```
AS400Text asFlag = new AS400Text(1);
parmlist[0] = new ProgramParameter(asFlag.toBytes("A"), 1);
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
   java.math.BigDecimal(0)), 3);
```

- The program parameter list is defined and initialized in the same manner as in the getRecord method. Here, the first parameter is set to **A** to tell the program to retrieve all records from the parts file. Zero is supplied for the part number.

```
if (pgm.run() != true)
```

- Calls the program the first time to open the parts file and position the read pointer to the first record in the file.

```
flag = (String)(asFlag.toObject(parmlist[0].getOutputData(),0));
if (flag.equals(Y))
```

- Checks if there was any record at all to retrieve. If the initial call to the program was successful, retrieve the value of the first parameter and check for an *operation succeeded* code (Y).

```
parmlist[0] = new ProgramParameter( asFlag.toBytes(" F") , 1);
```

- Change the value of the first parameter to an `F` to tell the program to retrieve the next record from the file.

Execute the program inside a do loop until the value returned in the first parameter is not a **Y**. This means that there are no more records to retrieve from the file. Upon each successful call to the program, use the same techniques as in the getRecord method to retrieve the values of output parameters. Place their Java String converted value into a string array for an addition to the list box.

3.5.14.5 The updateRecord Method

The updateRecord method is called by the ToolboxGUI class when the Update/Add Part button is pressed.

```
public String updateRecord(String partNo, String partDesc, String partQty, String partPrice, String
partDate) throws Exception
{
     ProgramParameter[] parmlist = new ProgramParameter[6];
     AS400Text asFlag = new AS400Text(1);
     parmlist[0] = new ProgramParameter(asFlag.toBytes("U"), 1);
     AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
     parmlist[1] = new ProgramParameter(asPartNo.toBytes(new java.math.BigDecimal(partNo.trim())),
                      3);
     AS400Text asDesc = new AS400Text(25);
     parmlist[2] = new ProgramParameter(asDesc.toBytes(partDesc), 25);
     AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
     parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(partQty)), 3);
     AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
     parmlist[4] = new ProgramParameter(asPrice.toBytes(new java.math.BigDecimal(partPrice)), 4);
     AS400Text asDate = new AS400Text(10);
     parmlist[5] = new ProgramParameter(asDate.toBytes(partDate), 10);
     pgm.setProgram(progName, parmlist);
     if (pgm.run() != true)
     {
             System.out.println("program failed:" + progName);
             AS400Message[] messagelist = pgm.getMessageList();
             for (int i = 0; i < messagelist.length; i++)
             {System.out.println(messagelist[i]);}
             return "Program call failed!";
     }
     else
     {
             if (((String) (asFlag.toObject(parmlist[0].getOutputData(), 0))).equals("Y"))
             {return "Record updated.";}
             else
             {return "Record added.";}
     }
}
```

Figure 107. Distributed Program Call Example updateRecord Method

Class: DPCExample Method: updateRecord

The updateRecord method highlights include:

```
AS400Text asFlag = new AS400Text(1);
parmlist[0] = new ProgramParameter(asFlag.toBytes("U"), 1);
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
   java.math.BigDecimal(partNo.trim())), 3);
AS400Text asDesc = new AS400Text(25);
parmlist[2] = new ProgramParameter(asDesc.toBytes(partDesc), 25);
AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
parmlist[3] = new ProgramParameter(asQty.toBytes(new
   java.math.BigDecimal(partQty)), 3);
AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
parmlist[4] = new ProgramParameter(asPrice.toBytes(new
   java.math.BigDecimal(partPrice)), 4);
AS400Text asDate = new AS400Text(10);
parmlist[5] = new ProgramParameter(asDate.toBytes(partDate), 10);
```

- Here, the first parameter is set to **U** to tell the program to retrieve a single record by part number, update it with the supplied attribute values, and return a **Y**, meaning that the record was updated. If the record cannot be found in the database file, the program writes it into the database file. The record is written with all the supplied data and the program returns an **X**, meaning that the record was added.

3.5.14.6 The deleteRecord Method

The deleteRecord method is called by the ToolboxGUI class when the Delete Part button is pressed. It works in the same way as the getRecord method, but a **D** is supplied as the operation code.

3.5.14.7 The dispose Method

The dispose method is called when the application window is closed.

```
public void dispose()
{
     try
     {
            as400.disconnectAllServices();
     }
     catch (Exception e)
     {
     };
     super.dispose();
     System.exit(0);
     return;
}
```

Figure 108. Distributed Program Call Example dispose Method

Class: DPCExample Method: dispose

The dispose method supports:

`as400.disconnectAllServices();`

- Releases all connections to the AS/400 system and releases resources associated with server jobs processing requests for the client.

All other methods used in the dispose method are the same as in the JDBC and StoredProcedure examples.

3.5.15 Data Queues

Data Queue classes allow a Java program to create, delete, write, and read data queues on the AS/400 system.

DataQueue classes allow a Java program to interact with AS/400 data queues. AS/400 data queues have the following characteristics:

- The data queue is a fast means of communications between jobs. Therefore, it is an excellent way to synchronize and pass data between jobs.
- Many jobs can access them simultaneously.

- Messages on a data queue are free format. Fields are not required as in database files.
- The data queue can be used for either synchronous or asynchronous processing.
- The messages on a data queue can be ordered in one of three ways:
  - *Last in, first out (LIFO)*

    The last (newest) message placed on the data queue is the first message taken off the queue.
  - *First in, first out (FIFO)*

    The first (oldest) message placed on the data queue is the first message taken off the queue.
  - *Keyed*

    Each message on the data queue has a key associated with it. A message can only be taken off the queue by specifying the key that is associated with it.
- Data queues allow for time independent applications. The client and server applications are not communicating directly and can work independent of each other.

The DataQueue class provides a complete set of interfaces to access AS/400 data queues from a Java program. It is an excellent way to communicate between Java programs and AS/400 programs. The AS/400 program can be written in any language.

A required parameter of the DataQueue constructor is the AS400 object that represents the AS/400 system that has the data queue or where the data queue is to be created. The DataQueue constructor requires the integrated file system path name of the data queue.

Two types of data queues are supported: keyed and non-keyed. Methods common to both types of queues are in the BaseDataQueue class. This class is extended by the DataQueue class to complete the implementation of non-keyed data queues. The BaseDataQueue class is extended by the KeyedDataQueue class to complete the implementation of keyed data queues.

When data is read from a data queue, it is placed in a DataQueueEntry object. This object holds the data for both keyed and non-keyed data queues. Additional data available when reading from a keyed data queue is placed in a KeyedDataQueueEntry object that extends the DataQueueEntry class. Consider this example:

```
// Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");

// Create the DataQueue object
DataQueue dq = new DataQueue(sys, "/QSYS.LIB/MYLIB.LIB/MYQUEUE.DTAQ");

// read data from the queue
DataQueueEntry dqData = dq.read();

// get the data out of the DataQueueEntry object as a byte array
byte[] data = dqData.getByteData();
```

```
// ... process the data

// Disconnect since I am done using data queues
sys.disconnectService("data queue");
```

The data queue classes do not alter data written to or read from the AS/400 data queue. It is up to the Java program to correctly format the data. The data conversion classes provide methods for converting data.

3.5.15.1 Keyed Data Queues

The BaseDataQueue and KeyedDataQueue classes provide the following methods for working with keyed data queues:

- Create a keyed data queue on the AS/400 system. The Java program must specify a key length and maximum size of an entry on the queue. The Java program can optionally specify authority information, save sender information, force to disk, and provide a queue description.
- Peek at an entry that matches the specified key without removing it from the queue. The Java program can wait or return immediately if no entry is currently on the queue that matches the specified key. The program can receive the entry as a string or as a byte array.
- Read an entry off the queue that matches the specified key. The Java program can wait or return immediately if no entry is available on the queue that matches the specified key. The program can read the entry as a string or as a byte array.
- Write an entry to the queue.
- Clear all entries that match the specified key.
- Delete the queue.

The BaseDataQueue and KeyedDataQueue classes also provide additional methods for retrieving the attributes of the data queue.

3.5.15.2 Non-Keyed Data Queues

Entries on a non-keyed AS/400 data queue are removed in FIFO or LIFO sequence. The BaseDataQueue and DataQueue classes provide the following methods for working with non-keyed data queues:

- Create a data queue on the AS/400 system. The Java program can optionally specify queue parameters (FIFO versus LIFO, save sender information, and so on) when the queue is created.
- Peek at an entry on the data queue without removing it from the queue. The Java program can wait or return immediately if no entry is currently on the queue, and can receive the entry as a string or as a byte array.
- Read an entry off the queue. The Java program can wait or return immediately if no entry is available on the queue, and can read the entry as a string or as a byte array.
- Write an entry to the queue.
- Clear all entries from the queue.
- Delete the queue.

The BaseDataQueue and DataQueue classes also provide additional methods for retrieving the attributes of the data queue.

3.5.16 Data Queue Application Example

In this example, we use the Data Queue interface to allow a client program to interface with an AS/400 program (see Figure 109).

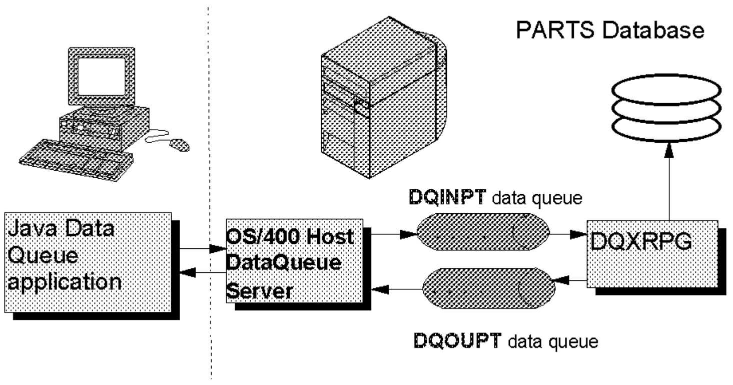
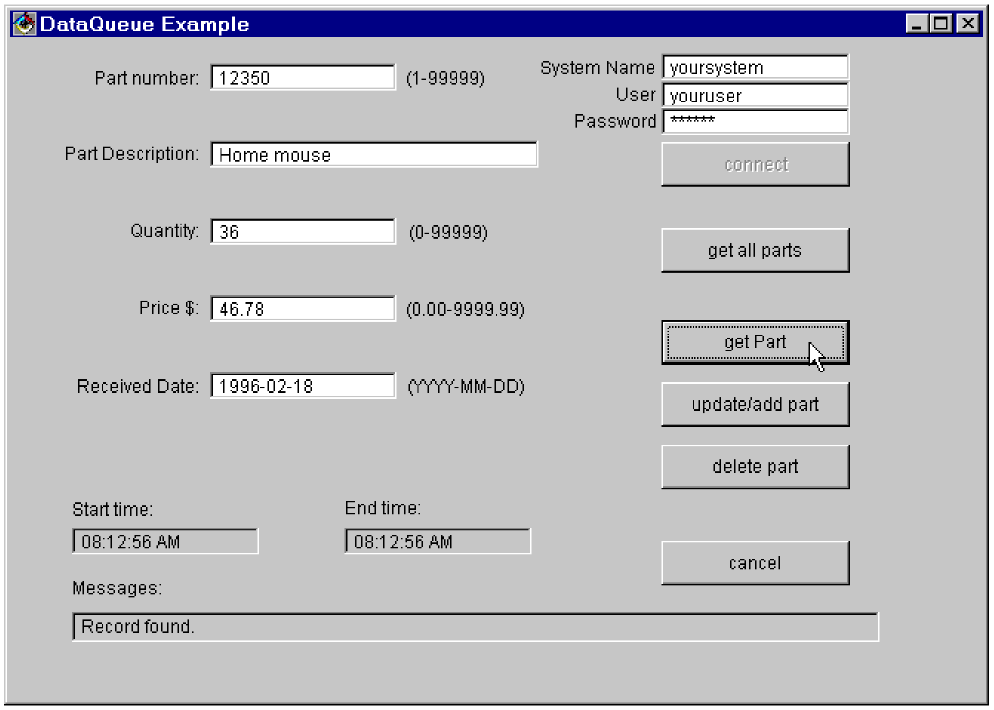


Figure 109. Data Queue Application

The client program requests data from the AS/400 database by placing requests on an input AS/400 data queue. A host program monitors the input data queue for a request. If a request is received, the host program uses it to retrieve records from the AS/400 database. The output information is placed in an output data queue that is monitored by the client program. When using data queues, it is up to the application implementer to handle data conversions.



Part number:	12350	(1-99999)
Part Description:	Home mouse	
Quantity:	36	(0-99999)
Price $:	46.78	(0.00-9999.99)
Received Date:	1996-02-18	(YYYY-MM-DD)

System Name	yoursystem
User	youruser
Password	******

Start time: 08:12:56 AM

End time: 08:12:56 AM

Messages: Record found.

Figure 110. Data Queue Example

Class DataQueueExample is the main class in this application. It is functionally equivalent to the StoredProcedureExample application, but is implemented using different techniques. The ToolboxGUI and DisplayAllParts classes are used to handle all user interaction. A native RPG program waits for a request on an input data queue (APILIB/DQINPT) and places results on an output data queue (APILIB/DQOUPT). Record create, read, update, and delete are implemented in this example.

3.5.16.1 Data Queue Server Program Background

An input queue and output queue were created with these commands:

```
CRTDTAQ DTAQ(APILIB/DQINPT) MAXLEN(48) TEXT('Data Queue for Parts Input')
CRTDTAQ DTAQ(APILIB/DQOUPT) MAXLEN(48) TEXT('Data Queue for Parts Output')
```

Table 26 shows the DQINPT layout for the data queue.

Table 26. Data Queue DQINPT Layout

Queue Position	Description	Length/Type	Values
1 - 1	Operation Code	1 character	S - read single record A - read all records E - end the program D - delete single record U - update/add single record
2 - 6	Part Number	5.0 zoned	
7 - 31	Part Description	25 character	
32 - 34	Part Quantity	5.0 packed	
35 - 38	Part Price	6.2 packed	
39 - 48	Part Date Received	10 date	

Table 27 shows the DQOUPT layout for the data queue.

Table 27. Data Queue DQOUPT Layout

Queue Position	Description	Length/Type	Values
1 - 1	Return Code	1 character	Y - record found, deleted, updated X - record not found, added, End of File
2 - 6	Part Number	5.0 zoned	
7 - 31	Part Description	25 character	
32 - 34	Part Quantity	5.0 packed	
35 - 38	Part Price	6.2 packed	
39 - 48	Part Date Received	10 date	

3.5.17 DataQueueExample Class

In this section, we investigate the key methods of the DataQueueExample Class.

3.5.17.1 Instance Variables

The following instance variables are declared for the class:

```
private AS400 as400;
private DataQueue dqInput;
private DataQueue dqOutput;
private RecordFormat rfInput;
private RecordFormat rfOutput;
```

3.5.17.2 The connectToDB Method

The connectToDB method is called by the ToolboxGUI class when the Connect button is pressed. String parameters representing the AS/400 system name, user ID, and password are passed to the method.

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
     as400 = new AS400(systemName, userid, password);
     dqInput = new com.ibm.as400.access.DataQueue(as400, "/QSYS.LIB/APILIB.LIB/DQINPT.DTAQ");
     dqOutput = new com.ibm.as400.access.DataQueue(as400, "/QSYS.LIB/APILIB.LIB/DQOUPT.DTAQ");
     dqInput.clear();
     dqOutput.clear();
     return;
}
```

Figure 111. Data Queue Example connectToDB Method

Class: DataQueueExample Method: connectToDB

This method supports these functions:

```
as400 = new com.ibm.as400.access.AS400(systemName, userid, password);
```

- Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

  Note: If you import the com.ibm.as400.access classes, you do not have to fully qualify the class name. You can write it as shown.

```
dqInput = new com.ibm.as400.access.DataQueue(as400,
   "/QSYS.LIB/APILIB.LIB/DQINPT.DTAQ");
dqOutput = new com.ibm.as400.access.DataQueue(as400,
   "/QSYS.LIB/APILIB.LIB/DQOUPT.DTAQ");
```

- Creates new DataQueue objects for the input and output queues. The fully-qualified IFS name of the data queues are passed in the constructor.

```
dqInput.clear();
dqOutput.clear();
```

- Clears both DataQueues, so there are no remaining entries from interrupted or unsuccessful tests. DataQueues that are not properly initialized can provoke confusing and unwanted results in your application.

3.5.17.3 The getRecord Method

The getRecord method is called by the ToolboxGUI class when the Get Part button is pressed.

```
public String getRecord(String partNo, java.awt.TextField partDesc, java.awt.TextField partQty,
java.awt.TextField partPrice, java.awt.TextField partDate) throws Exception
{
     if (rfInput == null) initRecordFormat();
     Record rInput = rfInput.getNewRecord();
     rInput.setField("flag","S");
     rInput.setField("partno",new java.math.BigDecimal(partNo.trim()));
     rInput.setField("partds","");
     rInput.setField("partqy",new java.math.BigDecimal(0));
     rInput.setField("partpr",new java.math.BigDecimal(0));
     rInput.setField("partdt","0001-01-01");
     dqInput.write(rInput.getContents());
     DataQueueEntry dqe = null;
     while (dqe == null)
     {
            dqe = dqOutput.read();
     }
     Record rOutput = rfOutput.getNewRecord(dqe.getData());
     if (((String)rOutput.getField("flag")).equals("Y"))
     {
            partDesc.setText(((String)rOutput.getField("partds")).trim());
            partQty.setText(((java.math.BigDecimal)rOutput.getField("partqy")).toString());
            partPrice.setText(((java.math.BigDecimal)rOutput.getField("partpr")).toString());
            partDate.setText((String)rOutput.getField("partdt"));
     }
     else
     {
            partDesc.setText("");
            partQty.setText("0");
            partPrice.setText("0.00");
            partDate.setText("");
            return "Record not found.";
     }
     return "Record found.";
}
```

Figure 112. Data Queue Example getRecord Method

Class: DataQueueExample Method: getRecord

Method highlights include:

```
if (rfInput == null) initRecordFormat();
```

- Uses lazy initialization to create the input and output record format objects. See the initRecordFormat method for details.

```
Record rInput = rfInput.getNewRecord();
```

- Creates a new input record object from the input record format. A RecordFormat is only a description of a record. A record is an object that can have field values.

```
rInput.setField("flag","S");
rInput.setField("partno",new java.math.BigDecimal(partNo.trim()));
rInput.setField("partds","");
rInput.setField("partqy",new java.math.BigDecimal(0));
rInput.setField("partpr",new java.math.BigDecimal(0));
rInput.setField("partdt","0001-01-01");
```

- Sets the value of the flag field in the record to `S` to tell the server program to retrieve a single record from the database. Set the value of the part field to the part number passed on the parameter list. Initialize the remaining fields with values accepted by the server program. This has to be done because the input parameters must match the requirements of the AS/400 data types and avoids having such server program troubles as decimal data errors or date conversion errors.

```
dqInput.write(rInput.getContents());
```

- Writes the input record to the input data queue. The getContents method returns a byte array of the value of the record in the AS/400 format.

```
DataQueueEntry dqe = null;
while (dqe == null) {dqe = dqOutput.read();}
```

- Initializes a DataQueueEntryObject. In a loop, the method dqOutput.read() reads the next entry off the output data queue. This returns a data queue entry object. Since we do not know how long it will take the server program to write the answer of our request into the output dataQueue, the program must loop until the expected record can be retrieved from there.

```
Record rOutput = rfOutput.getNewRecord(dqe.getData());
```

- Creates a new output record by using the output record format and setting field values that use the array of bytes returned by the getData method of the data queue entry object.

```
if (((String)rOutput.getField("flag")).equals("Y"))
```

- This statement checks the value of the flag field in the record format to see if the part record was retrieved successfully. The getField method uses an object of type Object. It must be typecast as a string object to use string methods.

```
partDesc.setText(((String)rOutput.getField("partds")).trim());
partQty.setText(((java.math.BigDecimal)rOutput.getField("partqy")).toString(
   ));
partPrice.setText(((java.math.BigDecimal)rOutput.getField("partpr")).toString
   ());
partDate.setText((String)rOutput.getField("partdt"));
```

- The same technique is used to retrieve and field values from the output record object to Java objects. The string representation of each output parameter is used to set the text property of the associated TextFields on the window.

3.5.17.4 The initRecordFormat Method

The initRecordFormat method initializes the input and output record format objects. It is called by the getRecord and populateListBox methods if the record formats are not already initialized.

```
public void initRecordFormat()
{
     CharacterFieldDescription asFlag = new CharacterFieldDescription(new AS400Text(1),"flag");
     ZonedDecimalFieldDescription asPartNo = new ZonedDecimalFieldDescription(new
                      AS400ZonedDecimal(5,0),"partno");
     CharacterFieldDescription asPartDS = new CharacterFieldDescription(new
                      AS400Text(25),"partds");
     PackedDecimalFieldDescription asPartQy = new PackedDecimalFieldDescription(new
                      AS400PackedDecimal(5,0),"partqy");
     PackedDecimalFieldDescription asPartPR = new PackedDecimalFieldDescription(new
                      AS400PackedDecimal(6,2),"partpr");
     DateFieldDescription asPartDt = new DateFieldDescription(new AS400Text(10),"partdt");
     rfInput = new RecordFormat();
     rfInput.addFieldDescription(asFlag);
     rfInput.addFieldDescription(asPartNo);
     rfInput.addFieldDescription(asPartDS);
     rfInput.addFieldDescription(asPartQy);
     rfInput.addFieldDescription(asPartPR);
     rfInput.addFieldDescription(asPartDt);
     rfOutput = new RecordFormat();
     rfOutput.addFieldDescription(asFlag);
     rfOutput.addFieldDescription(asPartNo);
     rfOutput.addFieldDescription(asPartDS);
     rfOutput.addFieldDescription(asPartQy);
     rfOutput.addFieldDescription(asPartPR);
     rfOutput.addFieldDescription(asPartDt);
     return;
}
```

Figure 113. Data Queue Example initRecordFormat Method

Class: DataQueueExample Method: initRecordFormat

This method includes:

```
CharacterFieldDescription asFlag = new CharacterFieldDescription(new
   AS400Text(1),"flag");
ZonedDecimalFieldDescription asPartNo = new ZonedDecimalFieldDescription(new
   AS400ZonedDecimal(5,0),"partno");
CharacterFieldDescription asPartDS = new CharacterFieldDescription(new
   AS400Text(25),"partds");
PackedDecimalFieldDescription asPartQy = new PackedDecimalFieldDescription(new
   AS400PackedDecimal(5,0),"partqy");
PackedDecimalFieldDescription asPartPR = new PackedDecimalFieldDescription(new
   AS400PackedDecimal(6,2),"partpr");
DateFieldDescription asPartDt = new DateFieldDescription(new
   AS400Text(10),"partdt");
```

- These statements create field description objects for the data fields that make up the input and output record formats. The field description constructor takes an AS/400 data type object and a field name.

```
rfInput = new RecordFormat();
rfInput.addFieldDescription(asFlag);
rfInput.addFieldDescription(asPartNo);
rfInput.addFieldDescription(asPartDS);
rfInput.addFieldDescription(asPartQy);
rfInput.addFieldDescription(asPartPR);
rfInput.addFieldDescription(asPartDt);
```

- Creates the input record format by adding field descriptions to a new RecordFormat object.

```
rfOutput = new RecordFormat();
rfOutput.addFieldDescription(asFlag);
rfOutput.addFieldDescription(asPartNo);
rfOutput.addFieldDescription(asPartDS);
rfOutput.addFieldDescription(asPartQy);
rfOutput.addFieldDescription(asPartPR);
rfOutput.addFieldDescription(asPartDt);
```

- Creates the output record format by adding field descriptions to a new RecordFormat object.

3.5.17.5 The populateListBox Method

The populateListBox method is called from the constructor. It sends a message on the input data queue to request all records from the parts file. It receives from the output data queue multiple times until all of the parts records are returned by the server program.

```
public void populateListBox(com.ibm.ivj.eab.dab.IMulticolumnListbox aListBox) throws Exception
{
     if (rfInput == null) initRecordFormat();
     Record rInput = rfInput.getNewRecord();
     rInput.setField("flag","A");
     rInput.setField("partno",new java.math.BigDecimal(0));
     rInput.setField("partds","");
     rInput.setField("partqy",new java.math.BigDecimal(0));
     rInput.setField("partpr",new java.math.BigDecimal(0));
     rInput.setField("partdt","0001-01-01");
     dqInput.write(rInput.getContents());
     String flag = null;
     do
     {
            DataQueueEntry dqe = null;
            while (dqe == null)
            {
                   dqe = dqOutput.read();
            }
            Record rOutput = rfOutput.getNewRecord(dqe.getData());
            flag = (String)rOutput.getField("flag");
            if (flag.equals("Y"))
            {
                   String[] array = new String[5];
                   array[0] =((java.math.BigDecimal)rOutput.getField("partno")).toString();
                   array[1] =(String)rOutput.getField("partds");
                   array[2] =WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                      rOutput.getField("partqy")).toString(), 5);
                   array[3] =WorkShop.DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                      rOutput.getField("partpr")).toString(), 8);
                   array[4] =(String)rOutput.getField("partdt");
                   aListBox.addRow(array, array[0]);
            }
     }
     while (flag.equals("Y"));
     return;
}
```

Figure 114. Data Queue Example populateListBox Method

Class: DataQueueExample Method: populateListBox

This method offers:

```
if (rfInput == null) initRecordFormat();
```

- The input and output record formats are initialized, if needed.

```
Record rInput = rfInput.getNewRecord();
```

- A new input record object is created from the input record format.

```
rInput.setField("flag","A");
```

- The flag field in the input record is set to an "A" to request all records from the parts file.

```
dqInput.write(rInput.getContents());
```

- Puts the current value of the input record format on the input data queue.

```
DataQueueEntry dqe = null;
while (dqe == null) {dqe = dqOutput.read();}
```

- Initializes a DataQueueEntryObject. In a loop, the method dqOutput.read() reads the next entry off the output data queue. This returns a data queue entry object. Since we do not know how long it will take the server program to write the answer of our request into the output dataQueue, the program must loop until the expected record can be retrieved from there.

```
Record rOutput = rfOutput.getNewRecord(dqe.getData());
```

- Uses the array of bytes returned by the method getData() to initialize a new output record object.

Execute the read inside a do loop until the value returned in the flag field is not a "Y". This means that there are no more records to retrieve from the file. Upon each successful read from the data queue, use the same techniques as the getRecord method to retrieve the values of output record fields and place their Java String converted value into a string array for addition to the list box.

3.5.17.6 The updateRecord Method

The updateRecord method is called by the ToolboxGUI class when the Update/Add Part button is pressed.

```
public String updateRecord(String partNo, String partDesc, String partQty, String partPrice, String
partDate) throws Exception
{
     if (rfInput == null)
            initRecordFormat();
     Record rInput = rfInput.getNewRecord();
     rInput.setField("flag", "U");
     rInput.setField("partno", new java.math.BigDecimal(partNo.trim()));
     rInput.setField("partds", partDesc);
     rInput.setField("partqy", new java.math.BigDecimal(partQty));
     rInput.setField("partpr", new java.math.BigDecimal(partPrice));
     rInput.setField("partdt", partDate);
     dqInput.write(rInput.getContents());
     DataQueueEntry dqe = null;
     while (dqe == null)
     {
            dqe = dqOutput.read();
     }
     Record rOutput = rfOutput.getNewRecord(dqe.getData());
     if (((String) rOutput.getField("flag")).equals("Y"))
     {
            return "1 Record updated.";
     }
     else
     {
            return "1 Record added.";
     }
}
```

Figure 115. Data Queue Example updateRecord Method

Class: DataQueueExample Method: updateRecord

The updateRecord method includes:

```
rInput.setField("flag", "U");
rInput.setField("partno", new java.math.BigDecimal(partNo.trim()));
rInput.setField("partds", partDesc);
rInput.setField("partqy", new java.math.BigDecimal(partQty));
rInput.setField("partpr", new java.math.BigDecimal(partPrice));
rInput.setField("partdt", partDate);
```

- Here, the first parameter is set to **U** to tell the program to retrieve a single record by its part number, update it with the supplied attribute values, and return a **Y**, meaning that the record was updated. If the record cannot be found in the database file, the program writes it into the database file. The record is written with all the supplied data and the program returns an **X**, meaning that the record has been added.

3.5.17.7 The deleteRecord Method

The deleteRecord method is called by the ToolboxGUI class when the Delete Part button is pressed. It works in the same way as the getRecord method, but a **D** is supplied as the operation code.

3.5.17.8 The dispose Method

The dispose method is called when the application window is closed.

```
public void dispose()
{
     try
     {
            as400.disconnectAllServices();
     }
     catch (Exception e)
     {
     };
     super.dispose();
     System.exit(0);
     return;
}
```

Figure 116. Data Queue Example dispose Method

Class: DataQueueExample Method: dispose

The dispose method includes:

`as400.disconnectAllServices();`

- Releases all connections to the AS/400 system and releases resources associated with server jobs processing requests for the client.

`super.dispose();`

- Calls the super class dispose method to make sure any resources used by the frame are properly freed.

All other methods used in the dispose method are the same as in the JDBC and StoredProcedure Examples.

3.6 Network Print

The network print classes provide the following functions:

- Read and write AS/400 spooled files
- Generate SCS data streams
- Manage print resources:
  - List, hold, and release spooled files
  - List, hold, and release output queues
  - Start and stop AS/400 writer jobs
  - List and retrieve attributes of printer devices
  - List and read AFP resources

Using the Network Print classes involves the following steps:

1. Establish a connection.
2. Create a spooled file list.
3. Set the user filter.
4. Open the spooled file list.
5. Retrieve entries.

6. Close the spooled file list.
7. Close the connection.

3.6.1 Print Example

In this example, we use the SpooledFileList feature of the AS/400 Toolbox to allow a Java program to directly access spooled files on the AS/400 system (Figure 117).

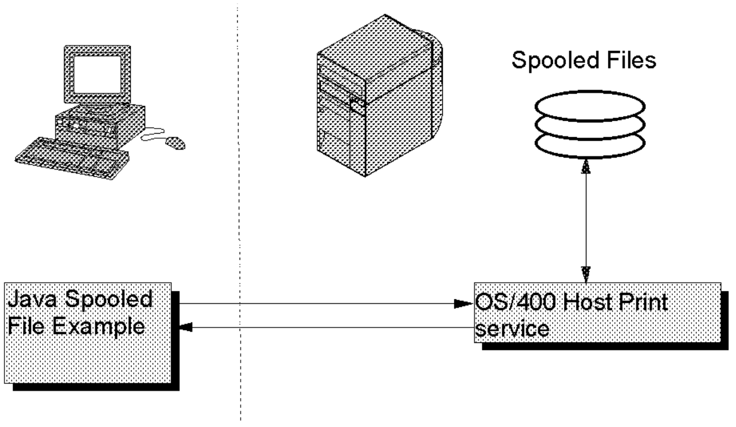


Figure 117. AS/400 Toolbox for Java Print

Spooled files can be created, deleted, held, and released. Spooled files can also be filtered by user name.

The Toolbox classes used are:

- **AS400(String, String, String)**

  Constructor for class com.ibm.as400.access.AS400. Constructs an AS400 object for the specified system, user ID, and password.

- **SpooledFileList(AS400)**

  Constructor for class com.ibm.as400.access.SpooledFileList. Constructs a SpooledFileList to an AS/400 system.

- **SpooledFile()**

  Constructor for class com.ibm.as400.access.SpooledFile. Constructs a spooled file object.

The spooledFileList methods used include:

- **openSynchronously()**

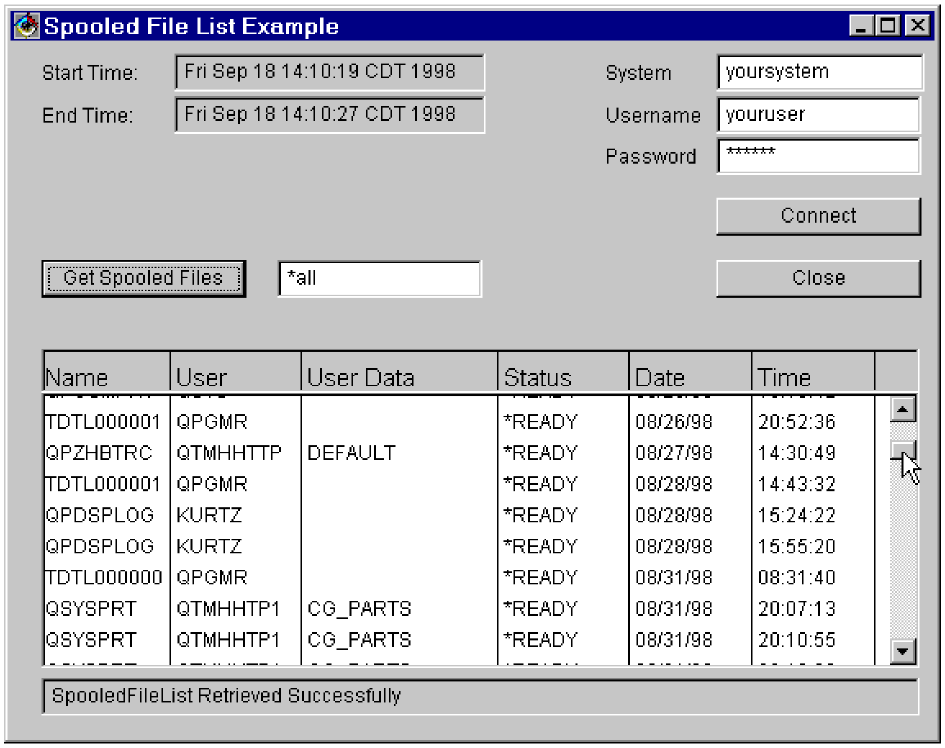
  Builds the list synchronously. This method does not return until the list is built completely.

- **getObject(int index)**

  Returns one object from the list.

- **close()**

  Closes the spooled file list.



Spooled File List Example

Start Time: Fri Sep 18 14:10:19 CDT 1998

End Time: Fri Sep 18 14:10:27 CDT 1998

System yoursystem

Username youruser

Password ******

Name	User	User Data	Status	Date	Time
TDTL000001	QPGMR		*READY	08/26/98	20:52:36
QPZHBTRC	QTMHHTTP	DEFAULT	*READY	08/27/98	14:30:49
TDTL000001	QPGMR		*READY	08/28/98	14:43:32
QPDSPLOG	KURTZ		*READY	08/28/98	15:24:22
QPDSPLOG	KURTZ		*READY	08/28/98	15:55:20
TDTL000000	QPGMR		*READY	08/31/98	08:31:40
QSYSPRT	QTMHHTP1	CG_PARTS	*READY	08/31/98	20:07:13
QSYSPRT	QTMHHTP1	CG_PARTS	*READY	08/31/98	20:10:55

SpooledFileList Retrieved Successfully

Figure 118. Spooled File Example

This application was created using VisualAge for Java and the AS/400 Toolbox classes. The spooled file list retrieval application allows us to retrieve a list of spooled files of the current logged on user, but any user name can be used. Figure 118 shows the spooled files for the current logged on user.

3.6.2 SpooledFileListExample Class

In this section, we investigate the key methods of the SpooledFileListExample class.

3.6.2.1 The connect Method

The connect method is called when the Connect button is pressed. String parameters representing the AS/400 system name, user ID, and password are passed to the method.

```
public String connect(String systemName, String userid, String password)
{
     try
     {
            as400 = new AS400(systemName, userid, password);
            as400.connectService(as400.PRINT);
     }
     catch (Exception e)
     {
            return "Exception " + e;
     }
     return "Connected";
}
```

Figure 119. Spooled File Example connect Method

This method supports these features:

`as400 = new AS400(systemName, userid, password);`

- Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

`as400.connectService(AS400.PRINT);`

- Connect to the AS/400 Program Call and Print Service server. This is not a required call. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method, as opposed to connecting the first time that the user requests a spooled file.

3.6.2.2 The formatSpooledFile Method

The formatSpooledFile method is called to format the Print objects so they can be added to the Multi-column list box. It is called by the getSpooledFilesForUser method. It is called with a SpooledFile Object as input. It retrieves the attributes of the object and returns a string array that contains the formatted attributes of the object.

```
public String[] formatSpooledFile(com.ibm.as400.access.SpooledFile theFile) throws Exception
{
     String result[] = new String[6];
     result[0] = theFile.getStringAttribute(PrintObject.ATTR_SPOOLFILE);
     result[1] = theFile.getStringAttribute(PrintObject.ATTR_JOBUSER);
     result[2] = theFile.getStringAttribute(PrintObject.ATTR_USERDATA);
     result[3] = theFile.getStringAttribute(PrintObject.ATTR_SPLFSTATUS);
     String date = theFile.getStringAttribute(PrintObject.ATTR_DATE);
     result[4] = date.substring(3, 5) + "/" + date.substring(5, 7) + "/" + date.substring(1, 3);
     String time = theFile.getStringAttribute(PrintObject.ATTR_TIME);
     result[5] = time.substring(0, 2) + ":" + time.substring(2, 4) + ":" + time.substring(4, 6);
     return result;
}
```

Figure 120. Spooled File Example formatSpooledFile Method

3.6.2.3 The getSpooledFilesForUser (String User) Method

The getSpooledFilesForUser method highlights include:

- New **spooledFileList (as400)**

  Constructs a spooled file list object using the system object. The default list shows all spooled files for the current user on the specified system.

- **list.setUserFilter (String user name)**

  Specifies the user data that the spooled file must have for it to be included in the list. The value can be any specific value or the special *ALL value. The value cannot be greater than 10 characters. The default is *ALL.

- **list.openSynchronously()**

  Builds the list synchronously. This method does not return until the list is built completely. The caller may then call the getObjects method to get an enumeration of the list.

- **list.size()**

  Returns the current size of the list.

- **currentFile = (SpooledFile)list.getObject(x)**

  Returns one object from the list.

- **getMultiColumnListbox1().addRow(formatSpooledFile(currentFile), currentFile)**

  Calls formatSpooledFile to retrieve the attributes of the object and then adds it to the list box.

- **list.close()**

  Closes the list so that objects in the list can be garbage collected.

```
public String getSpooledFilesForUser(String user)
{
     SpooledFileList list;
     try
     {
            clearListbox();
            list = new SpooledFileList(as400);
            list.setUserFilter(user.toUpperCase());
            list.openSynchronously();
            int listsize = list.size();
            SpooledFile currentFile;
            for (int x = 0; x < listsize; x++)
            {
                   currentFile = (SpooledFile) list.getObject(x);
                   getIMulticolumnListbox1().addRow(formatSpooledFile(currentFile), currentFile);
            }
     }
     catch (Exception e)
     {
            return "An exception occurred" + e;
     }
     list.close();
     getIMulticolumnListbox1().repaint();
     return "SpooledFileList Retrieved Successfully";
}
```

Figure 121. Spooled File Example getSpooledFileForUser Method

3.7 Integrated File Systems Access

Integrated file system support allows access to files in the AS/400 system's integrated file systems as a stream of bytes. It provides a function similar to the java.io package, plus the ability to:

- Specify a file sharing mode to deny access to the file while it is in use
- Specify a file creation mode to open, create, or replace the file
- Lock a section of the file to deny access to that part of the file while it is in use
- List the contents of a directory more efficiently
- Determine the number of bytes available on the AS/400 system
- Get detailed information on why an operation failed

The Integrated File System Stream File classes were created because the java.io package does not provide file redirection. The Integrated File System Stream File classes also provide functions that are not in the java.io package. The function provided by the Integrated File System Stream File classes is a superset of the function provided by the file IO classes in the java.io package. All methods in java.io InputStream, OutputStream, and RandomAccessFile are in the Integrated File System Stream File classes.

The Integrated File System Stream File classes also allow a Java applet to write files to the AS/400 file system. Java applets cannot use the java.io package to write to the file system.

A Java program can still use the java.io package, but a method of redirection must be provided by the client operating system. For example, if the Java program is running on a Windows 95 or Windows NT personal computer, the network drive function of AS/400 Client Access for 32-bit Windows is required for java.io calls to the AS/400 system. With the Integrated File System Stream File classes, Client Access is not required.

Integrated File System Stream File classes require the hierarchical name of the object in the integrated file system. Use the forward slash as the path separator character. For example, to access FILE1 in directory path DIR1/DIR2, the name is:

```
/DIR1/DIR2/FILE1
```

3.7.1 Integrated File System Example

In this example, we use the integrated file system classes of the AS/400 Toolbox to allow a Java program to interface with the OS/400 host servers to gain access to files in the AS/400 integrated file system (Figure 122).

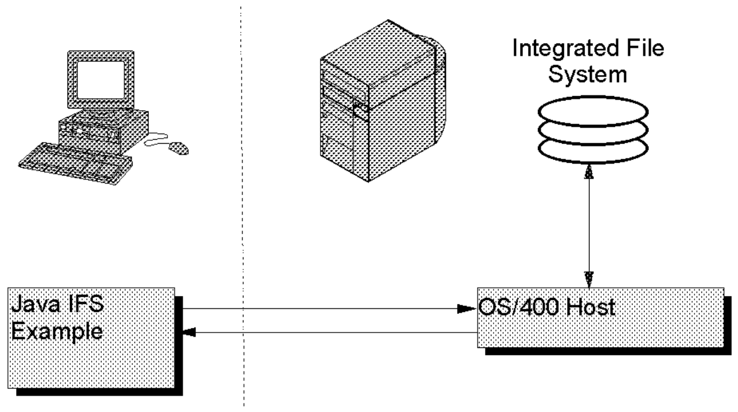


Figure 122. AS/400 Toolbox for Java IFS

The toolbox classes used in this example are:

- **AS400(String, String, String)**

  Constructor for class com.ibm.as400.access.AS400. Constructs an AS400 object for the specified system, user ID, and password.

- **IFSFile(AS400, String)**

  Constructor for class com.ibm.as400.access.IFSFile. Constructs an object referring to an IFS File on the AS/400 system.

- **IFSFileInputStream(AS400, IFSFile, int)**

  Constructor for class com.ibm.as400.access.IFSFileInputStream. Constructs an input stream to read contents of the file.

The IFS methods used in this example are:

- **list()**

  Method in class com.ibm.as400.access.IFSFile. If the IFSFile object represents a directory or folder, this method returns an array of strings that holds the list of all files and directories within.

- **getSystem()**

  Method in class com.ibm.as400.access.IFSFile. Returns the AS400 object from which this IFSFile was created.

- **getName()**

  Method in class com.ibm.as400.access.IFSFile. Returns a string with the name of the IFSFile.

- **isDirectory() and isFile()**

  Methods in class com.ibm.as400.access.IFSFile. Return booleans to determine whether the IFSFile object represents a file or directory.

- **length()**

  Method in class com.ibm.as400.access.IFSFile. Returns the length (in bytes) of the file.

- **lastModified()**

  Method in class com.ibm.as400.access.IFSFile. Returns the last date that the file was modified (as a long).

- **available()**

  Method in class com.ibm.as400.access.IFSFileInputStream. Returns the number of available bytes in the file.

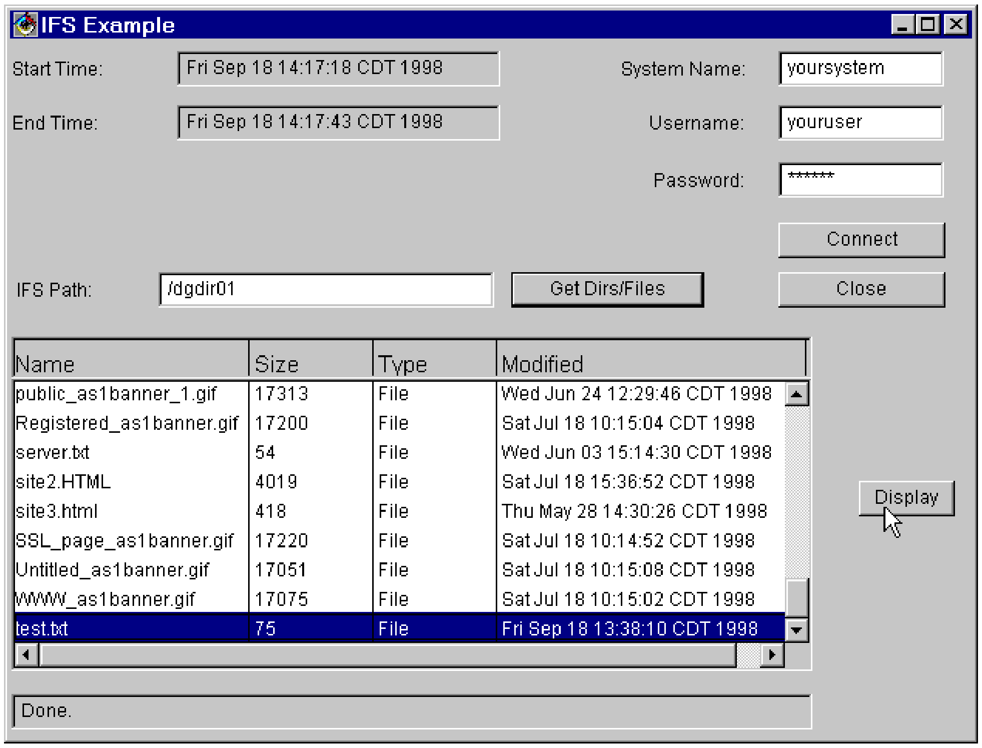
- **read(byte[])**

  Method in class com.ibm.as400.access.IFSFileInputStream. Reads the number of bytes available and stores in the byte array.

- **close()**

  Method in class com.ibm.as400.access.IFSFileInputStream. Closes the input stream.

This application was built using VisualAge for Java and the AS/400 Toolbox. In this example, we use the integrated file system classes of the AS/400 Toolbox to allow a Java program to retrieve a list of files from the AS/400 integrated file system. Figure 123 on page 175 shows the files stored in the IFS for the path entered in the text box.



Start Time: Fri Sep 18 14:17:18 CDT 1998

End Time: Fri Sep 18 14:17:43 CDT 1998

System Name: yoursystem

Username: youruser

Password: ******

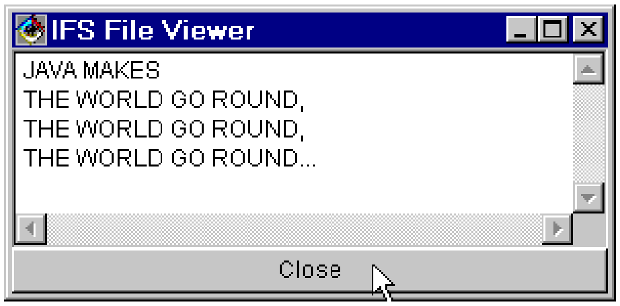
IFS Path: /dgdir01

Name	Size	Type	Modified
public_as1banner_1.gif	17313	File	Wed Jun 24 12:29:46 CDT 1998
Registered_as1banner.gif	17200	File	Sat Jul 18 10:15:04 CDT 1998
server.txt	54	File	Wed Jun 03 15:14:30 CDT 1998
site2.HTML	4019	File	Sat Jul 18 15:36:52 CDT 1998
site3.html	418	File	Thu May 28 14:30:26 CDT 1998
SSL_page_as1banner.gif	17220	File	Sat Jul 18 10:14:52 CDT 1998
Untitled_as1banner.gif	17051	File	Sat Jul 18 10:15:08 CDT 1998
WWW_as1banner.gif	17075	File	Sat Jul 18 10:15:02 CDT 1998
test.txt	75	File	Fri Sep 18 13:38:10 CDT 1998

Done.

Figure 123. Integrated File System Example (Directories/Files)

If a text file is selected from the list of files, its contents are displayed as shown in Figure 124.



```
JAVA MAKES
THE WORLD GO ROUND,
THE WORLD GO ROUND,
THE WORLD GO ROUND...
```

Figure 124. Integrated File System Example (File Viewer)

To build this application, the following steps are required:

1. Establish a connection.
2. Set the IFS Path to view.
3. Create an IFSFile object.
4. Retrieve the list of files.
5. Open an IFSFileInputStream.
6. Read the file contents.
7. Close the connection.

3.7.2 IFSExample Class

In this section, we investigate the key methods of the IFSExample class.

3.7.2.1 The connect Method

The connect method is called when the Connect button is pressed. String parameters representing the AS/400 system name, user ID, and password are passed to the method.

```
public void connect(String systemName, String userid, String password) throws Exception
{
     getStatus().setText("Connecting....");
     as400 = new com.ibm.as400.access.AS400(systemName, userid, password);
     as400.connectService(com.ibm.as400.access.AS400.FILE);
     getStatus().setText("Connected to AS400");
     return;
}
```

Figure 125. Integrated File System Example connect Method

The connect method offers the following highlights:

```
as400 = new com.ibm.as400.access.AS400(systemName, userid, password);
```

- Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

```
as400.connectService(AS400.com.ibm.as400.access.AS400.FILE);
```

- Connects to the AS/400 host file server. This is not a required call. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method as opposed to connecting the first time the user requests to access a file.

3.7.2.2 The populateList Method

The populateList method is called when the Get Dirs/Files button is pressed. A string parameter representing the IFS path is passed to the method.

```
public void populateList(String IFSPath)
{
     com.ibm.as400.access.IFSFile aFile;
     String[] files;
     Object rowKey;
     getStatus().setText("Retrieving...");
     clearListbox();
     try
     {
            aFile = new com.ibm.as400.access.IFSFile(as400, IFSPath);
            files = aFile.list();
            for (int i=0; i<files.length; i++)
            {
                   aFile = new com.ibm.as400.access.IFSFile(as400,IFSPath, files[i]);
                   rowKey = aFile;
                   String[] rowData = new String[4];
                   rowData[0] = files[i];
                   rowData[1] = String.valueOf(aFile.length());
                   if (aFile.isDirectory())
                   {
                          rowData[2] = "Directory";
                   }
                   else
                   {
                          rowData[2] = "File";
                   }
                   rowData[3] = new java.util.Date(aFile.lastModified()).toString();
                   getIMulticolumnListbox1().addRow(rowData, rowKey);
            }
     }
     catch (java.io.IOException ex)
     {
            System.out.println("Error Receiving Files: "+ex);
     }
     getIMulticolumnListbox1().repaint();
     getStatus().setText("Done.");
     return;
}
```

Figure 126. Integrated File System Example populateList Method

The populateList method highlights include:

```
aFile = new com.ibm.as400.access.IFSFile(as400, IFSPath);
```

- Creates a new IFSFile object using the AS400 object and the IFS path as parameters.

```
files = aFile.list();
```

- Uses the IFSFile list method to return an array of strings that holds a list of files and directories held in the IFSFile object.

```
for (int i=0; i<files.length; i++) {
   aFile = new com.ibm.as400.access.IFSFile(as400,IFSPath, files[i]);
   .
   .
```

- Loops through the list of file objects stored in the string array named files and builds an object array that contains the name of the file, the size, the type, and the last modified date.

```
getMultiColumnListbox1().addRow(rowData, rowKey);
```

- Adds a new entry to the multi-column list box for the file object.

3.7.2.3 The readFile() Method

The readFile() method reads the contents of a file as a stream of bytes and stores them in a byte array.

```
protected void readFile()
{
     byte[] data = null;

     // Determine if the file extension is .txt
     String name = _file.getName();
     int index = name.lastIndexOf(".");
     if (index == -1)
     {
            getFileContents().setText("Error: Only .txt files can be viewed.");
            return;
     }
     if (!(name.substring(index + 1).toUpperCase()).equals("TXT"))
     {
            getFileContents().setText("Error: Only .txt files can be viewed.");
            return;
     }
     try
     {
            IFSFileInputStream in = new IFSFileInputStream(_file.getSystem(), _file,
                      IFSFileInputStream.SHARE_ALL);
            int len = in.available();
            data = new byte[len];
            in.read(data);
            in.close();
     }
     catch (Exception ex)
     {
            System.err.println("Error reading file: " + ex);
     }
     String t = new String(data);
     getFileContents().setText(t);
}
```

Figure 127. Integrated File System Example readFile Method

Readfile() method highlights:

```
IFSFileInputStream in = new IFSFileInputStream(_file.getSystem(), _file,
          IFSFileInputStream.SHARE_ALL);
```

- Creates a new IFSFileInputStream object that is used to read the contents of a file.

```
int len = in.available();
```

- Uses the IFSFileInputStream method `available` to determine the number of bytes contained in the file.

```
data = new byte[len];
```

- Allocates a byte array to hold the data of the size returned previously.

```
in.read(data);
```

- Uses the IFSFileInputStream method `read` to read the stream of bytes into the byte array.

Chapter 4. AS/400 Toolbox for Java — GUI Classes

New to the AS/400 Toolbox for Java with Modification 1 are the graphical user interface (GUI) classes, which are part of the vaccess package. With the GUI classes, you can visually represent your AS/400 data and resources. This chapter covers the GUI classes available as part of the AS/400 Toolbox for Java Modification 1. This support is available with OS/400 V4R3. The AS/400 Toolbox for Java Modification 2 offers additional GUI classes. They are described in Chapter 5, "AS/400 Toolbox for Java Modification 2" on page 213.

Java programs that use the AS/400 Toolbox for Java GUI classes also need Sun's JDK Swing 1.0.1 support. Swing is available with Sun's JFC (Java Foundation Classes) 1.1. For more information about Swing, visit the Web site at: http://www.javasoft.com/products/jfc/index.html

VisualAge for Java 2.0 provides this support so you can use these classes in the Visual Composition Editor. The AS/400 Toolbox for Java provides GUI classes for the following resources:

- AS/400 panes
- Java Database Connectivity (JDBC)
- Data queues
- Command call
- Error events
- Jobs
- Messages
- Network print
- Program call
- Record-level access
- Users and groups

4.1 Overview of the GUI Classes

This section presents an overview of each of the GUI classes. Basic programming examples are provided by the *AS/400 Toolbox for Java API Users Guide* that is available with the AS/400 Toolbox for Java. This chapter includes examples that use the GUI classes and VisualAge for Java 2.0 to produce AS/400 client/server applications.

The source code for any of the examples discussed in this chapter is available on the Internet. Please refer to Section A.1, "Downloading the Files from the Internet" on page 396, for details.

4.1.1 AS/400 Panes

AS/400 panes are graphical user interface classes that present and allow manipulation of one or more AS/400 resource. The behavior of each AS/400 resource varies depending on the type of resource. All panes extend the Java Component class. As a result, they can be added to any AWT Frame, Window, or Container.

The following AS/400 panes are available:

- **AS400ListPane** presents a list of AS/400 resources and allows selection of one or more resources (Figure 128).

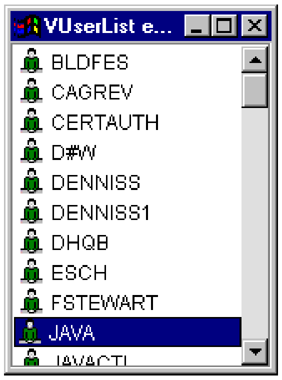
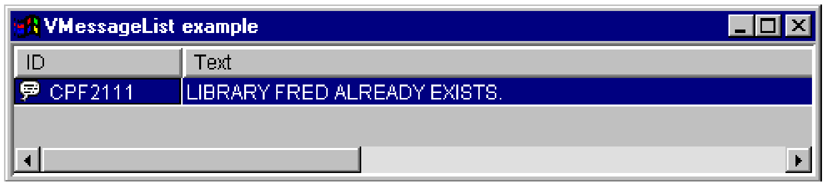


Figure 128. AS400ListPane

- **AS400DetailsPane** presents a list of AS/400 resources in a table where each row displays various details about a single resource. The table allows the selection of one or more resources (Figure 129).



ID	Text
CPF2111	LIBRARY FRED ALREADY EXISTS.

Figure 129. AS400DetailsPane

- **AS400TreePane** presents a tree hierarchy of AS/400 resources and allows the selection of one or more resources (Figure 130).

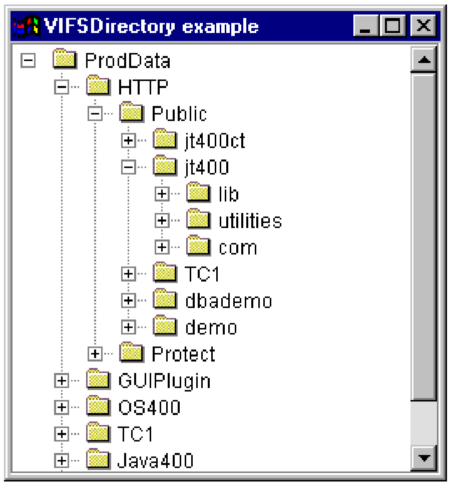
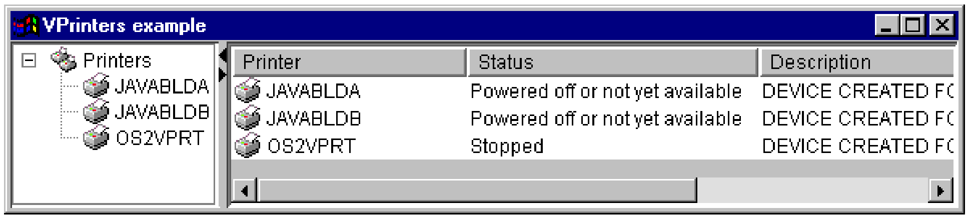


Figure 130. AS400TreePane

- **AS400ExplorerPane** combines an AS400TreePane and AS400DetailsPane so that the resource selected in the tree is presented in the details (Figure 131 on page 183).



Printer	Status	Description
JAVABLDA	Powered off or not yet available	DEVICE CREATED F(
JAVABLDB	Powered off or not yet available	DEVICE CREATED F(
OS2VPRT	Stopped	DEVICE CREATED F(

Figure 131. AS400ExplorerPane

4.1.2 JDBC

Java Database Connectivity (JDBC) graphical user interface classes allow a Java program to present various views and controls for accessing a database using SQL (Structured Query Language) statements and queries. The following classes are available:

- **SQLStatementButton** — A button that issues an SQL statement when clicked.
- **SQLStatementMenuItem** — A menu item that issues an SQL statement when selected.
- **SQLStatementDocument** — A document that can be used with any Java Foundation Classes (JFC) graphical text component to issue an SQL statement.
- **SQLResultSetFormPane** — Presents the results of an SQL query in a form.
- **SQLResultSetTablePane** — Presents the results of an SQL query in a table.
- **SQLResultSetTableModel** — Manages the results of an SQL query in a table.
- **SQLQueryBuilderPane** — Presents an interactive tool for dynamically building SQL statements.

4.1.2.1 SQL Connections

An SQLConnection object represents a connection to a database using JDBC. The SQLConnection object is used with all of the JDBC graphical user interface classes. To use an SQLConnection, set the URL property using the constructor or setURL() method. This identifies the database to which the connection is made. Other optional properties can be set:

- Use **setProperties()** to specify a set of JDBC connection properties.
- Use **setUserName()** to specify the user name for the connection.
- Use **setPassword()** to specify the password for the connection.

The actual connection to the database is not made when the SQLConnection object is created. Instead, it is made when the getConnection() method is called. This method is normally called automatically by the JDBC graphical user interface classes, but it can be called at any time to control when the connection is made.

An SQLConnection object can be used for more than one JDBC graphical user interface component. All such classes use the same connection, which can improve performance and resource usage. Alternately, each JDBC graphical user interface component can use a different SQL object. It is sometimes necessary to use separate connections so that SQL statements are issued in different transactions. When the connection is no longer needed, close the

SQLConnection object using the close() method. This frees up JDBC resources on both the client and server.

4.1.3 Command Call

The command call graphical user interface classes allow a Java program to present a button or menu item that calls a non-interactive AS/400 command. A CommandCallButton object represents a button that calls an AS/400 command when pressed. The CommandCallButton class extends the Java Foundation Classes (JFC) JButton class so that all buttons have a consistent appearance and behavior. Similarly, a CommandCallMenuItem object represents a menu item that calls an AS/400 command when selected. The CommandCallMenuItem class extends the JFC JMenuItem class so that all menu items also have a consistent appearance and behavior. To use a command call graphical user interface component, set both the system and command properties. These properties can be set using a constructor or through the setSystem() and setCommand() methods.

4.1.4 Data Queues

The data queue graphical classes allow a Java program to use any Java Foundation Classes (JFC) graphical text component to read or write to an AS/400 data queue. The DataQueueDocument and KeyedDataQueueDocument classes are implementations of the JFC Document interface. These classes can be used directly with any JFC graphical text component. Several text components, such as single line fields (JTextField) and multiple line text areas (JTextArea), are available in JFC. Data queue documents associate the contents of a text component with an AS/400 data queue. A *text component* is a graphical component used to display text that the user can optionally edit. The Java program can read and write between the text component and data queue at any time. Use DataQueueDocument for sequential data queues, and use Keyed DataQueueDocument for keyed data queues.

To use a DataQueueDocument, set both the system and path properties. These properties can be set using a constructor or through the setSystem() and setPath() methods. The DataQueueDocument object is "plugged" into the text component, usually using the text component's constructor or setDocument() method. KeyedDataQueueDocuments work the same way.

4.1.5 Error Events

In most cases, the AS/400 Toolbox for Java GUI classes fire *error events* instead of throwing exceptions. An error event is a wrapper around an exception that is thrown by an internal component.

You can provide an error listener that handles all error events fired by a particular graphical user interface component. When an exception is thrown, the listener is called and can provide appropriate error reporting. By default, no action takes place when error events are fired.

The AS/400 Toolbox for Java provides a graphical user interface component called ErrorDialogAdapter, which automatically displays a dialog to the user when an error event is fired.

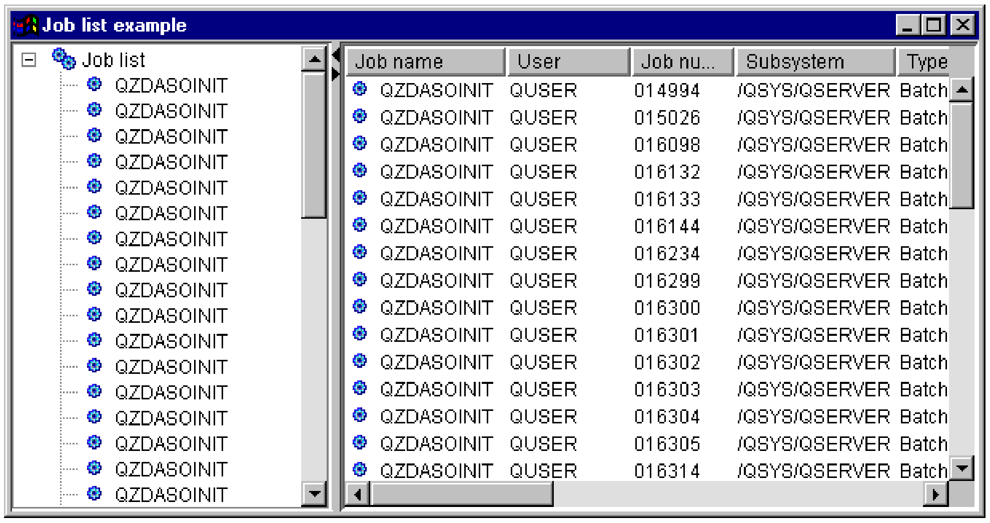
4.1.6 Jobs

The jobs graphical user interface classes allow a Java program to present lists of AS/400 jobs and job log messages in a graphical user interface. The following classes are available:

- **VJobList object** — A resource that represents a list of AS/400 jobs for use in AS/400 panes.
- **VJob object** — A resource that represents the list of messages in a job log for use in AS/400 panes.

You can use AS/400 panes, VJobList objects, and VJob objects together to present many views of a job list or job log.

The VJobList example shown in Figure 132 presents an AS400ExplorerPane filled with a list of jobs. The list shows jobs on the system that have the same job name.



Job name	User	Job nu...	Subsystem	Type
QZDASOINIT	QUSER	014994	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	015026	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016098	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016132	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016133	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016144	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016234	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016299	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016300	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016301	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016302	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016303	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016304	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016305	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016314	/QSYS/QSERVER	Batch

Figure 132. VJobList Graphical User Interface Component

4.1.7 Messages

The messages graphical user interface classes allow a Java program to present lists of AS/400 messages in a graphical user interface.

The following classes are available:

- **VMessageList object** — A resource that represents a list of messages for use in AS/400 panes. This is for message lists generated by command or program calls.
- **VMessageQueue object** — A resource that represents the messages in an AS/400 message queue for use in AS/400 panes.

You can use AS/400 pane, VMessageList, and VMessageQueue objects together to present many views of a message list and to allow the user to select and perform operations on messages.

4.1.8 Network Print

The network print graphical user interface classes allow a Java program to present lists of AS/400 network print resources in a graphical user interface.

The following classes are available:

- **VPrinters object** — A resource that represents a list of printers for use in AS/400 panes. A VPrinter object is a resource that represents a printer and its spooled files for use in AS/400 panes.
- **VPrinterOutput object** — A resource that represents a list of spooled files for use in AS/400 panes.

AS/400 pane, VPrinters, VPrinter, and VPrinterOutput objects can be used together to present many views of network print resources and allow the user to select and perform operations on them.

4.1.9 Program Call

The program call graphical user interface classes allow a Java program to present a button or menu item that calls an AS/400 program. Input, output, and input/output parameters can be specified using ProgramParameter objects. When the program runs, the output and input/output parameters contain data returned by the AS/400 program.

A ProgramCallButton object represents a button that calls an AS/400 program when pressed. The ProgramCallButton class extends the Java Foundation Classes (JFC) JButton class so that all buttons have a consistent appearance and behavior.

Similarly, a ProgramCallMenuItem object represents a menu item that calls an AS/400 program when selected. The ProgramCallMenuItem class extends the JFC JMenuItem class so that all menu items also have a consistent appearance and behavior.

To use a program call graphical user interface component, set both the system and program properties. Set these properties by using a constructor or the setSystem() and setProgram() methods.

4.1.10 Record-Level Access

The record-level access graphical user interface classes allow a Java program to present various views of AS/400 files. The following classes are available:

- **RecordListFormPane** — Presents a list of records from an AS/400 file in a form.
- **RecordListTablePane** — Presents a list of records from an AS/400 file in a table.
- **RecordListTableModel** — Manages the list of records from an AS/400 file for a table.

4.1.10.1 Keyed Access

You can use the record-level access graphical user interface classes with keyed access to an AS/400 file. Keyed access means that the Java program can access the records of a file by specifying a key.

Keyed access works the same for each record-level access graphical user interface component. Use the setKeyed() method to specify keyed access instead of sequential access. Specify a key using the constructor or the setKey() method.

By default, only records whose keys are equal to the specified key are displayed. To change this, specify the searchType property using the constructor or setSearchType() method. Possible choices include:

- **KEY_EQ** — Displays records whose keys are equal to the specified key.
- **KEY_GE** — Displays records whose keys are greater than or equal to the specified key.
- **KEY_GT** — Displays records whose keys are greater than the specified key.
- **KEY_LE** — Displays records whose keys are less than or equal to the specified key.

4.1.11 Users and Groups

The users and groups graphical user interface classes allow a Java program to present lists of AS/400 users and groups in a graphical user interface.

The VUserList object class is available. It is a resource that represents a list of AS/400 users and groups for use in AS/400 panes. AS/400 panes and VUserList objects can be used together to present many views of the list and allow the user to select users and groups. Figure 133 shows the VUserList graphical user interface component.

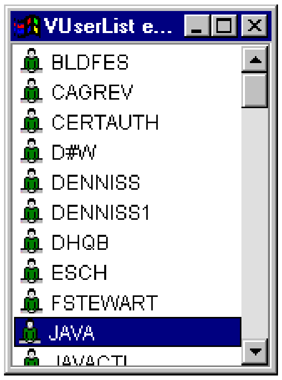


Figure 133. VUserList Graphical User Interface Component

4.2 JDBC Examples

This section explains how to build AS/400 client/server applications using the AS/400 Toolbox for Java GUI JDBC classes and VisualAge for Java 2.0.

4.2.1 Using the AS/400 Toolbox Classes in the VCE

The AS/400 Toolbox for Java classes are available in the VisualAge for Java Visual Composition editor. To use the Toolbox classes in the VCE, select the AS/400 Toolbox from the palette pulldown choice box. Hover help is provided so you can move the mouse pointer over the component to display its name.

Figure 134 shows the AS/400 Toolbox classes as they appear in the VisualAge for Java Visual Composition Editor.

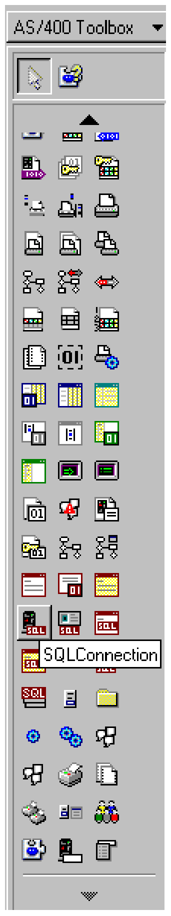


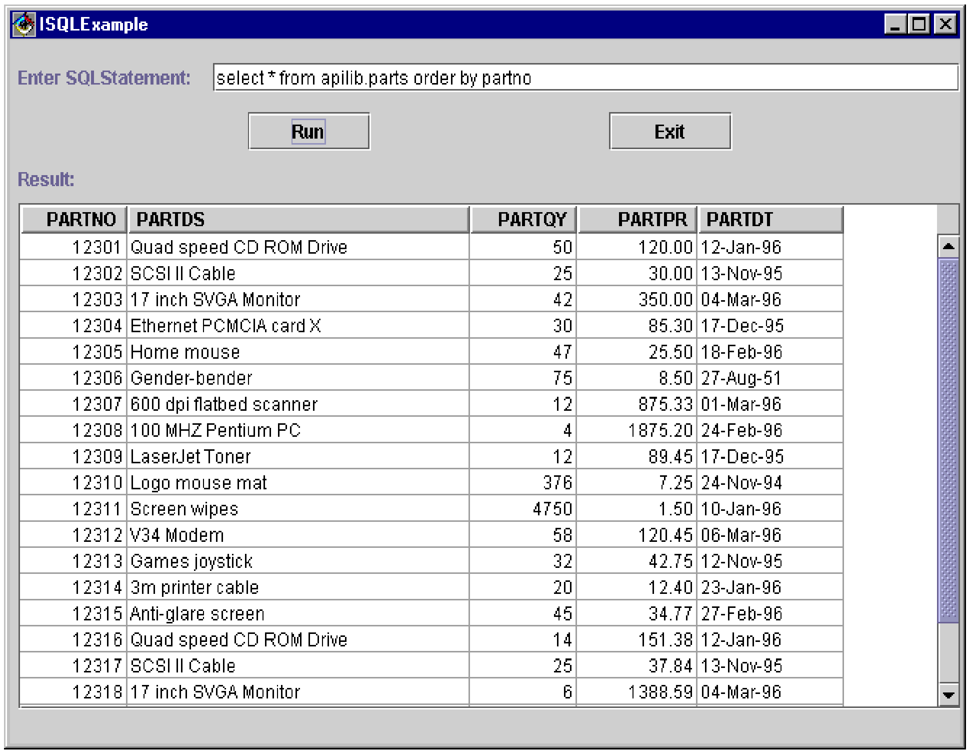
Figure 134. AS/400 Toolbox Classes in the VCE

4.2.2 SQLResultSetTablePane

The SQLResultSetTablePane presents the results of an SQL query in a table. In this example, we demonstrate building an AS/400 client/server application using the SQLResultSetTablePane. We also use an SQLConnection component to provide connectivity to the AS/400 system and an ErrorDialogAdapter to handle error conditions. All of these classes are available with the AS/400 Toolbox for Java. This example demonstrates building an AS/400 client/server application without writing code. The GUI classes do all the work.

4.2.2.1 SQLResultSetTablePane Application

The SQLResultSetTablePane application allows the end user to enter SQL statements that run against a DB2/400 database. The results are displayed in an SQLResultSetTable Pane. Figure 135 on page 189 shows the completed application.



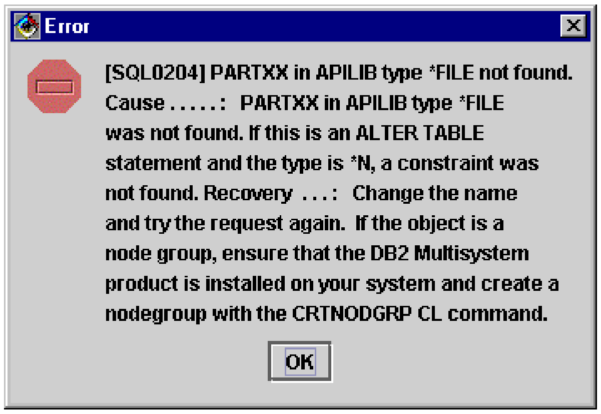
```
select * from apilib.parts order by partno
```

PARTNO	PARTDS	PARTQY	PARTPR	PARTDT
12301	Quad speed CD ROM Drive	50	120.00	12-Jan-96
12302	SCSI II Cable	25	30.00	13-Nov-95
12303	17 inch SVGA Monitor	42	350.00	04-Mar-96
12304	Ethernet PCMCIA card X	30	85.30	17-Dec-95
12305	Home mouse	47	25.50	18-Feb-96
12306	Gender-bender	75	8.50	27-Aug-51
12307	600 dpi flatbed scanner	12	875.33	01-Mar-96
12308	100 MHZ Pentium PC	4	1875.20	24-Feb-96
12309	LaserJet Toner	12	89.45	17-Dec-95
12310	Logo mouse mat	376	7.25	24-Nov-94
12311	Screen wipes	4750	1.50	10-Jan-96
12312	V34 Modem	58	120.45	06-Mar-96
12313	Games joystick	32	42.75	12-Nov-95
12314	3m printer cable	20	12.40	23-Jan-96
12315	Anti-glare screen	45	34.77	27-Feb-96
12316	Quad speed CD ROM Drive	14	151.38	12-Jan-96
12317	SCSI II Cable	25	37.84	13-Nov-95
12318	17 inch SVGA Monitor	6	1388.59	04-Mar-96

Figure 135. SQLResultSetTablePane Example

4.2.2.2 Error Handling

We use an ErrorDialogAdapter as a Listener to handle and display error conditions. In this case, we entered an SQL statement which could not be executed on the AS/400 system. The resulting error condition is displayed in a dialog box.



```
[SQL0204] PARTXX in APILIB type *FILE not found.
Cause . . . . . :   PARTXX in APILIB type *FILE
was not found. If this is an ALTER TABLE
statement and the type is *N, a constraint was
not found. Recovery  . . . :   Change the name
and try the request again.  If the object is a
node group, ensure that the DB2 Multisystem
product is installed on your system and create a
nodegroup with the CRTNODGRP CL command.
```

Figure 136. ErrorDialogAdapter Dialog Box

4.2.2.3 Building the Application

This section covers building this application using VisualAge for Java 2.0. We use the Visual Composition Editor (VCE) to construct the application from Toolbox classes.

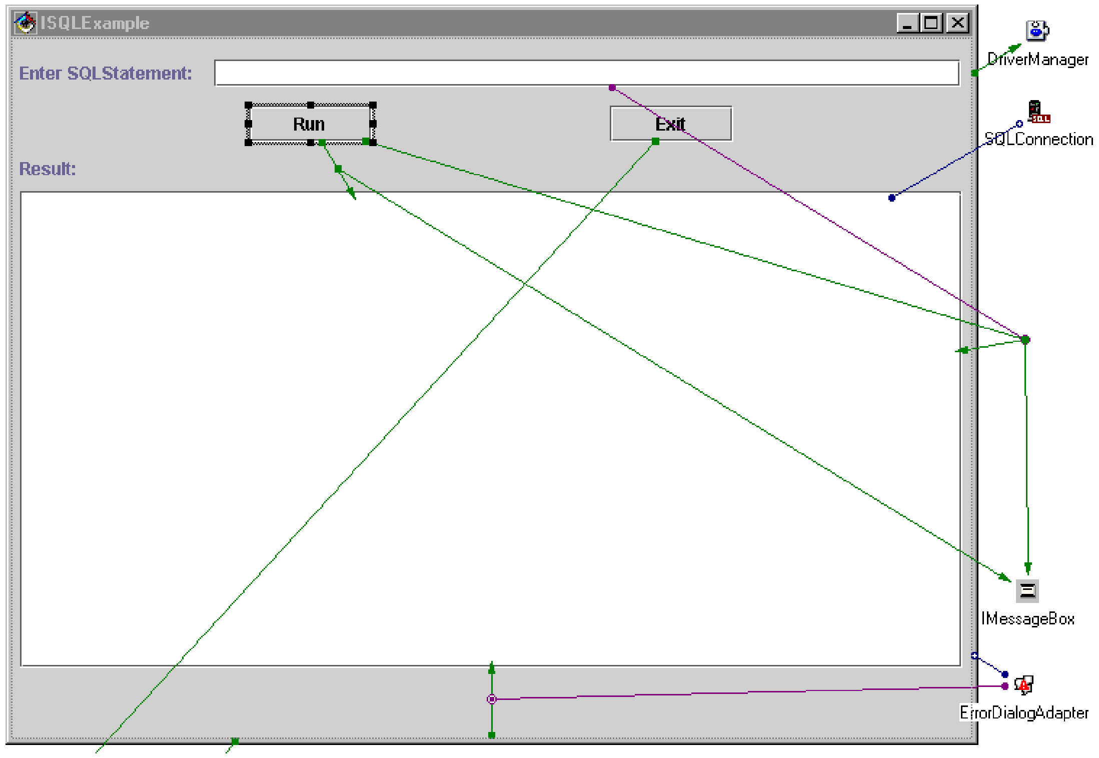


Figure 137. Building the Application with the VisualAge for Java VCE

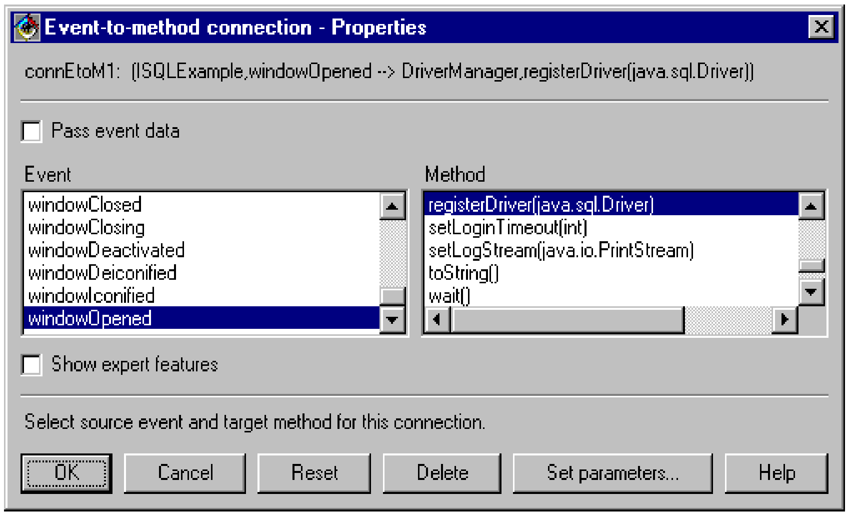
Figure 137 shows the application in the VisualAge for Java Visual Composition Editor. We use one visual and four non-visual classes to build the application. The visual component is the SQLResultSetTablePane that we place on the frame. It displays the results of an SQL statement entered in the TextField at the top of the Frame. The DriverManager class is used to load the proper JDBC driver. The SQLConnection class is used to connect to the AS/400 system. The IMessageBox is used to display any application-defined messages. The ErrorDialogAdapter displays any errors generated by the SQLResultSetTablePane.

4.2.2.4 Registering the Driver Manager

All JDBC graphical user interface classes communicate with the database using a JDBC driver. The JDBC driver must be registered with the JDBC driver manager for any of these classes to work. The following code example registers the AS/400 Toolbox for Java JDBC driver:

```
// Register the JDBC driver
DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver ());
```

In this example, we drop a DriverManager object, from the java.sql package, outside the frame and use the windowOpened event of the frame to register the proper JDBC driver with the DriverManager object. Figure 138 on page 191 shows the connection that is made.



connEtoM1: (ISQLExample,windowOpened --> DriverManager.registerDriver(java.sql.Driver))

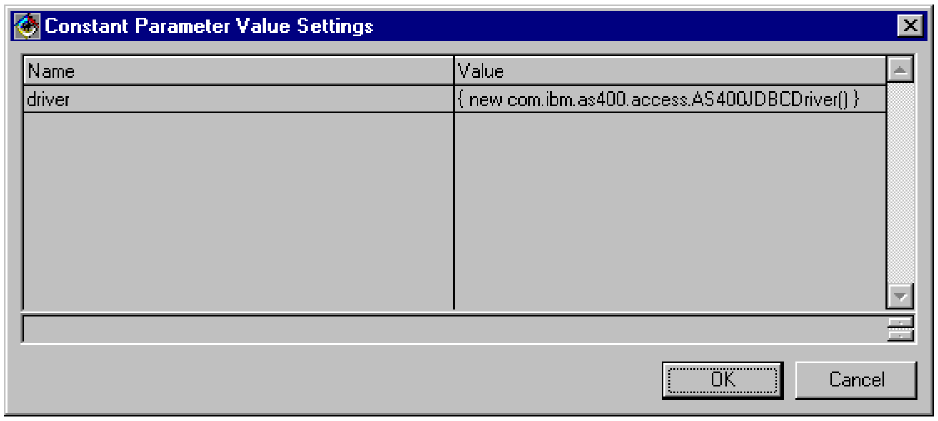
Event: windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified, windowOpened (selected)

Method: registerDriver(java.sql.Driver) (selected), setLoginTimeout(int), setLogStream(java.io.PrintStream), toString(), wait()

Select source event and target method for this connection.

Figure 138. JDBC registerDriver

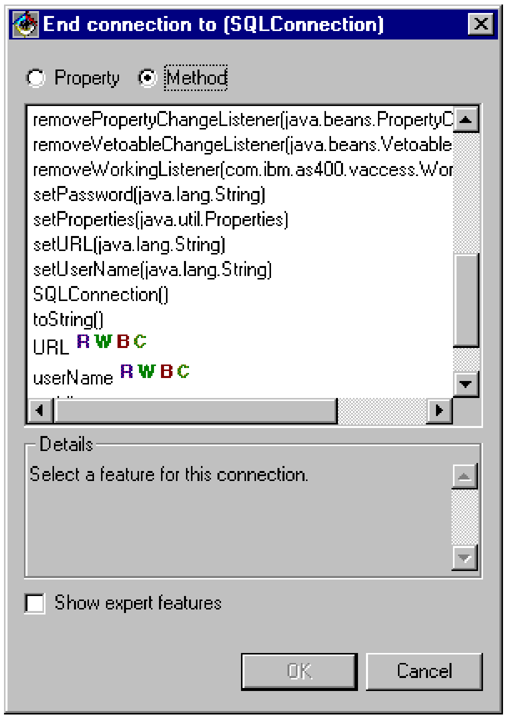
In the VCE, we click on the Set Parameters button and enter the name of the AS/400 Toolbox for Java JDBC driver as the name of the driver that we want to use as shown in Figure 139.



Name	Value
driver	{ new com.ibm.as400.access.AS400JDBCDriver() }

Figure 139. Registering the AS/400 Toolbox JDBC Driver

We use the Toolbox SQLConnection class to handle the connection to the AS/400 system. It provides methods that allow you to obtain and set its properties. Figure 140 on page 192 shows the SQLConnection properties.

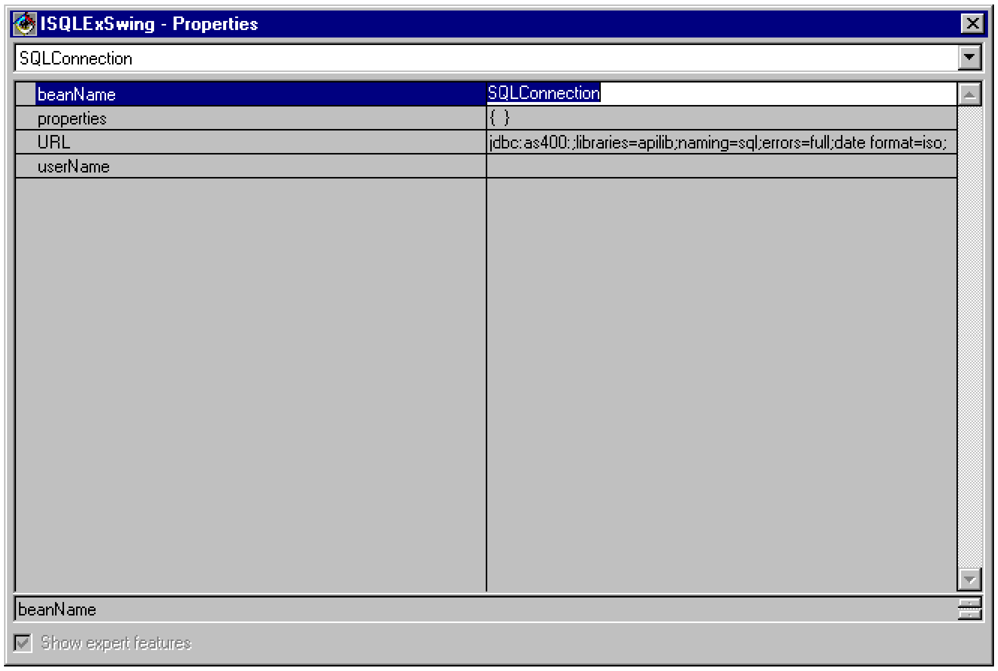


removePropertyChangeListener(java.beans.PropertyC
removeVetoableChangeListener(java.beans.Vetoable
removeWorkingListener(com.ibm.as400.vaccess.Wor
setPassword(java.lang.String)
setProperties(java.util.Properties)
setURL(java.lang.String)
setUserName(java.lang.String)
SQLConnection()
toString()
URL R W B C
userName R W B C

Select a feature for this connection.

Figure 140. SQLConnection Methods

As shown in Figure 141, we set the URL property in the VCE. We do not set the name of the AS/400 system, the user ID, or the password. The first time we try to run a SQL statement, we are prompted for this information.



beanName	SQLConnection
properties	{ }
URL	jdbc:as400:;libraries=apilib;naming=sql;errors=full;date format=iso;
userName	

beanName

Figure 141. Setting the SQLConnection URL Property

The SQLResultSetTablePane is placed on the frame. It is used to display the results of the query. The connection property of the SQLResultSetTable is set to the SQLConnection component that we created earlier. This is shown in Figure 142.

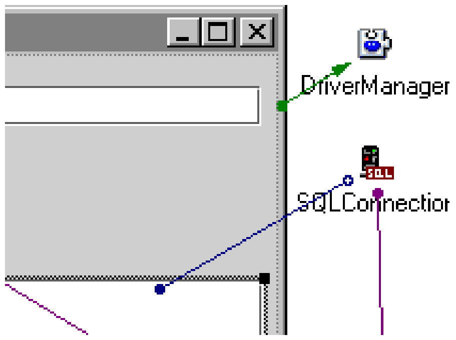
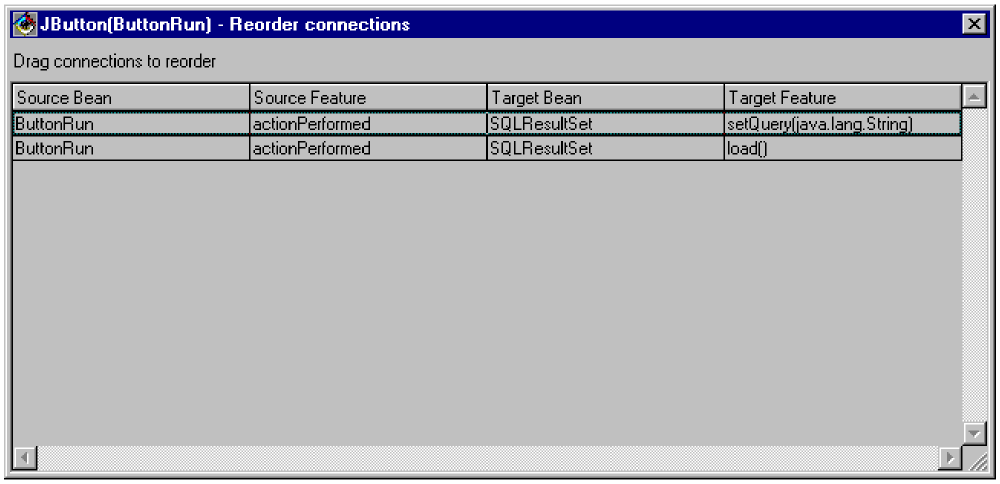


Figure 142. Setting the SQLResultSetTablePane Connection Property

The Run button is used to initiate the running of the query and display the results. It causes the setQuery method of the SQLResultSetTablePane to be executed with the TextField supplying the input parameter. The load method causes the query to run and fill the SQLResultSetTablePane with the rows from the database. Figure 143 shows the Run button events.

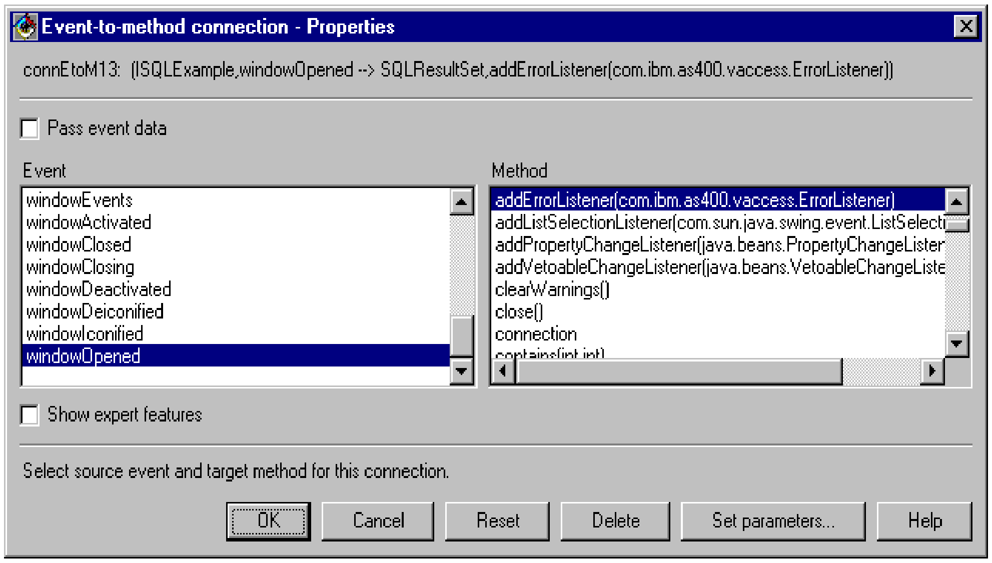

JButton(ButtonRun) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
ButtonRun	actionPerformed	SQLResultSet	setQuery(java.lang.String)
ButtonRun	actionPerformed	SQLResultSet	load()

Figure 143. Run Button Events

The final component to add is the ErrorDialogAdapter to handle error conditions for our application. As shown in Figure 144 on page 194, we use the windowOpened event to add an ErrorListener. We pass in the ErrorDialogAdapter as a parameter to the ErrorListener.


Event-to-method connection - Properties

connEtoM13: (ISQLExample,windowOpened --> SQLResultSet,addErrorListener(com.ibm.as400.vaccess.ErrorListener))

Event	Method
windowEvents	addErrorListener(com.ibm.as400.vaccess.ErrorListener)
windowActivated	addListSelectionListener(com.sun.java.swing.event.ListSelecti
windowClosed	addPropertyChangeListener(java.beans.PropertyChangeLister
windowClosing	addVetoableChangeListener(java.beans.VetoableChangeListe
windowDeactivated	clearWarnings()
windowDeiconified	close()
windowIconified	connection
windowOpened	contains(int,int)

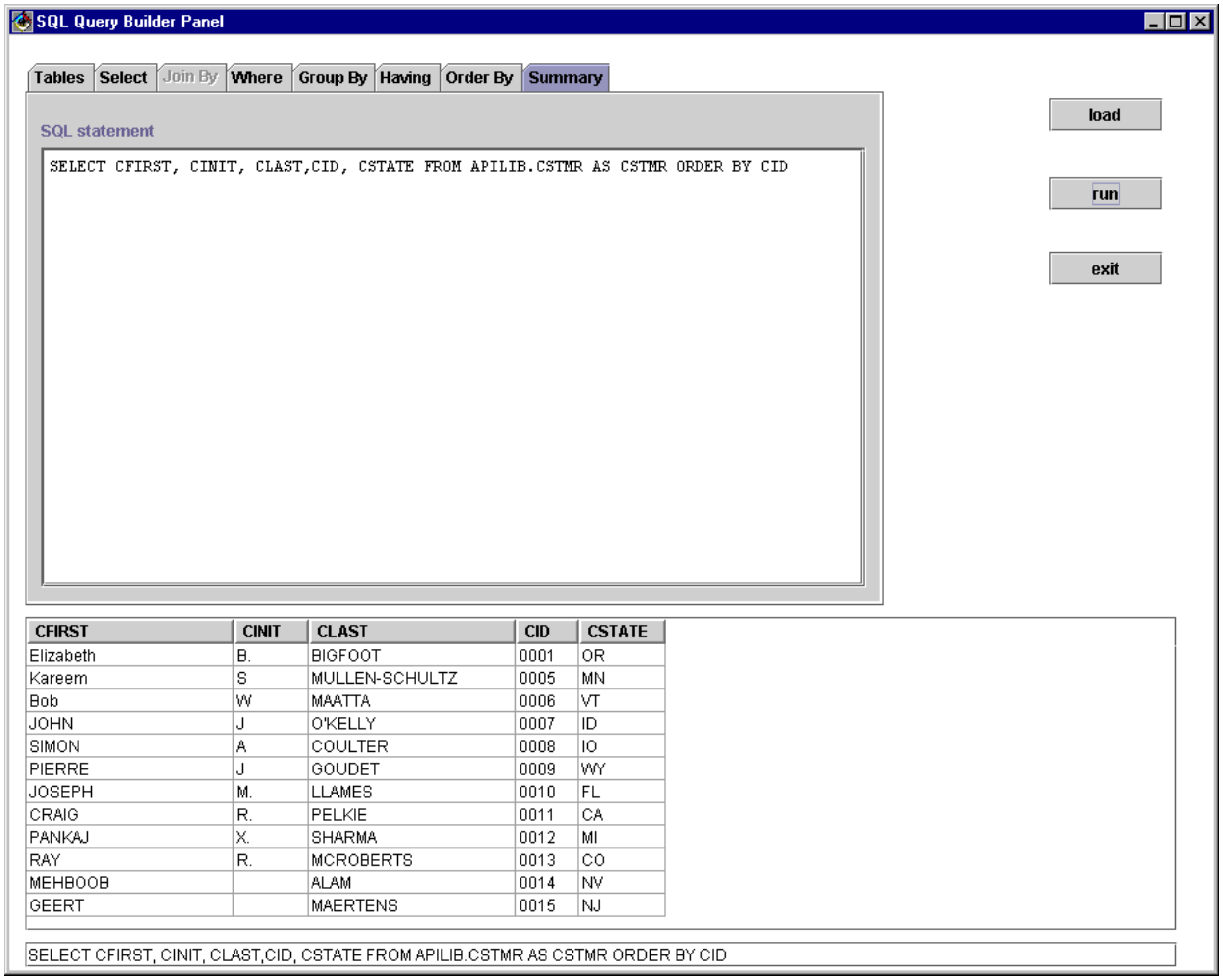
Select source event and target method for this connection.

Figure 144. Adding an ErrorListener

This completes the application. We have created an AS/400 client/server application that allows us to use JDBC to execute SQL statements against DB2/400 databases. We use a Listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

4.2.3 SQLQueryBuilderPane

The SQLQueryBuilderPane presents an interactive tool for dynamically building an SQL queries. You can use the SQLQueryBuilderPane to build and execute SQL statements against a DB2/400 database. It allows you to select the table that you want to use. It retrieves the table columns and displays them. You can choose which columns you want, selection criteria for the Where clause, and how to group or order the information. As you build the SQL statement, you can use the Summary tab to display the statement. When you are satisfied with the statement, you can execute it. The example shown in Figure 145 on page 195 displays the results in a SQLResultSetTablePane.



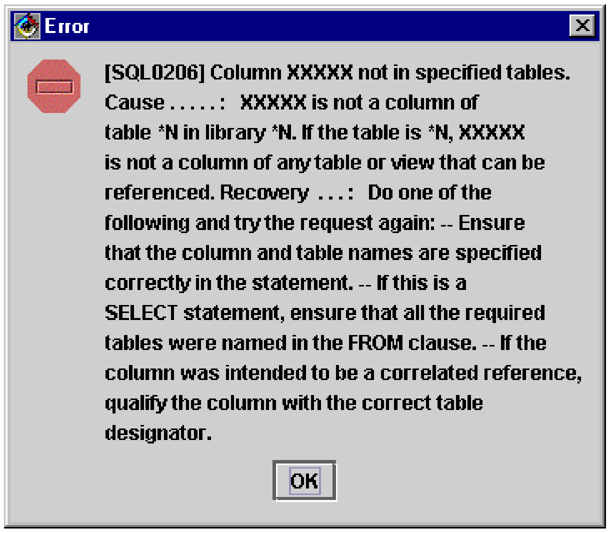
```
SELECT CFIRST, CINIT, CLAST,CID, CSTATE FROM APILIB.CSTMR AS CSTMR ORDER BY CID
```

CFIRST	CINIT	CLAST	CID	CSTATE
Elizabeth	B.	BIGFOOT	0001	OR
Kareem	S	MULLEN-SCHULTZ	0005	MN
Bob	W	MAATTA	0006	VT
JOHN	J	O'KELLY	0007	ID
SIMON	A	COULTER	0008	IO
PIERRE	J	GOUDET	0009	WY
JOSEPH	M.	LLAMES	0010	FL
CRAIG	R.	PELKIE	0011	CA
PANKAJ	X.	SHARMA	0012	MI
RAY	R.	MCROBERTS	0013	CO
MEHBOOB		ALAM	0014	NV
GEERT		MAERTENS	0015	NJ

```
SELECT CFIRST, CINIT, CLAST,CID, CSTATE FROM APILIB.CSTMR AS CSTMR ORDER BY CID
```

Figure 145. SQLQueryBuilderPane Example

We use an ErrorDialogAdapter to display any error conditions. The SQLQueryBuilderPane allows you to modify the SQL statement that you are building. For example, if you enter an invalid column name, the dialog box shown in Figure 146 on page 196 appears when you execute the statement.



[SQL0206] Column XXXXX not in specified tables. Cause . . . . . : XXXXX is not a column of table *N in library *N. If the table is *N, XXXXX is not a column of any table or view that can be referenced. Recovery . . . : Do one of the following and try the request again: -- Ensure that the column and table names are specified correctly in the statement. -- If this is a SELECT statement, ensure that all the required tables were named in the FROM clause. -- If the column was intended to be a correlated reference, qualify the column with the correct table designator.

Figure 146. SQLQueryBuilderPane Error Dialog

As shown in Figure 147 on page 197, this example was created using the VisualAge for Java Visual Composition Editor. We use two visual classes:

- **SQLQueryBuilderPane** — To build the SQL statements
- **SQLResultSetTablePane** — To execute the statement and display the results

We use three non-visual classes: DriverManager from the java.sql package, SQLConnection, and ErrorDialogAdapter.

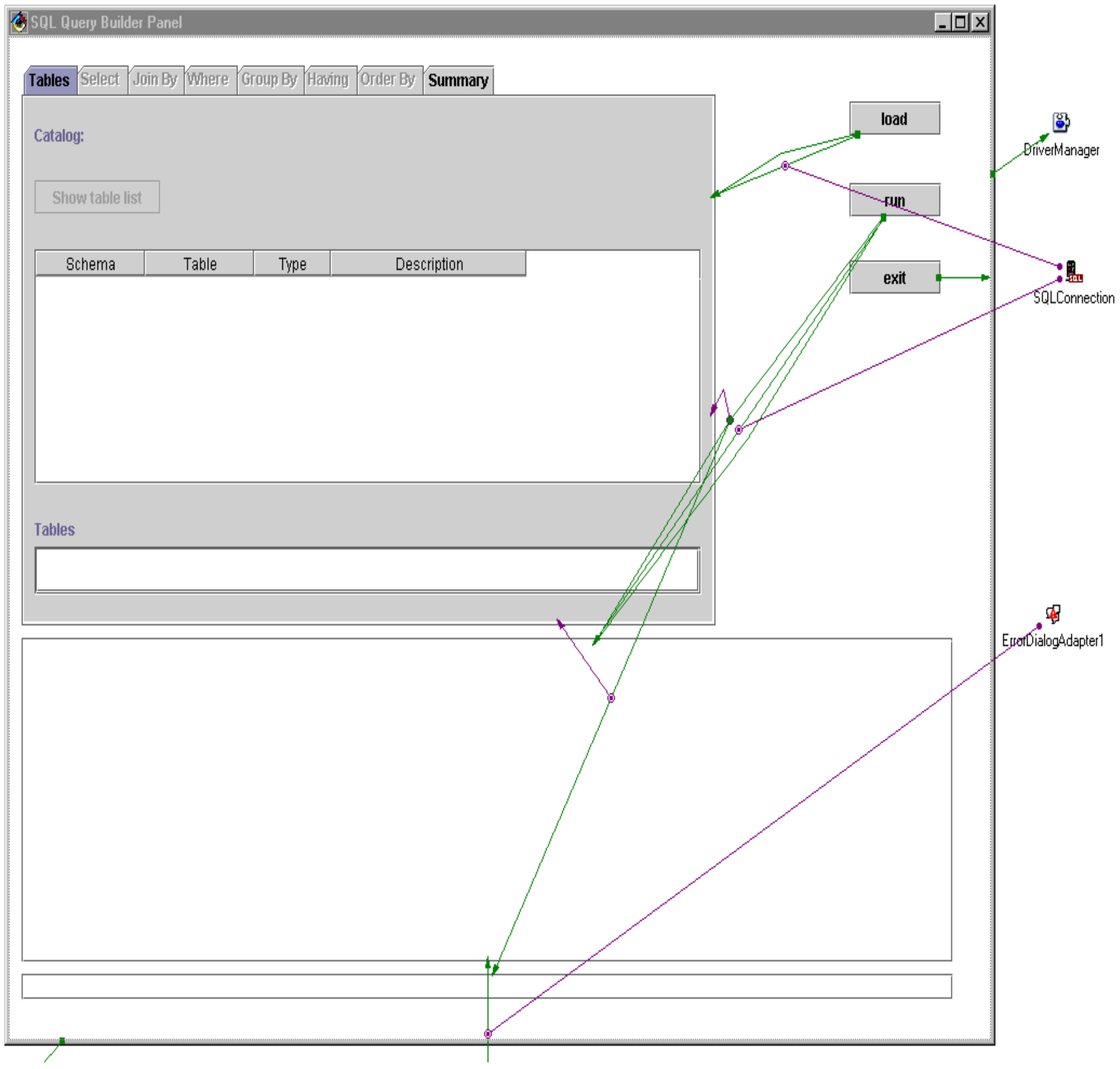
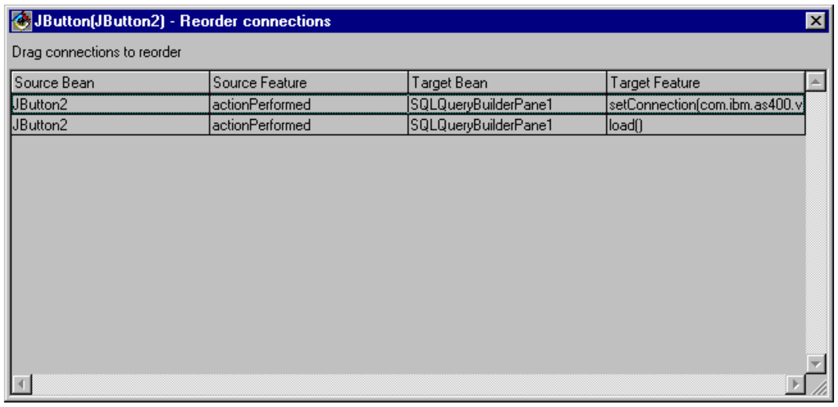


Figure 147. SQLQueryBuilder Example in the VisualAge for Java VCE

Processing for the non-visual classes is exactly the same as shown in the SQLResultSetTablePane example discussed earlier in this chapter. Please see Section 4.2.2.1, “SQLResultSetTablePane Application” on page 188, for details.

The Load button controls the SQLQueryBuilderPane. We use the setConnection method to set the SQL connection. We pass in the SQLConnection object as the parameter for the connection to use. We use the load method to load the database schemas and tables with which we want to work. Figure 148 on page 198 shows the load button connections.



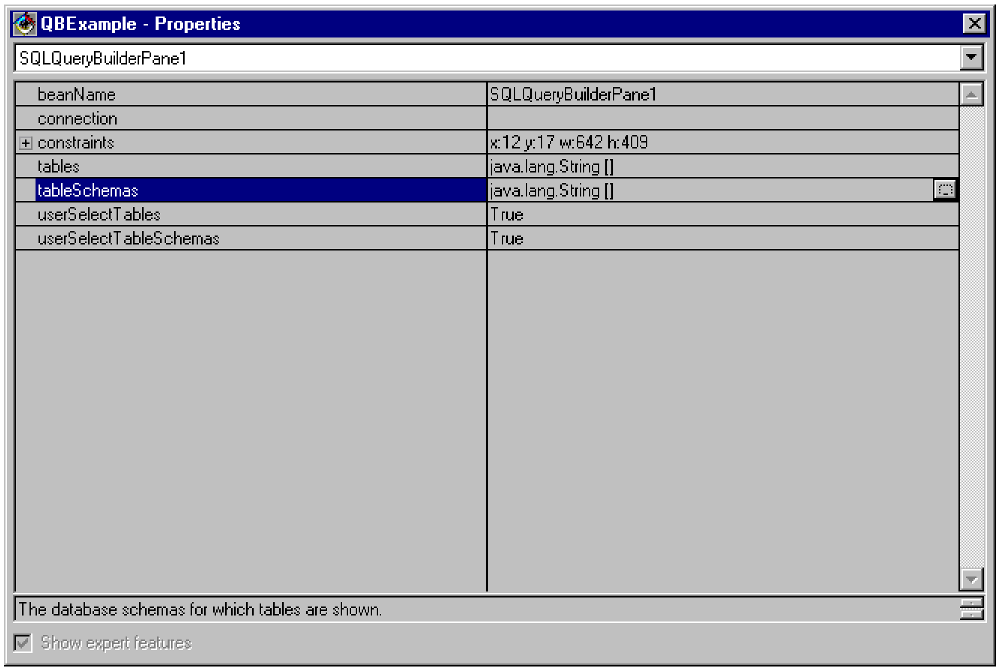
JButton(JButton2) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
JButton2	actionPerformed	SQLQueryBuilderPane1	setConnection(com.ibm.as400.v
JButton2	actionPerformed	SQLQueryBuilderPane1	load()

Figure 148. Load Button Connections

Figure 149 shows the SQLQueryBuilderPane properties dialog box. We set the tableSchema property value to APILIB. This causes all the tables in the library named APILIB to be shown.



QBExample - Properties

SQLQueryBuilderPane1

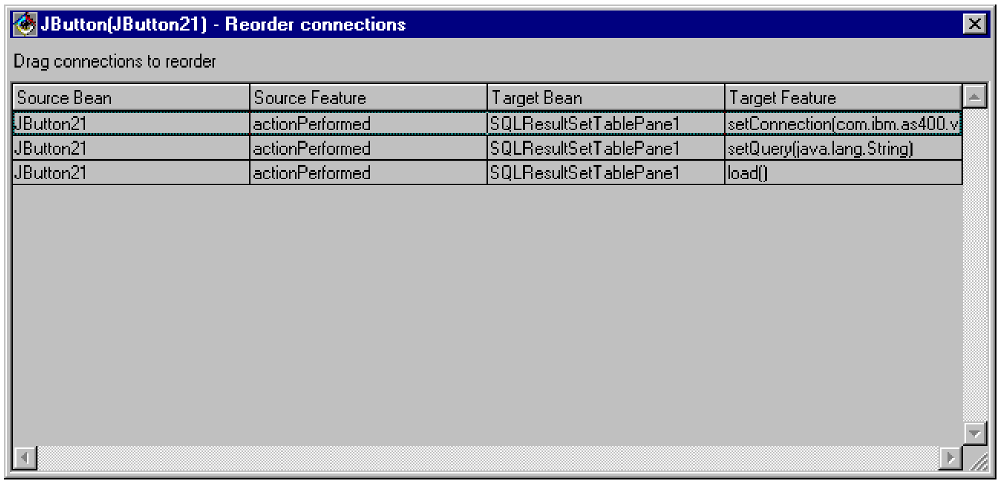
beanName	SQLQueryBuilderPane1
connection	
constraints	x:12 y:17 w:642 h:409
tables	java.lang.String []
tableSchemas	java.lang.String []
userSelectTables	True
userSelectTableSchemas	True

The database schemas for which tables are shown.

Show expert features

Figure 149. Setting the Tables Schema

The Run button controls the SQLResultSetTablePane. First, we set the connection property with the SQLConnect object. We use the getQuery() method of the SQLQueryBuilder pane to retrieve the query that was built and use it as the parameter for the setQuery method of the SQLResultSetTable Pane. We then use the load() method to cause the query to run and the results to be displayed. Figure 150 on page 199 shows the Run button connections.



JButton(JButton21) - Reorder connections

Drag connections to reorder

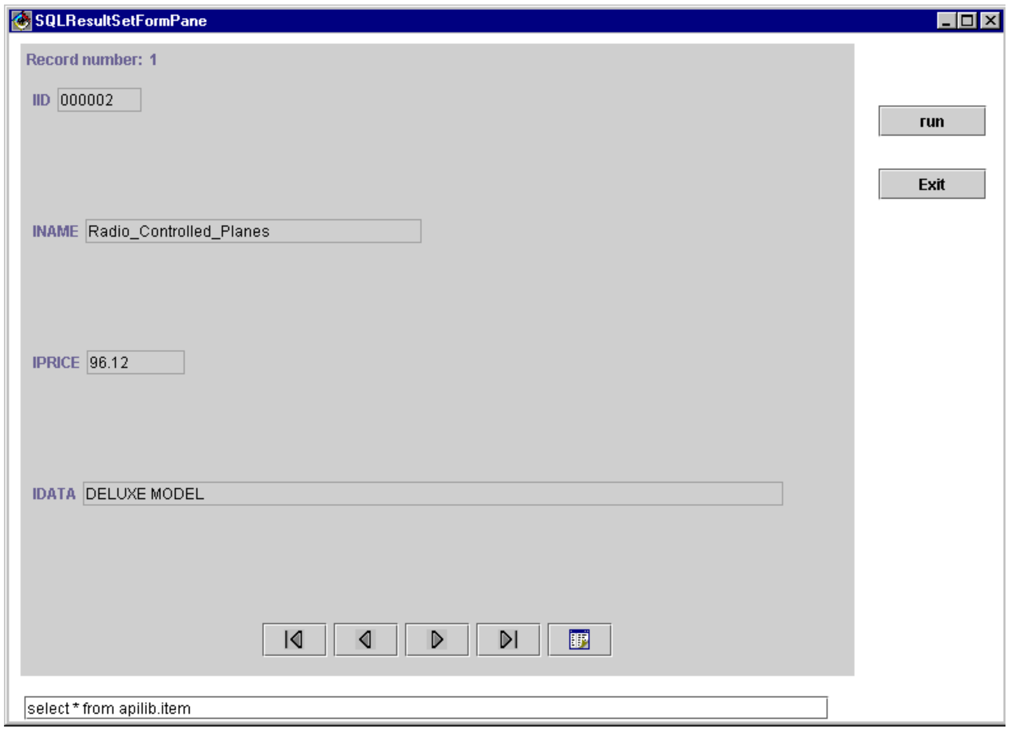
Source Bean	Source Feature	Target Bean	Target Feature
JButton21	actionPerformed	SQLResultSetTablePane1	setConnection(com.ibm.as400.v
JButton21	actionPerformed	SQLResultSetTablePane1	setQuery(java.lang.String)
JButton21	actionPerformed	SQLResultSetTablePane1	load()

Figure 150. Run Button Connections

This completes the application. We have created an AS/400 client/server application that allows us to use JDBC to interactively build and execute SQL statements against DB2/400 databases. We use a Listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

4.2.4 SQLResultSetFormPane

The SQLResultSetFormPane presents the results of a query in a form. The form provides controls that allow you to scroll forward and backward through the result set returned by the query. Figure 151 shows the completed example.



SQLResultSetFormPane

Record number: 1

IID 000002

INAME Radio_Controlled_Planes

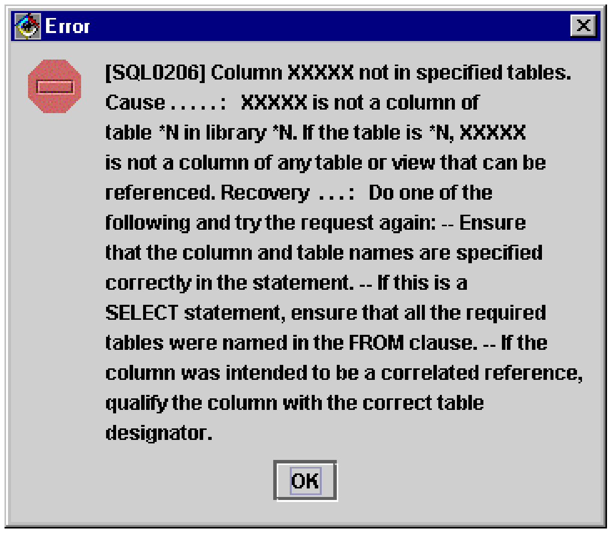
IPRICE 96.12

IDATA DELUXE MODEL

select * from apilib.item

Figure 151. SQLResultSetFormPane Example

We use a ErrorDialog Adapter to display any error conditions. If you entered an invalid column name, for example, the dialog box shown in Figure 152 appears.



[SQL0206] Column XXXXX not in specified tables. Cause . . . . . : XXXXX is not a column of table *N in library *N. If the table is *N, XXXXX is not a column of any table or view that can be referenced. Recovery . . . : Do one of the following and try the request again: -- Ensure that the column and table names are specified correctly in the statement. -- If this is a SELECT statement, ensure that all the required tables were named in the FROM clause. -- If the column was intended to be a correlated reference, qualify the column with the correct table designator.

Figure 152. SQLResultSetFormPane Example Error

As shown in Figure 153 on page 201, this example was created using the VisualAge for Java Visual Composition Editor. We use a visual component, the SQLResultSetFormPane, to execute an SQL statement and display the results. We use three non-visual classes: DriverManager from the java.sql package, SQLConnection, and ErrorDialogAdapter.

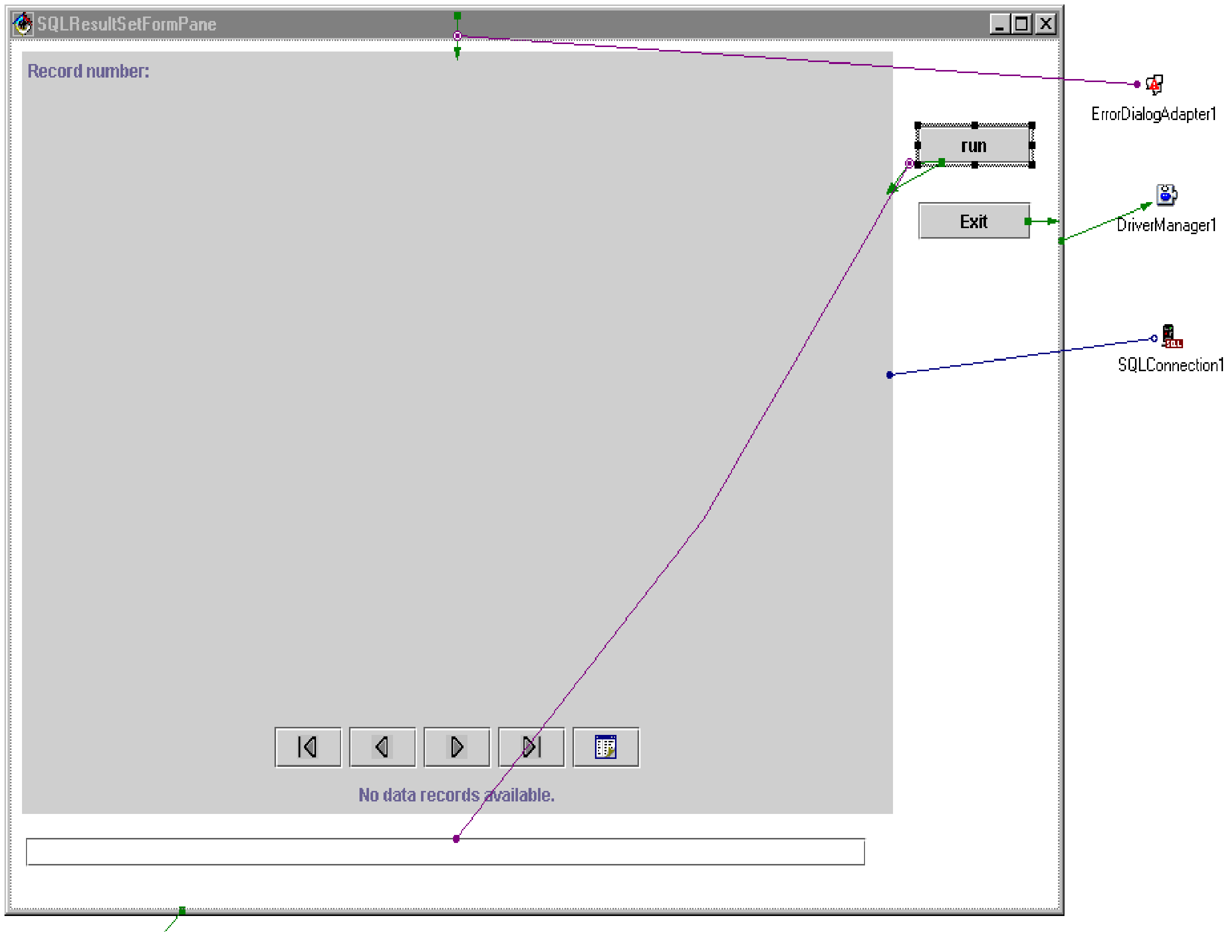
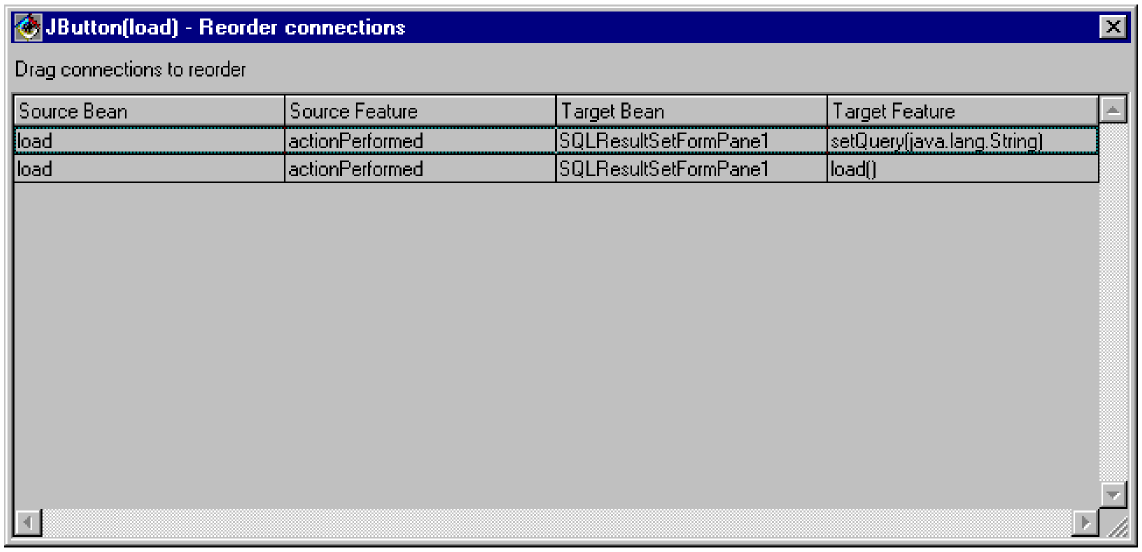


Figure 153. SQLResultSetForm in the VCE

Processing for the non-visual classes is exactly the same as shown in the SQLResultSetTablePane example discussed earlier in this chapter. Please see Section 4.2.2.1, “SQLResultSetTablePane Application” on page 188, for details.

The Run button controls the processing of the SQLResultSetFormPane processing. Figure 154 on page 202 shows the Run button connections.


JButton(load) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
load	actionPerformed	SQLResultSetFormPane1	setQuery(java.lang.String)
load	actionPerformed	SQLResultSetFormPane1	load()

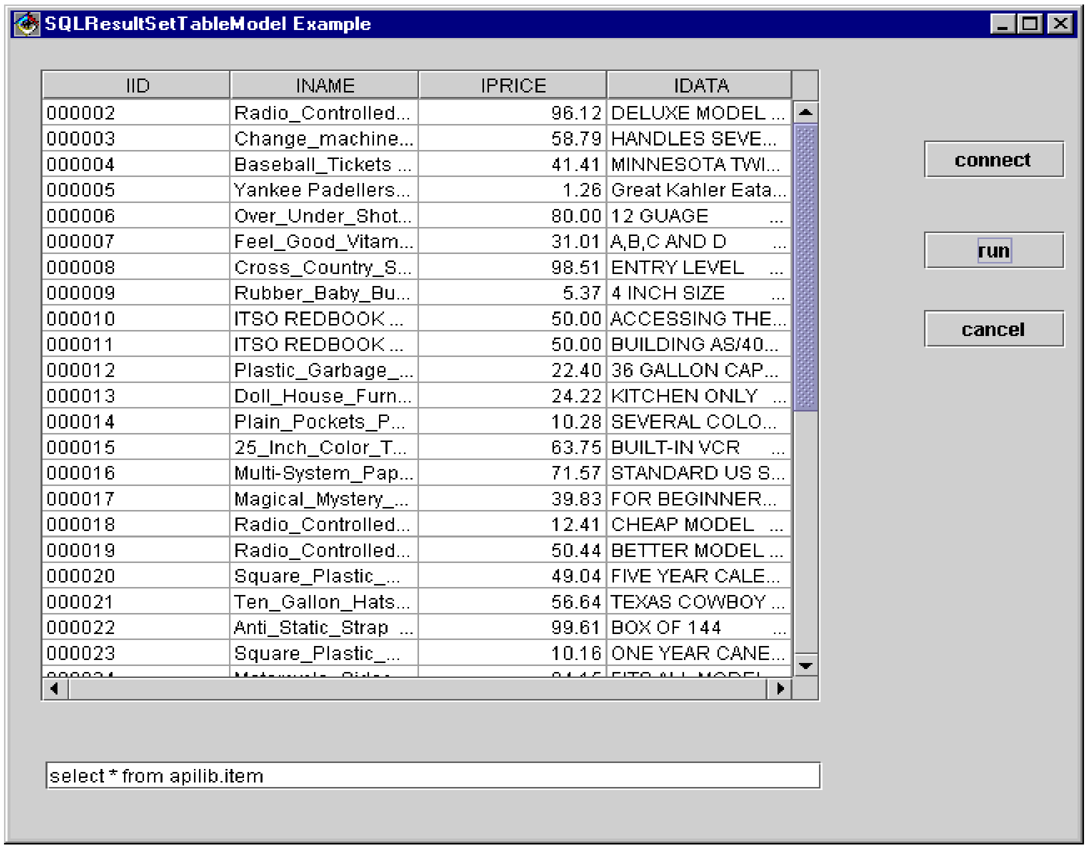
Figure 154. Run Button for the SQLResultSetFormPane

The setQuery method is used to set the SQL statement to be executed. The value from the TextField is used as input. The execution of the load method causes the SQL statement to be executed. The results are displayed in the SQLResultSetFormPane.

This completes the application. We created an AS/400 client/server application, which allows us to use JDBC to interactively build and execute SQL statements against DB2/400 databases. We use a Listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

4.2.5 SQLResultSetModel

The SQLResultSetModel class manages the results of an SQL query in a table. The SQLResultSetModel class allows you to have more control over the classes that you are using. In this example, we use the class in conjunction with a Swing JScrollPane. The SQLResultSetModel controls the execution and results of the SQL statement, but the JScrollPane actually displays the results. This class allows you to keep the SQL processing separate from the actual display. Figure 155 on page 203 shows the completed example.

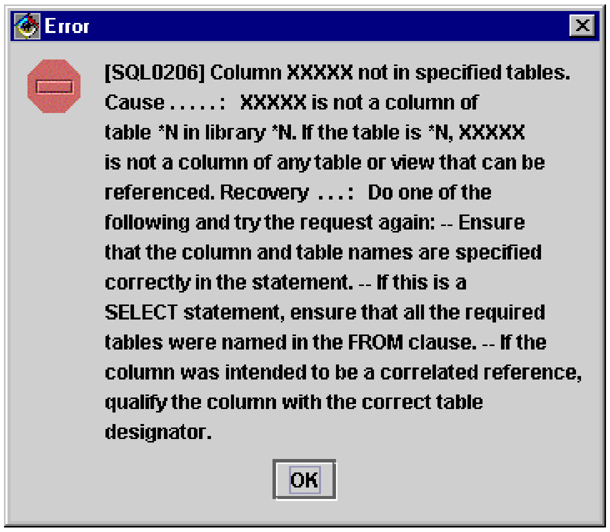


IID	INAME	IPRICE	IDATA
000002	Radio_Controlled...	96.12	DELUXE MODEL ...
000003	Change_machine...	58.79	HANDLES SEVE...
000004	Baseball_Tickets ...	41.41	MINNESOTA TWI...
000005	Yankee Padellers...	1.26	Great Kahler Eata...
000006	Over_Under_Shot...	80.00	12 GUAGE ...
000007	Feel_Good_Vitam...	31.01	A,B,C AND D ...
000008	Cross_Country_S...	98.51	ENTRY LEVEL ...
000009	Rubber_Baby_Bu...	5.37	4 INCH SIZE ...
000010	ITSO REDBOOK ...	50.00	ACCESSING THE...
000011	ITSO REDBOOK ...	50.00	BUILDING AS/40...
000012	Plastic_Garbage_...	22.40	36 GALLON CAP...
000013	Doll_House_Furn...	24.22	KITCHEN ONLY ...
000014	Plain_Pockets_P...	10.28	SEVERAL COLO...
000015	25_Inch_Color_T...	63.75	BUILT-IN VCR ...
000016	Multi-System_Pap...	71.57	STANDARD US S...
000017	Magical_Mystery_...	39.83	FOR BEGINNER...
000018	Radio_Controlled...	12.41	CHEAP MODEL ...
000019	Radio_Controlled...	50.44	BETTER MODEL ...
000020	Square_Plastic_...	49.04	FIVE YEAR CALE...
000021	Ten_Gallon_Hats...	56.64	TEXAS COWBOY ...
000022	Anti_Static_Strap ...	99.61	BOX OF 144 ...
000023	Square_Plastic_...	10.16	ONE YEAR CANE...

select * from apilib.item

Figure 155. SQLResultSetTableModel Example

We use an ErrorDialog Adapter to display any error conditions. If you enter an invalid column name, for example, the dialog box shown in Figure 156 appears.



[SQL0206] Column XXXXX not in specified tables. Cause . . . . . : XXXXX is not a column of table *N in library *N. If the table is *N, XXXXX is not a column of any table or view that can be referenced. Recovery . . . : Do one of the following and try the request again: -- Ensure that the column and table names are specified correctly in the statement. -- If this is a SELECT statement, ensure that all the required tables were named in the FROM clause. -- If the column was intended to be a correlated reference, qualify the column with the correct table designator.

Figure 156. SQLResultSetTableModel Error Dialog

As shown in Figure 157 on page 204, this example was created using the VisualAge for Java Visual Composition Editor. We use a visual component, the Swing JScrollPane, to display the results. We use four non-visual classes:

DriverManager from the java.sql package, SQLResultSetTableModel, SQLConnection, and ErrorDialogAdapter.

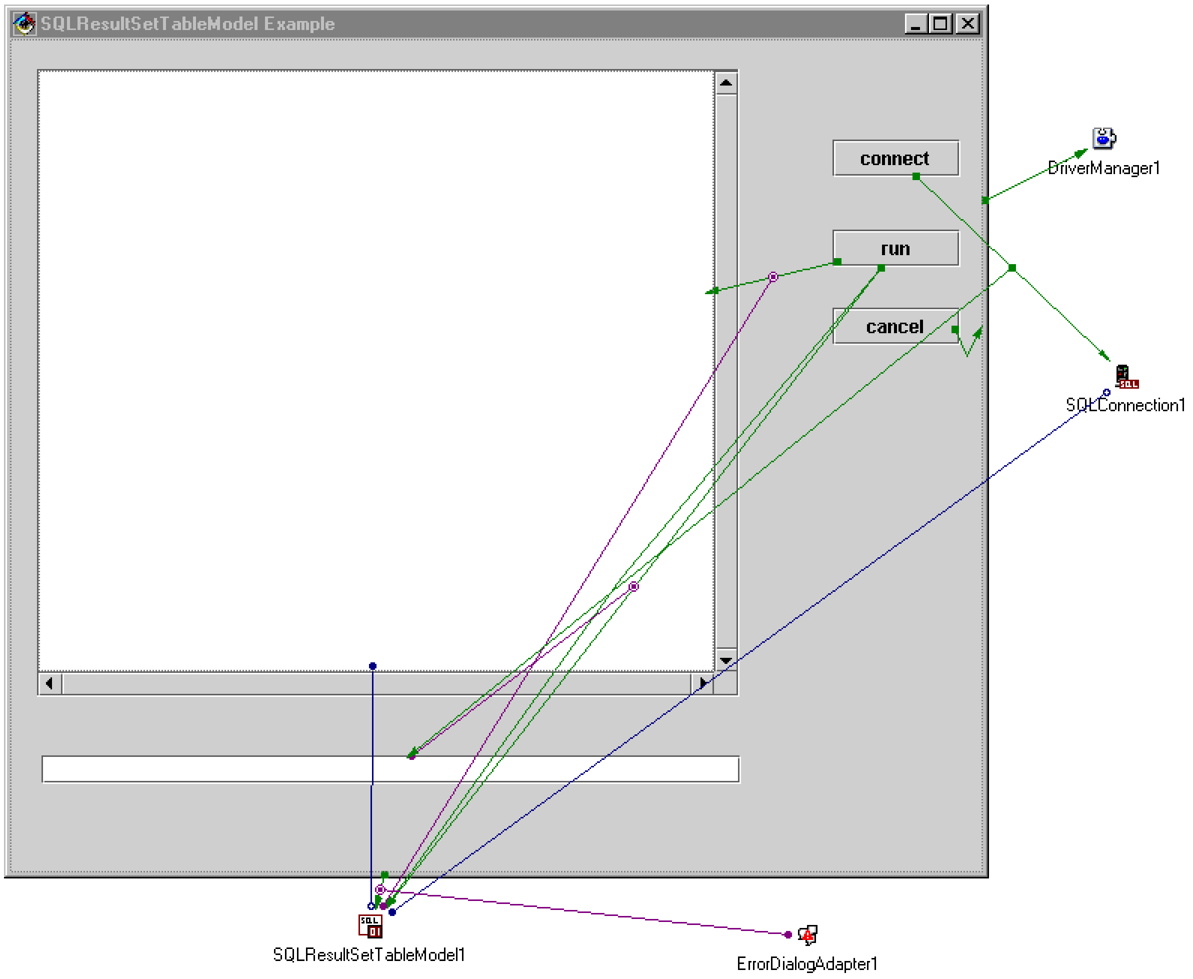
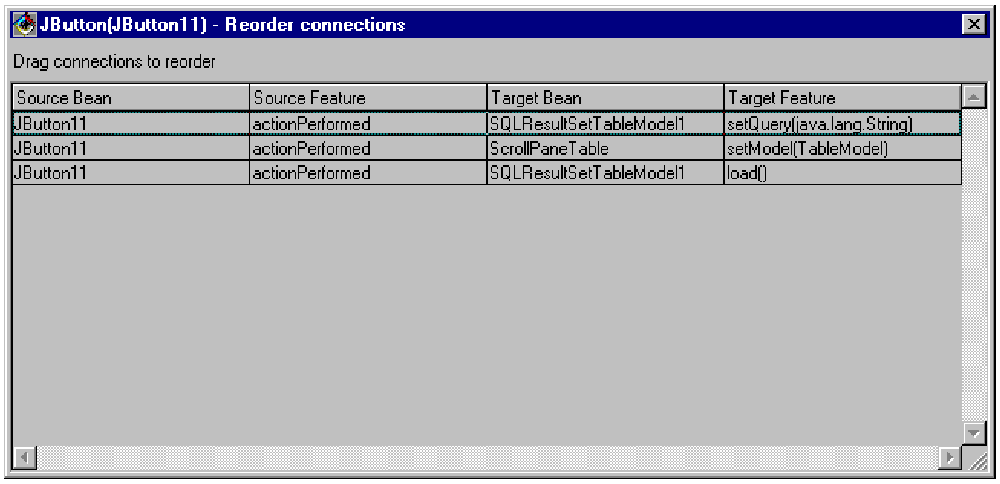


Figure 157. SQLResultSetModel in the VCE

Processing for the DriverManager, SQLConnect, and the ErrorDialogAdapter is exactly the same as shown in the SQLResultSetTablePane example discussed earlier in this chapter. Please see Section 4.2.2.1, “SQLResultSetTablePane Application” on page 188, for details.

The Run button controls the processing of the SQLResultSetTableModel and the JScrollPane. Figure 158 on page 205 shows the Run button connections.


JButton(JButton11) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
JButton11	actionPerformed	SQLResultSetTableModel1	setQuery(java.lang.String)
JButton11	actionPerformed	ScrollPaneTable	setModel(TableModel)
JButton11	actionPerformed	SQLResultSetTableModel1	load()

Figure 158. SQLResultSetTableModel Example Run Button

The SQLResultSetTableModel setQuery method is used to set the SQL statement using the JTextField as input. The JScrollTable setModel method is used to set the TableModel property to the SQLResultSetTableModel. The load method of the SQLResultSetTableModel is executed to run the SQL statement and return the results. The results are displayed in the JScrollPane.

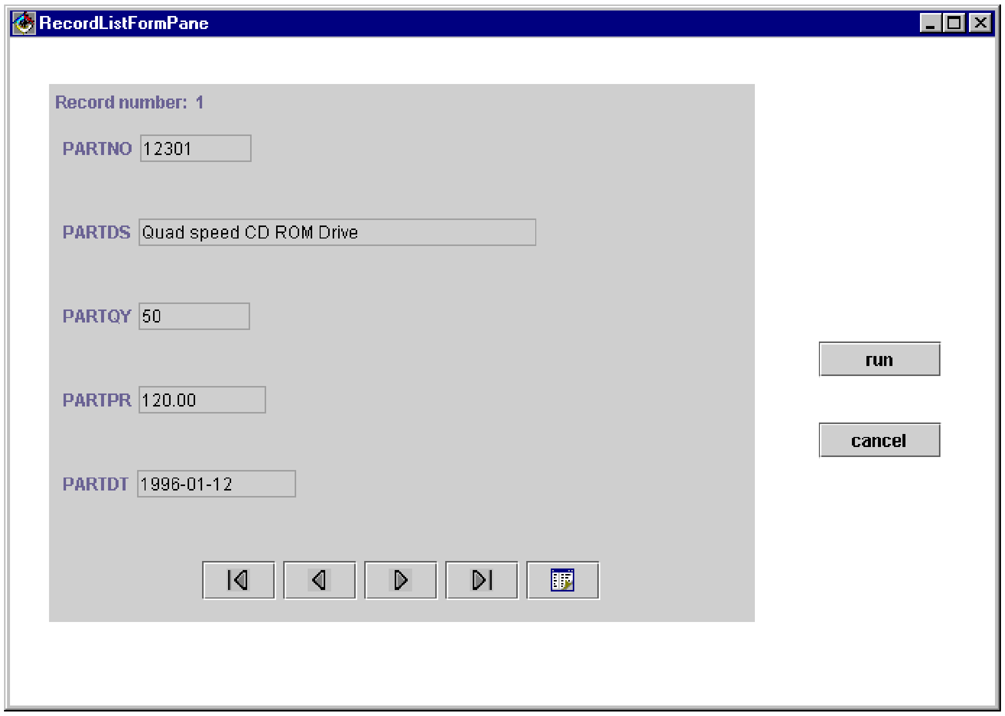
This completes the application. We created an AS/400 client/server application which allows us to use JDBC to interactively build and then execute SQL statements against DB2/400 databases. We use a Listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code. This application is similar to the SQLResultSetTablePane example in Section 4.2.2.1, “SQLResultSetTablePane Application” on page 188. The difference here is that we use a Swing component, which is not part of the AS/400 Toolbox for Java, to display the results.

4.3 Record Level Access GUI Examples

This section explains how to build AS/400 client/server applications using the AS/400 Toolbox for Java GUI Record Level Access classes and VisualAge for Java. In these examples, we access the database using the DDM Record Level Access interface.

4.3.1 RecordListFormPane

The RecordListFormPane class presents a list of records from an AS/400 file in a form. In this example, we use it to display the records from an AS/400 file named PARTS. Figure 159 on page 206 shows the completed example.


RecordListFormPane

Record number: 1

PARTNO 12301

PARTDS Quad speed CD ROM Drive

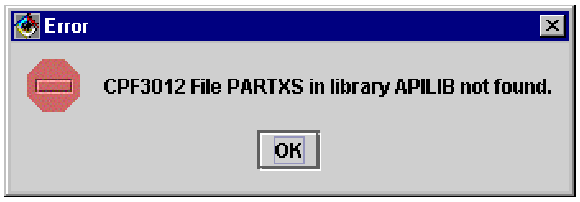
PARTQY 50

PARTPR 120.00

PARTDT 1996-01-12

Figure 159. RecordListFormPane Example

We use an ErrorDialog Adapter to view any error conditions. If you try to access a non-existent file, for example, the dialog box shown in Figure 160 appears.


Error

CPF3012 File PARTXS in library APILIB not found.

Figure 160. Record-Level Access Error Dialog

As shown in Figure 161 on page 207, this example was created using the VisualAge for Java Visual Composition Editor. We use three non-visual classes: AS400, AS400FileRecordDescription, and ErrorDialog Adapter. We use a RecordListFormPane to view the records retrieved from the AS/400 system.

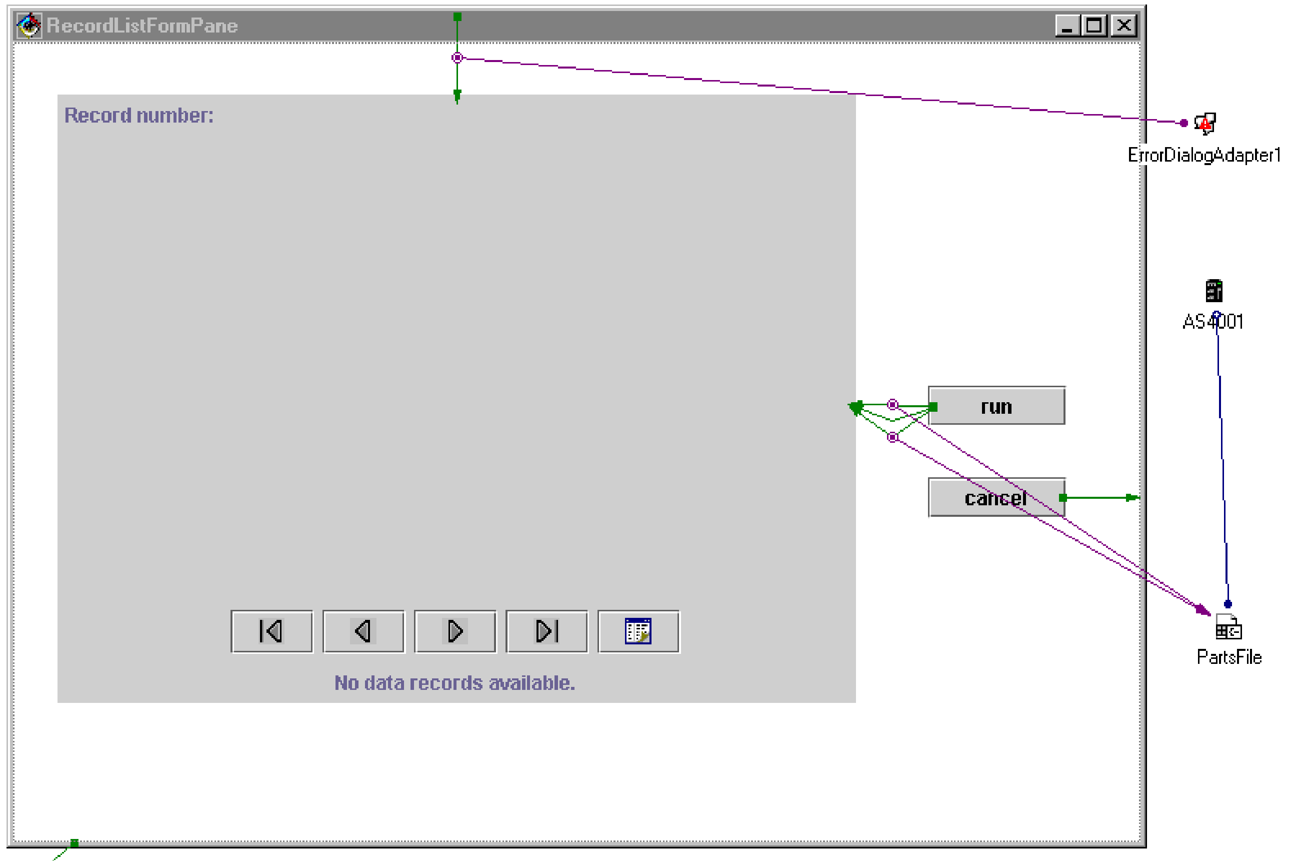
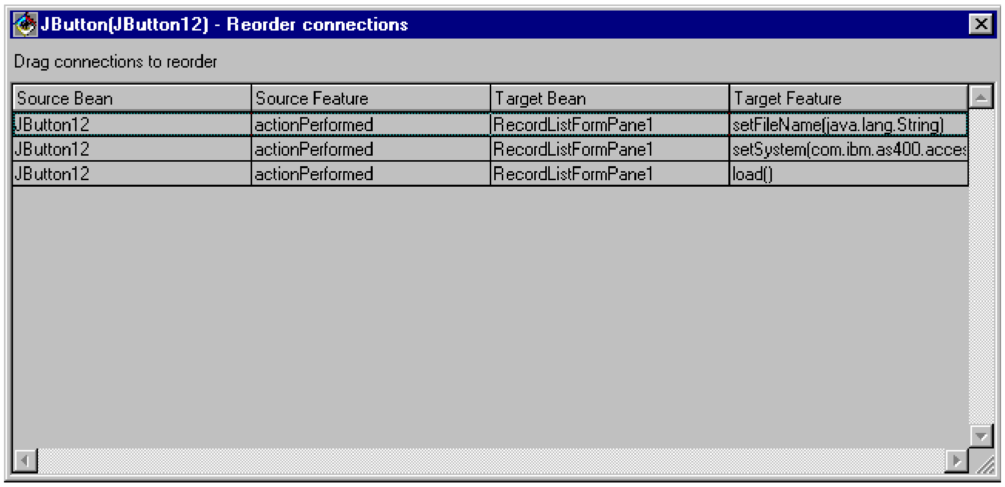


Figure 161. RecordListFormPane in the VCE

By dropping an AS400 object outside the frame in the VCE, we instantiate an AS400 object. We use the AS/400 for the connection to the AS/400 system. We do not set the AS400 properties for system name, user ID, or password. This causes a signon dialog to appear the first time we try to access the AS/400 system. We use the AS400FileRecordDescription object, which we name PartsFile, to control the name of the AS/400 file with which we want to work.

We set the path attribute to /QSYS.LIB/APILIB/PARTS.FILE. We use AS/400 IFS naming conventions to set the name of the file. The Run button controls the execution of the application. The Run button connections are shown in Figure 162 on page 208.


JButton(JButton12) - Reorder connections
Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
JButton12	actionPerformed	RecordListFormPane1	setFileName(java.lang.String)
JButton12	actionPerformed	RecordListFormPane1	setSystem(com.ibm.as400.acces
JButton12	actionPerformed	RecordListFormPane1	load()

Figure 162. Run Button Connections

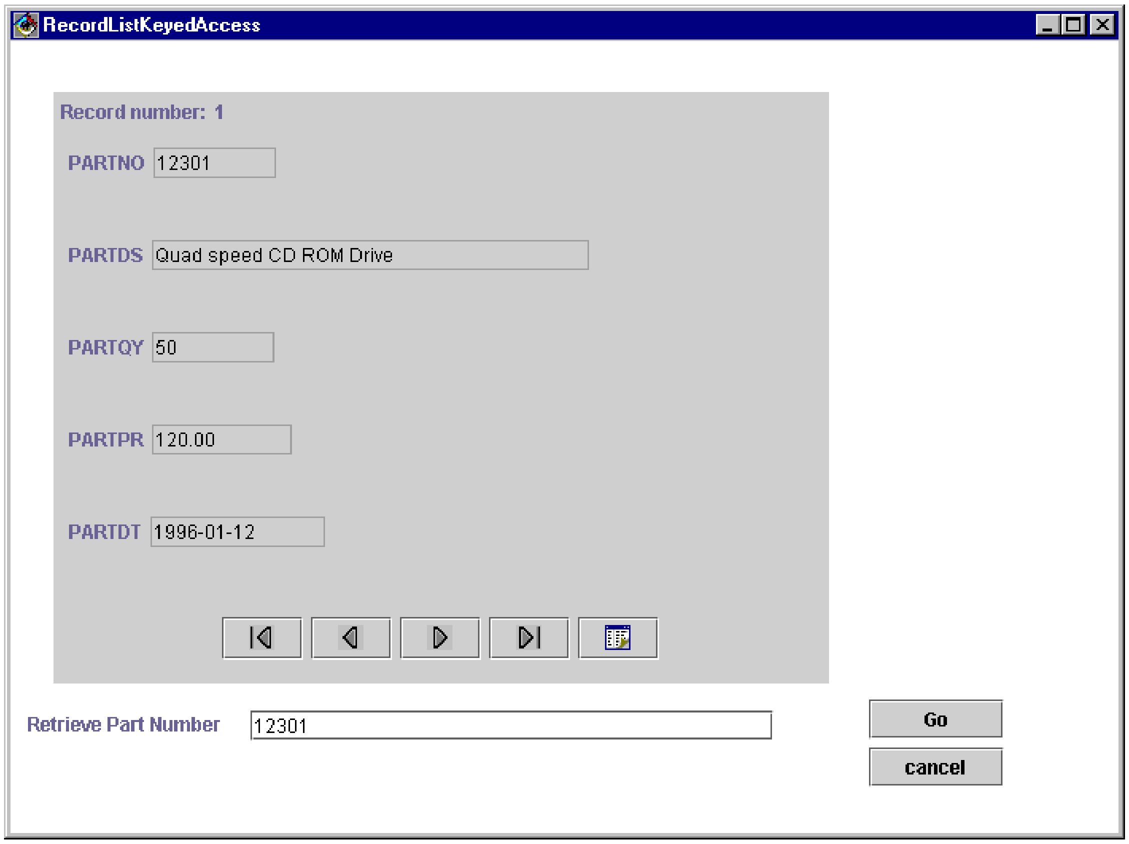
We use the setFileName method of the RecordListTablePane class to set the name of the file with which we want to work. We use the value from the AS400FileRecordDescription object. Next, we set the system name again using the value from the AS400FileRecordDescription object. Finally, we execute the load method to cause the records to be displayed.

The final component to add is the ErrorDialogAdapter to handle error conditions for our application. We use the windowOpened event to add an ErrorListener. We pass in the ErrorDialogAdapter as a parameter to the ErrorListener. This is done exactly the same as in the RecordListTablePane example.

This completes the application. We created an AS/400 client/server application that allows us to use Record Level Access to retrieve records from an AS/400 file. We use a Listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

4.3.2 RecordListFormPane Using the Keyed Access Example

In this example, we use keyed access to retrieve records by key and display them in a RecordListFormPane. Figure 163 on page 209 shows the completed example.


RecordListKeyedAccess

Record number: 1

PARTNO 12301

PARTDS Quad speed CD ROM Drive

PARTQY 50

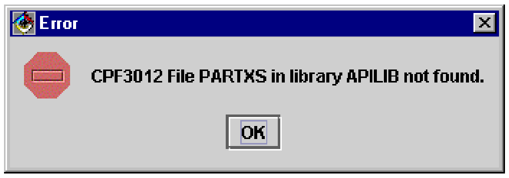
PARTPR 120.00

PARTDT 1996-01-12

Retrieve Part Number 12301

Figure 163. RecordListFormPane Example

We use an ErrorDialog Adapter to display any error conditions. If you try to access an non existent file, for example, the dialog box shown in Figure 164 appears.


Error

CPF3012 File PARTXS in library APILIB not found.

Figure 164. Record-Level Access Error Dialog

As shown in Figure 165 on page 210, this example was created using the VisualAge for Java Visual Composition Editor. We use three non-visual classes: AS400, AS400FileRecordDescription, and ErrorDialog Adapter. We use a RecordListFormPane to display the records retrieved from the AS/400 system.

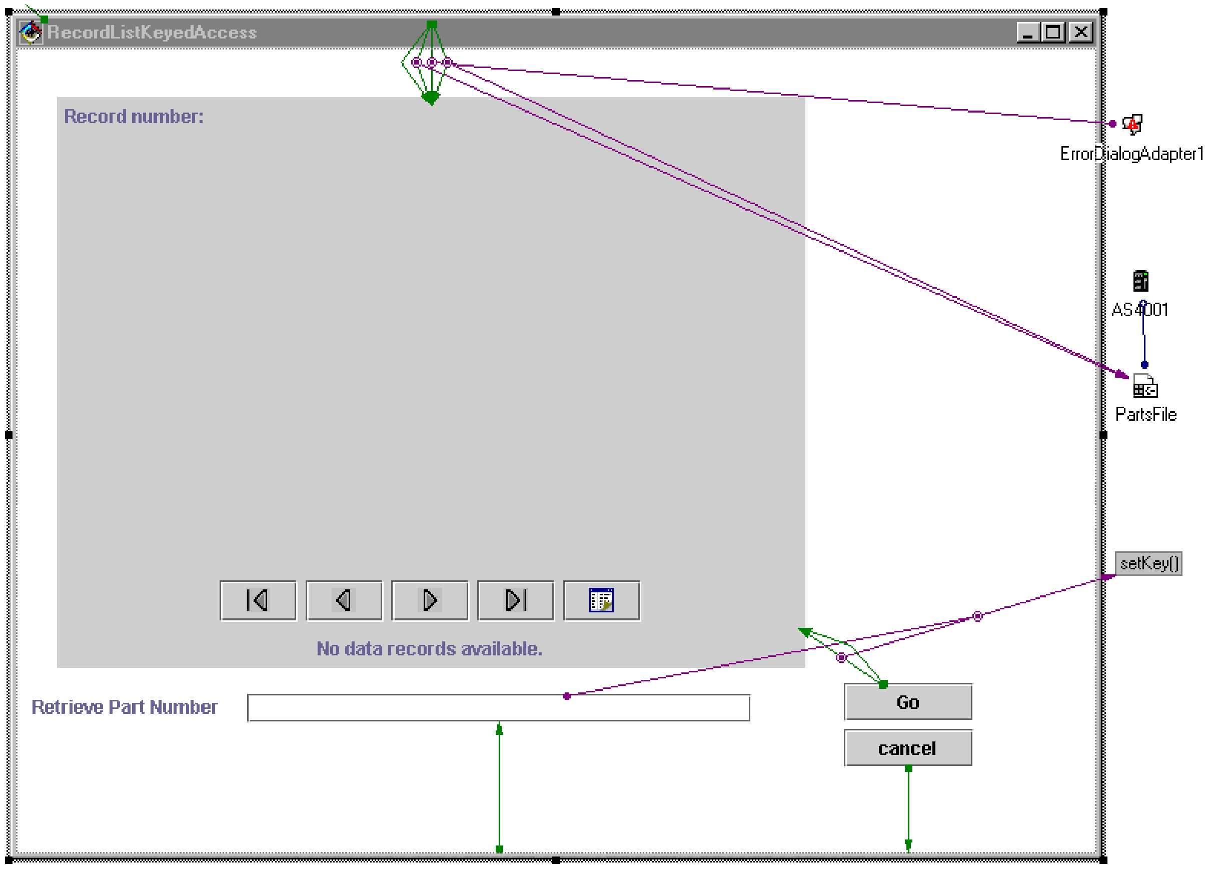
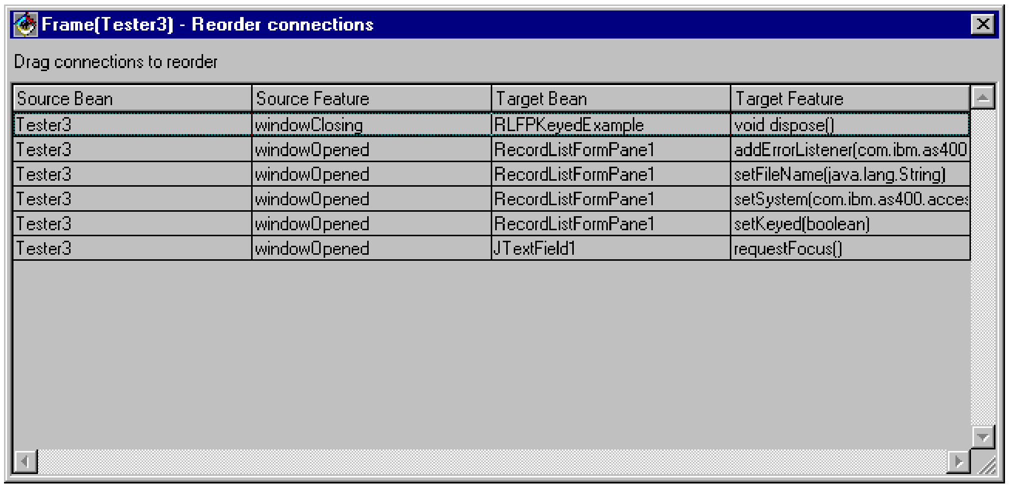


Figure 165. RecordListFormPane in the VCE

By dropping an AS400 object outside the frame in the VCE, we instantiate an AS400 object. We use the AS400 object for the connection to the AS/400 system. We do not set the AS400 properties for system name, user ID, or password. This causes a signon dialog to appear the first time we try to access the AS/400 system. We use the AS400FileRecordDescription object to control the name of the AS/400 file with which we want to work.

We set the path attribute to /QSYS.LIB/APILIB/PARTS.FILE. We use AS/400 IFS (Integrated File System) naming conventions to set the name of the file. This is done exactly the same as in the RecordListTablePane example.

We use the windowOpened event of the frame to control the required initialization processing. Figure 166 on page 211 shows the window events.



Drag connections to reorder

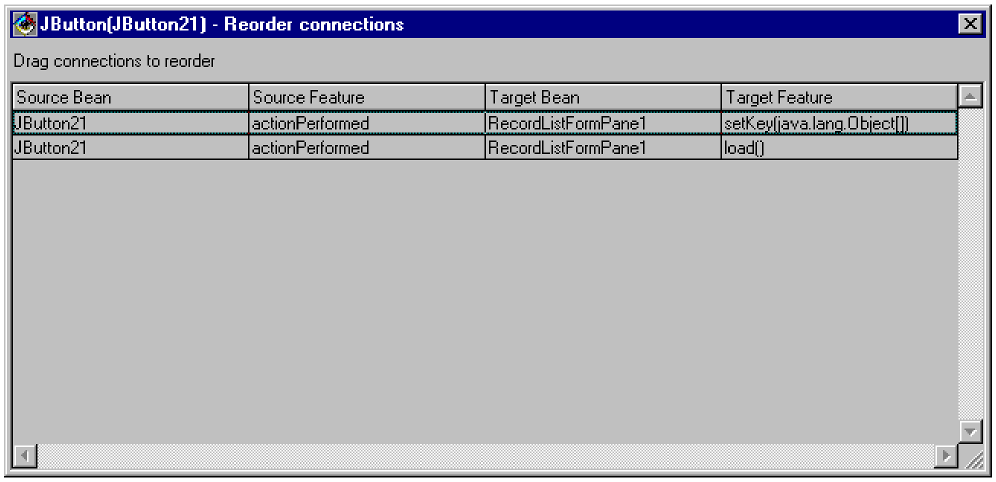
Source Bean	Source Feature	Target Bean	Target Feature
Tester3	windowClosing	RLFPKeyedExample	void dispose()
Tester3	windowOpened	RecordListFormPane1	addErrorListener(com.ibm.as400
Tester3	windowOpened	RecordListFormPane1	setFileName(java.lang.String)
Tester3	windowOpened	RecordListFormPane1	setSystem(com.ibm.as400.acces
Tester3	windowOpened	RecordListFormPane1	setKeyed(boolean)
Tester3	windowOpened	JTextField1	requestFocus()

Figure 166. Window Events Connections

We add an ErrorDialogAdapter to handle error conditions for our application. We use the windowOpened event to add an ErrorListener. We pass in the ErrorDialogAdapter as a parameter to the ErrorListener.

We use the setFileName method of the RecordListTablePane class to set the name of the file with which we want to work. We use the value from the AS400FileRecordDescription object. Next, we set the system name again using the value from the AS400FileRecordDescription object. We use the setKeyed method with a value of true to specify keyed access. Finally, we request focus on the JTextField where we enter the key value of the record that we want to retrieve.

The Go button controls retrieving the record specified in the JTextField from the AS/400 system. The Go button events are shown in Figure 167.



Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
JButton21	actionPerformed	RecordListFormPane1	setKey(java.lang.Object[])
JButton21	actionPerformed	RecordListFormPane1	load()

Figure 167. Go Button Connections

We provide a Java method named setKey to convert the information in the JTextField to a Object[] format, which is required by the RecordListFormPane setKey method.

```
public Object[] setKey(String partNo) {
    Object [] theKey = new Object[1];
    theKey[0] = new java.math.BigDecimal(partNo);
    return theKey;.
}
```

This completes the application. We created an AS/400 client/server application that allows us to use Record Level Access to retrieve records from an AS/400 file using keyed reads. We use a Listener to handle and display any error conditions that occur.

4.4 Conclusion

The AS/400 Toolbox for Java GUI classes allow you to visually represent your AS/400 data and resources. You can quickly build AS/400 client/server applications using these classes. In many cases, you can do this without writing any Java code. You can also combine the GUI classes with your own application-specific code.

Java programs that use the AS/400 Toolbox for Java GUI classes also need Sun's JDK Swing 1.0.1 support. Swing is available with Sun's JFC (Java Foundation Classes) 1.1. See http://www.javasoft.com/products/jfc/index.html for more information about Swing. VisualAge for Java 2.0 provides this support so you can use these classes in the VisualAge for Java Visual Composition Editor.

Chapter 5. AS/400 Toolbox for Java Modification 2

The AS/400 Toolbox for Java Modification 2, as shipped with OS/400 V4R4, adds many new features. Unfortunately, this results in the following requirements for VisualAge for Java 2.0:

- Some new features require JDK 1.1.7 and Swing 1.0.3 support.
- JDBC 2.0 requires Java 2 (JDK 1.2) support.

  The AS400 Toolbox for Java Modification 2 still retains the JDBC 1.22 APIs for use with JDK1.1.7.

To meet the first requirement, some updates must be made to VisualAge for Java 2.0. Some of the updates are temporary fixes that are not currently fully supported. However, JDBC 2.0 is not supported in the current version of VisualAge for Java, since it requires Java 2 (JDK 1.2). The JDBC 2.0 examples discussed in this chapter were created outside of VisualAge for Java using Java 2 (JDK1.2) and the Notepad editor.

For these reasons, we decided to keep the new Modification 2 features in one chapter, rather than integrating them with the Modification 1 information. This way, we hope to allow you to evaluate the new functions and be aware of the requirements they place on your development and deployment systems.

Note: Be aware that you can only use the licensed program for Modification 2 of the AS/400 Toolbox for Java (5769JC1 V4R2M0) with V4R2 or later AS/400 systems. In addition, some functions will only work when communicating with V4R4 systems.

This chapter covers:

- The AS/400 Toolbox for Java GUI Builder
- PDML
- PCML
- JDBC 2.0
- The new access and vaccess classes

You can learn about other new features in Chapter 10, “Deployment Considerations and Tools” on page 365, such as:

- How to secure an application with SSL
- How to reduce the size of deployment archives with the JarMaker tool

5.1 Upgrading the AS/400 Toolbox Contained in VisualAge for Java 2.0

The AS/400 Toolbox for Java version, currently shipped with VisualAge for Java Enterprise Edition version 2.0, is Modification 1. You must update VisualAge for Java to use the AS/400 Toolbox for Java Modification 2 classes.

If you want to use the latest features in the AS/400 Toolbox for Java that are available, you must update VisualAge for Java 2.0 prior to installing the new Toolbox support.

Note: Some of the new AS/400 Toolbox for Java features, such as scrollable cursors, are part of Java 2 (JDK1.2) and will not be available in the VisualAge for

Java IDE. Requirements for the AS/400 Toolbox Modification 2, which ships with OS/400 V4R4 are:

- JDK 1.1.7
- Swing 1.0.3

You must install the VisualAge for Java Rollup2 fix pack to provide the required JDK and Swing support. Be aware that you need approximately 40 MB of free space to install the fix pack. Fix packs, instructions for applying fix packs, and additional information can be found by following the links from http://www.software.ibm.com/ad/vajava to the VisualAge for Java Developer Domain (VADD).

Important

At the time of the writing of this redbook, it was necessary to download two temporary patches for VisualAge, named *1f6xh0w* and *1f78fqu*, from the VisualAge Developer Domain. Without these fixes, we found that the Visual Composition Editor became unstable and would not work properly with AS/400 Toolbox for Java Modification 2 support.

Once you successfully install Rollup2, you can download the AS/400 Toolbox for Java Modification 2 from your AS/400 system or the AS/400 Web site. On the AS/400 system, the Toolbox classes are found in the /QIBM/ProdData/HTTP/Public/jt400/lib/ directory of the integrated file system.

The following instructions enable you to download and install the AS/400 Toolbox for Java modification into the VisualAge IDE:

1. Using Client Access or through the native SMB server, map a network drive to the /QIBM directory of an AS/400 V4R4 system.
2. Start IBM VisualAge for Java.
3. If you are using the Enterprise Edition of VisualAge for Java, ensure that the IBM Enterprise Toolkit for AS/400 V 2.0 feature is loaded into your workspace.
4. Open the project containing the AS/400 Toolbox for Java. If you have the Enterprise Edition, the Toolbox classes are contained in the IBM Enterprise Toolkit for AS/400 V2.0 project.
5. Create an Open Edition of the project.
6. Delete the two packages: **com.ibm.as400.access** and **com.ibm.as400.vaccess**, as shown in Figure 168 on page 215. During the deletion, many error messages will appear. Ignore them for now since they will be fixed when you load the new AS/400 Toolbox for Java.

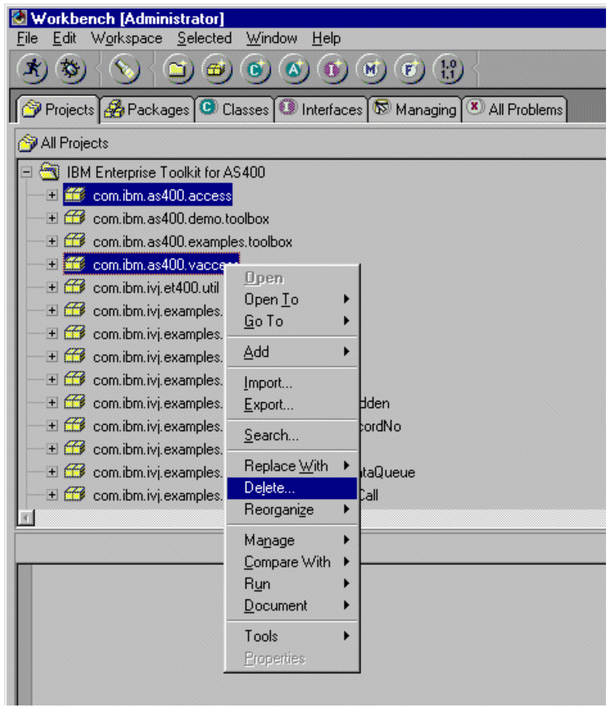
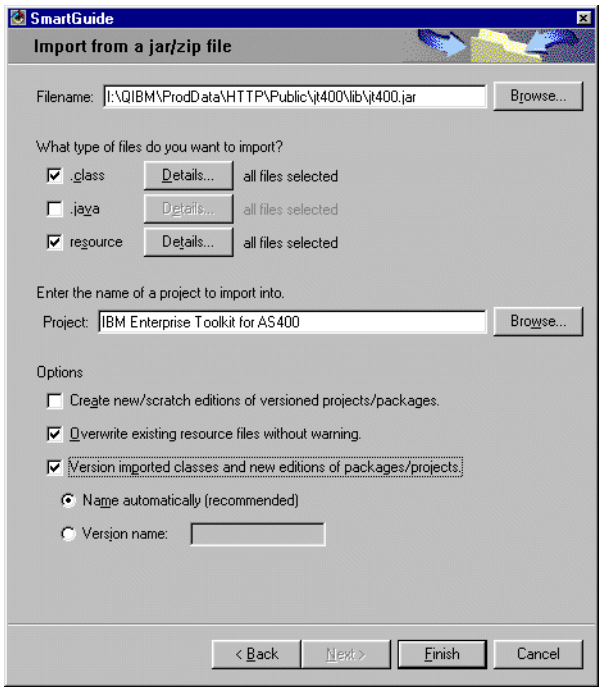


Figure 168. Deleting the Old Packages

7. Reselect the project containing these packages. Select **File—>Import**.
8. Select the **Import from Jar** option, and click **Next.**
9. You are prompted for the file name of the JAR file to import. Locate the **JT400.JAR** file located in the ProdData/HTTP/Public/jt400/lib/ directory of your mapped drive.
10. Check the **Overwrite existing resources without warning** option.
11. Check the **Version imported classes and new editions of packages/projects** option. The complete import smart guide should appear as shown in Figure 169 on page 216.


SmartGuide

Import from a jar/zip file

Filename: I:\QIBM\ProdData\HTTP\Public\jt400\lib\jt400.jar

What type of files do you want to import?

- [✓] .class — all files selected
- [ ] .java — all files selected
- [✓] resource — all files selected

Enter the name of a project to import into.

Project: IBM Enterprise Toolkit for AS400

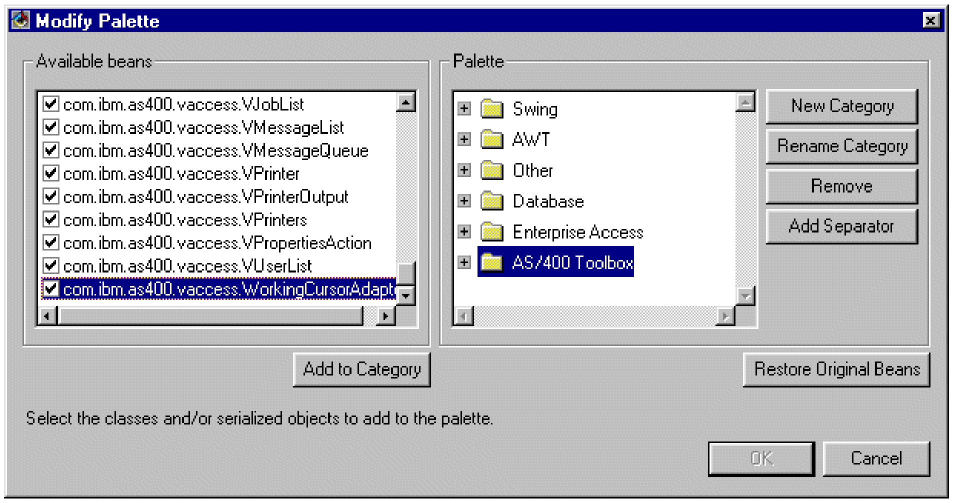
Options

- [ ] Create new/scratch editions of versioned projects/packages.
- [✓] Overwrite existing resource files without warning.
- [✓] Version imported classes and new editions of packages/projects.
  - [●] Name automatically (recommended)
  - [○] Version name:

Figure 169. Importing the New Toolbox

12.Click the **Next** button to start the import process. This process is long running on all but the most powerful machines.

13.After the new AS/400 Toolbox for Java is versioned, you are prompted to add the new beans to a VCE folder. Select all the new beans and add them to the AS/400 Toolbox folder as shown in Figure 170.


Modify Palette

Available beans

- [✓] com.ibm.as400.vaccess.VJobList
- [✓] com.ibm.as400.vaccess.VMessageList
- [✓] com.ibm.as400.vaccess.VMessageQueue
- [✓] com.ibm.as400.vaccess.VPrinter
- [✓] com.ibm.as400.vaccess.VPrinterOutput
- [✓] com.ibm.as400.vaccess.VPrinters
- [✓] com.ibm.as400.vaccess.VPropertiesAction
- [✓] com.ibm.as400.vaccess.VUserList
- [✓] com.ibm.as400.vaccess.WorkingCursorAdapt

Palette

- Swing
- AWT
- Other
- Database
- Enterprise Access
- AS/400 Toolbox

Select the classes and/or serialized objects to add to the palette.

Figure 170. Adding the New AS/400 Beans to the VCE Palette

14.Version the IBM AS400 Toolbox for Java V2.0 project.

You should now be able to use AS/400 Toolbox for Java Modification 2 within IBM VisualAge for Java. You can verify this by using one of the new classes (for example, SecureAS400) within the VCE.

5.2 XML

XML is an abbreviation for eXtensible Markup Language. XML is a tag-based language based on SGML (Standard Generalized Markup Language). SGML is a document language intended for creating large complex documents. XML provides a format for describing data and information. It presents an application with data and describes what the data represents at the same time. Many industries, noticeably publishing and scientific research, are finding that XML provides them a solution to the age-old problem of describing data in an application neutral format. XML looks very similar to HTML in the sense that they are both tag-based languages. However, HTML is more concerned about how to lay out information to present it to a user rather than the meaning of the data. XML does provide the possibility of attaching style information to an XML data file so that it can be rendered in a desired fashion. Its primary concern is data, not display.

The real power of XML is the ability to create tags that have meaning to each particular application. It allows the separation of the data from the presentation of the data, so the data can remain intelligent. The AS/400 Toolbox for Java provides two implementations using XML, which are:

- PDML (Panel Definition Markup Language)
- PCML (Program Call Markup Language)

PDML is a platform-independent way to describe user interfaces in terms of panels and panel components. Many programmers have some initial difficulty in learning Swing or AWT programming. A good proportion of user interfaces can be coded in PDML, which is more simple than AWT or Swing since it is a tagged-based language. PDML also has the ability to use databean classes to assist in the separation of the user interface and data.

PCML is a way to describe program call interfaces. The Java to AS/400 implementation for PCML automatically converts data between AS/400 data types and Java Object classes (such as Integer, String, Float, and so on). In addition, you only need to define the program interface in a single PCML file. This encourages reuse and makes maintaining the interface between legacy programs and Java easier.

5.3 PDML

PDML is based on XML and defines a platform-independent language for describing the layout of user interface elements. Once panels are defined in PDML, you can use the runtime API provided by the Graphical Toolbox to display them. The API displays panels by interpreting the PDML tags and rendering the user interface using Java Foundation Classes.

Although PDML can replace many forms of a user interface, it is not currently a complete replacement for Swing or AWT programming. For example, PDML currently only handles three types of buttons: OK, Cancel, and Help. Other buttons would require you to write event handlers in Java.

PDML is a language developed using XML. It is similar in structure to HTML and SGML.

Here are some examples of PDML tags:

- **<panel>** — Defines a panel
- **<title>** — Specifies the title of the panel or field
- **<size>** — Specifies the size of the panel or field
- **<label>** — Defines a label on the panel (static text field)
- **<location>** — Specifies the location of the field on the panel
- **<button>** — Defines a button on the panel
- **<textfield>** — Defines a textfield on the panel

Some of the tags used in PDML are:

- **<pdml> </pdml>** — Used to identify the start and end of a PDML definition.
- **<panel name="mypanel1">** and **</panel>** — Used to identify the start and end of a panel called mypanel1.
- **<title>xxx</title>** — Sets the title used when displaying a panel. If the panel is the only contents of a window, then this is the window title.
- **<label name="label_1">Name</label>** — Defines a text label (output only) to show on the display.

5.3.1 PDML Example

Figure 171 shows an example of PDML source file.

```
<PDML version="1.0" source="JAVA" basescreensize="1024x768">

   <PANEL name="EXAMPLE_PANEL">
      <TITLE>EXAMPLE_PANEL</TITLE>
      <SIZE>278,120</SIZE>
      <LABEL name="NAME_LABEL" disabled="no">
          <TITLE>NAME_LABEL</TITLE>
          <LOCATION>15,20</LOCATION>
          <SIZE>100,19</SIZE>
      </LABEL>
      <TEXTFIELD name="NAME" masked="no" editable="yes" disabled="no">
          <TITLE>NAME</TITLE>
          <LOCATION>161,14</LOCATION>
          <SIZE>100,26</SIZE>
      </TEXTFIELD>
      <BUTTON name="CLOSE_BTN" disabled="no">
          <TITLE>CLOSE_BTN</TITLE>
          <LOCATION>89,83</LOCATION>
          <SIZE>100,26</SIZE>
      </BUTTON>
   </PANEL>

</PDML>
```

Figure 171. PDML Source Code

Figure 172 on page 219 shows how the panel appears to the end user.

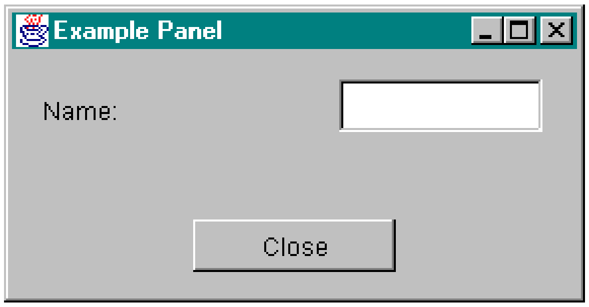


Figure 172. PDML Example Panel

5.4 The Graphical Toolbox

The AS/400 Toolbox for Java Modification 2 includes a user interface framework to provide a productive development environment for building graphical panels. It is called the *Graphical Toolbox*. The framework automatically handles the exchange of data. The developer needs only to create one or more data beans and bind them to the panel components using the Panel Definition Markup Language (PDML).

The Graphical Toolbox consists of two components:

- A graphical user interface builder tool to develop Java GUIs. This tool is a WYSIWYG (What You See is What You Get) GUI editor. It is called the GUI Builder.
- A Resource Script Converter, which converts Windows dialogs to equivalent Java panels.

Underlying these tools is the PDML. The output of both tools are PDML source files. Rather than writing PDML tags yourself, you use these tools to generate PDML source files.

You can use the GUI Builder to quickly and easily create new panels from scratch. Or, you can use the Resource Script Converter to convert existing Windows-based panels to Java. Both tools support internationalization for different locales.

The GUI Builder helps you create custom user interface panels in Java. You can incorporate the panels into your Java applications, applets, or Operations Navigator plug-ins. The panels may contain data obtained from the AS/400 system, or data obtained from another source such as a file in the local file system or a program on the network.

The GUI Builder is a WYSIWYG visual editor for creating Java dialogs, property sheets, and wizards. With the GUI Builder you can add, arrange, or edit user interface controls on a panel, and then preview the panel to verify that the layout behaves the way you expected. You can use the panel definition in a dialog, insert panels into property sheets and wizards, or arrange the panels in splitter panes, deck panes, and tabbed panes.

The Resource Script Converter converts Windows user interface elements into a form usable by Java programs. With the Resource Script Converter, you can process Windows resource scripts (RC files) from your existing Windows applications and produce definitions of dialogs, property sheets, and wizards that can be displayed in Java.

5.4.1 Installing the Graphical Toolbox on Your Workstation

To develop Java programs using the Graphical Toolbox, you should install the Graphical Toolbox jar files on your workstation. There are two ways to do this.

If you already installed the AS/400 Toolbox for Java licensed program on an AS/400 system, you can copy the jar files from the directory: /QIBM/ProdData/HTTP/Public/jt400/lib. You can use FTP to do this (ensure that you transfer the files in binary mode), or map a network drive and copy them.

You can also install the Graphical Toolbox when you install Client Access Express V4R4M0. The AS/400 Toolbox for Java is shipped as part of Client Access Express V4R4M0. If you are installing Client Access Express for the first time, choose **Custom Install** and select the AS/400 Toolbox for Java component on the installation menu. If you have already installed Client Access Express, you can use the Selective Setup program to install this component if it is not already present.

Before you start using the AS/400 Toolbox for Java GUI Builder, it is essential that you make sure the classpath settings are correct. The simplest way to achieve this is to set up additional environment variables and then append them to the existing classpath. We show you two batch files to achieve this. The file addtoolbox.bat is used to add just the com.ibm.as400.access and com.ibm.as400.vaccess packages to the system CLASSPATH. The file adduitools.bat adds the GUI Builder tools and runtime classes.

The Graphical Toolbox is delivered as a set of jar files, which include:

- **uitools.jar** — Contains the GUI Builder and Resource Script Converter tools.
- **jui400.jar** — Contains the runtime API for the Graphical Toolbox. Java programs use this API to display the panels constructed using the tools. These classes may be redistributed with applications.
- **data400.jar** — Contains the runtime API for the PCML. Java programs use this API to call AS/400 programs whose parameters and return values are identified using PCML. These classes may be redistributed with applications.
- **util400.jar** — Contains utility classes for formatting AS/400 data and handling AS/400 messages. These classes may be redistributed with applications.
- **x4j400.jar** — Contains the XML parser used by the API classes to interpret PDML and PCML documents.

To use the Graphical Toolbox, you must add these jar files to your CLASSPATH environment variable (or specify them on the classpath option on the command line). For example, if you copied the files to the directory C:\jt400\lib on your workstation, you must add the following path names to your classpath:

```
C:\jt400\lib\uitools.jar;
C:\jt400\lib\jui400.jar;
C:\jt400\lib\data400.jar;
C:\jt400\lib\util400.jar;
C:\jt400\lib\x4j400.jar;
```

If you installed the Graphical Toolbox using Client Access Express, the jar files will all reside in the directory \Program Files\Ibm\Client Access\jt400\lib on the drive where you installed Client Access Express. The path names in your classpath should reflect this.

Note

Internationalized versions of the GUI Builder and Resource Script Converter tools are available. To run a non-U.S. English version, you must add the correct version of the uitools.jar for your language and country to your Graphical Toolbox installation. These jar files are available on the AS/400 system in the /QIBM/ProdData/HTTP/Public/jt400/Mri29xx directory, where 29xx is the four-digit OS/400 NLV code corresponding to your language and country. The non-English jar files are automatically installed by Client Access Express if the Client Access primary language is not English.

5.5 Java Plug-in for Operations Navigator

This section explains how to build an extension to Operations Navigator V4R4M0 using the AS/400 Toolbox for Java GUI Builder tool. If you have Operations Navigator installed on your workstation, you can follow these steps to build your own extension. Figure 173 shows the System Status menu item that we add to the System context menu.

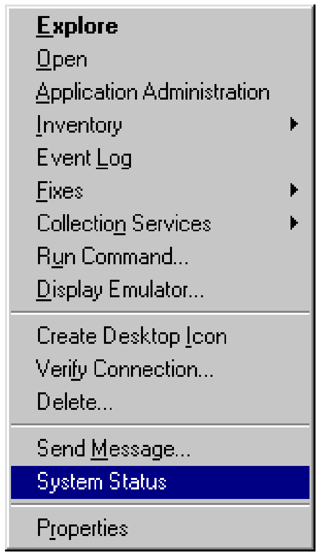
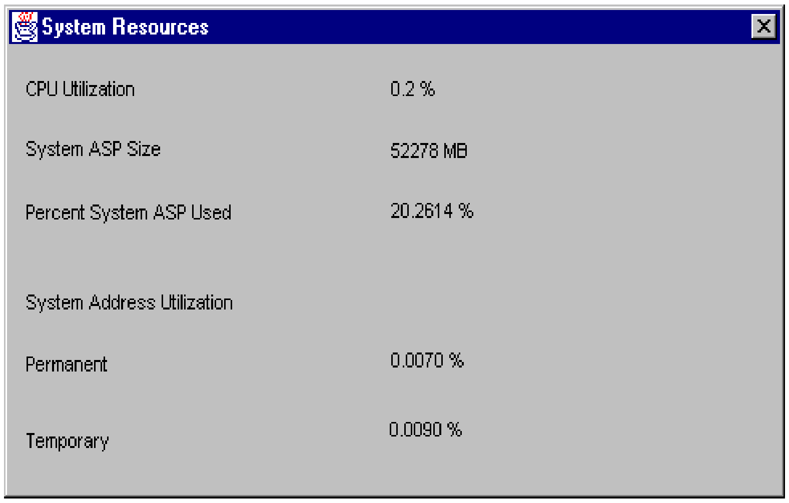


Figure 173. The Operations Navigator Extension

Figure 174 on page 222 shows the panel displayed when the System Status menu option is selected.



CPU Utilization 0.2 %

System ASP Size 52278 MB

Percent System ASP Used 20.2614 %

System Address Utilization

Permanent 0.0070 %

Temporary 0.0090 %

Figure 174. The System Resources Panel

To build an extension, we follow these steps:

1. Start the GUI Builder.
2. Create a new panel definition.
3. Modify the generated databean to retrieve data from the AS/400 system.
4. Test the application.
5. Write an ActionsManager implementor to extend Operations Navigator.
6. Modify the Windows registry to reflect the extension.
7. Test the Operations Navigator extension.

5.5.1 Setting Up the GUI Builder

Before you start using the AS/400 Toolbox for Java, you need to make sure that your CLASSPATH settings are correct. The easiest way to make sure that your setup is correct, is to create additional environment variables. Then, append them to your existing CLASSPATH. For these examples, we installed the AS/400 Toolbox for Java Modification 2 in a directory named AS400ToolBoxMod2 and the Java Swing support in a directory named Swing.0.3. We used JDK 1.1.7B.

You can set your CLASSPATH by performing either of these tasks:

- Add the **com.ibm.as400.access**, **com.ibm.as400.vaccess**, and other user-interface (UI) related packages directly to your CLASSPATH.
- Create small batch files that add these packages to your CLASSPATH.

If you want to add just the com.ibm.as400.access and com.ibm.as400.vaccess classes to your system CLASSPATH, create a batch file named addtoolbox.bat. Your batch file should look like the one shown in Figure 175.

```
set SWING_HOME=C:\Swing1.0.3
set TOOLBOX_HOME=C:\AS400ToolBoxMod2\

set CLASSPATH=%CLASSPATH%;%SWING_HOME%\swingall.jar;
                   %TOOLBOX_HOME%\lib\jt400.jar
```

Figure 175. The addtoolbox.bat File

If you want to add the GUI Builder tools and runtime environment to your system, create a batch file named adduitools.bat. Your batch file should look like the one shown in Figure 176.

```
set GUITOOLS=%TOOLBOX_HOME%\lib\data400.jar;%TOOLBOX_HOME%\lib\ui400.jar
set GUITOOLS=%GUITOOLS%;%TOOLBOX_HOME%\lib\uitools.jar
set GUITOOLS=%GUITOOLS%;%TOOLBOX_HOME%\lib\util400.jar
set GUITOOLS=%GUITOOLS%;%TOOLBOX_HOME%\lib\x4u400.jar

set GUITOOLS=%CLASSPATH%;%GUITOOLS%
```

Figure 176. The adduitools.bat File

Once you run these two batch files in a command prompt window, you are ready to use the AS/400 Toolbox for Java classes.

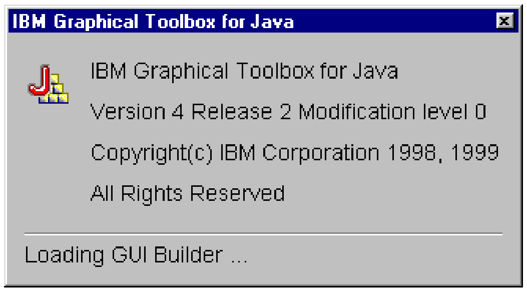
5.5.2 Starting the GUIBuilder

Before we start the GUI Builder, we create a directory named C:\L04\Student to hold all the generated files. Then, we change to that directory.

To start the GUIBuilder, from the command line, enter:

```
java com.ibm.as400.ui.tools.GUIBuilder
```

Providing the classpath was set up properly, you are presented with a loading GUI Builder flash, as shown in Figure 177.


IBM Graphical Toolbox for Java

IBM Graphical Toolbox for Java
Version 4 Release 2 Modification level 0
Copyright(c) IBM Corporation 1998, 1999
All Rights Reserved

Loading GUI Builder ...

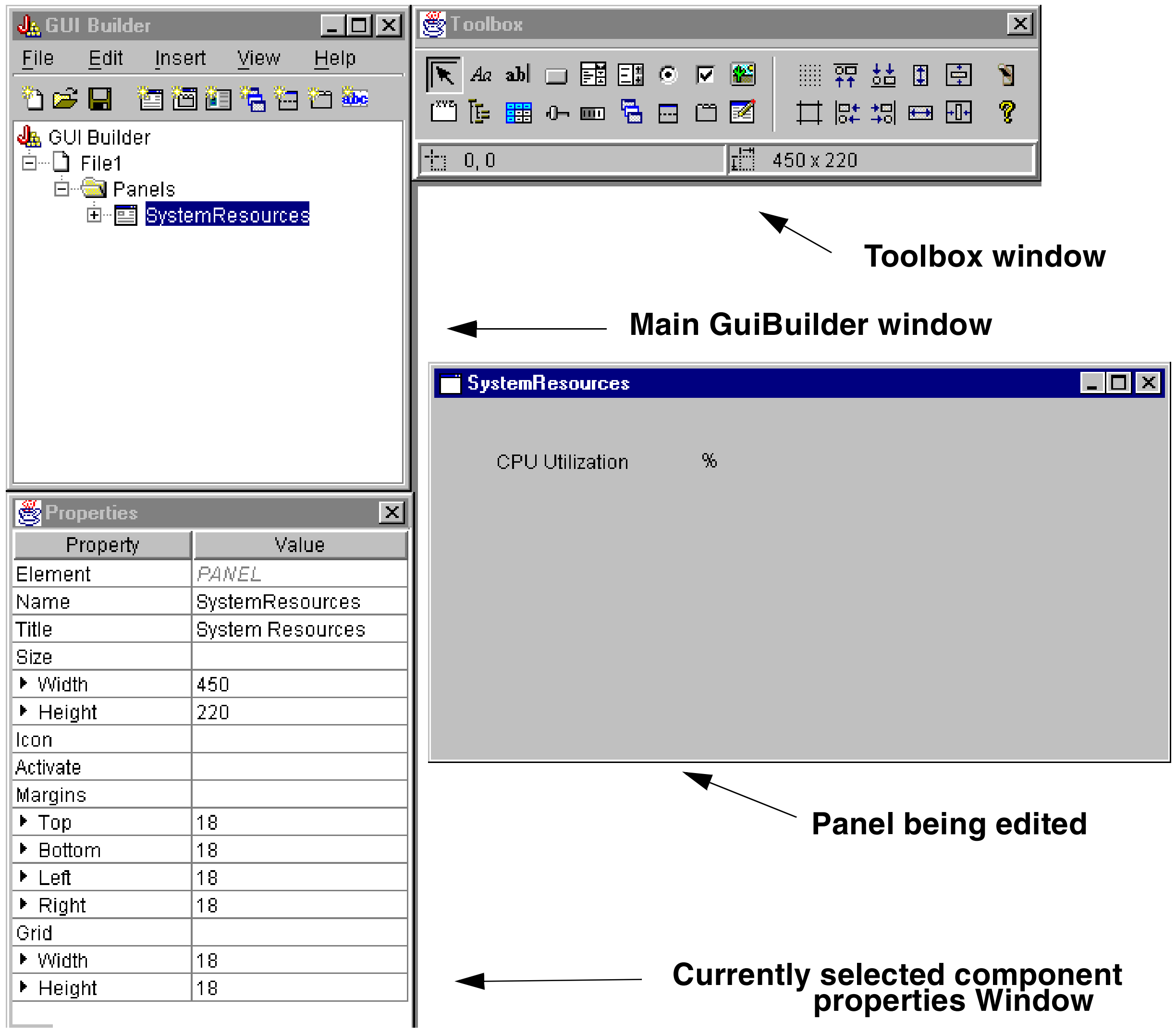
Figure 177. Loading the GUIBuilder Tool

Once the IBM Graphical Toolbox for Java is loaded, a window appears, similar to that shown in Figure 178 on page 224. The GUIBuilder main window shows the loaded PDML files in an hierarchical layout. The various tree nodes can be expanded or collapsed to enable you to select and work on specific PDML components.

The Properties window shows the properties of the currently selected component. In the illustration, the SystemResources panel is the selected panel. Modifying any properties in this window can affect the SystemResources panel.

The Toolbox window is key for GUI development. Various tool items, such as labels, text fields, buttons, lists, radio buttons, check boxes and many other GUI components can be selected from here and added onto a panel or other container GUI component. The GUI Toolbox window also allows you to select various tools to assist in the layout of the GUI container. All of the tool items have fly-over text.

If you are searching for a tool, simply move the mouse pointer along the items until you find the required tool.



Property	Value
Element	PANEL
Name	SystemResources
Title	System Resources
Size	
Width	450
Height	220
Icon	
Activate	
Margins	
Top	18
Bottom	18
Left	18
Right	18
Grid	
Width	18
Height	18

Figure 178. The GUI Builder Windows

5.5.3 Creating the New Panel Definition

In this section, we create a new panel definition, add some basic components, and generate skeleton Java code. In subsequent sections, we modify the generated Java code to retrieve data from the AS/400 system. This code is called a *databean*.

During this task, we perform the following steps:

1. Create a new PDML file.
2. Create a panel and populate it with the required user interface.
3. Bind some of the user interface objects to a databean.
4. Save the new PDML files and generate the skeleton Java code for the databeans.

Here are the steps that are necessary to complete this task:

1. Run the GUIBuilder.

   Enter `java com.ibm.as400.ui.tools.GUIBuilder` to start the builder.

2. Create a new PDML file

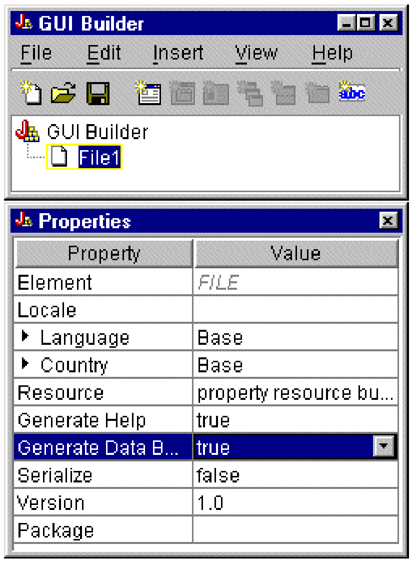
   a. Create a new file by selecting **File—>New** or by clicking on the **new file** icon.

   b. Select the Properties window.

   c. Change the Generate DataBean property to "true" by highlighting the property value field.

   d. Change the Generate Help to "true".

   The main and property panels appear as shown in Figure 179.



Property	Value
Element	FILE
Locale	
Language	Base
Country	Base
Resource	property resource bu...
Generate Help	true
Generate Data B...	true
Serialize	false
Version	1.0
Package	

Figure 179. Modifying the PDML File Properties

3. Create a new Panel.

   a. Select **Insert—>Panel** from the main window menu, as illustrated in Figure 180.

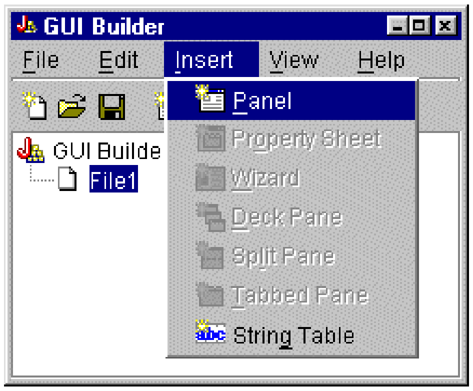
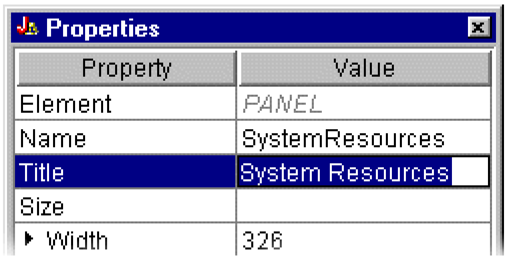


Figure 180. Inserting a New Panel

This causes the appearance of a new window in the GUIBuilder. The window is the newly created panel. The title of the window will be the name of the panel. At this point, it should appear as Panel_1.

b. Once added, highlight the panel in the main window and then modify the panel properties. The name of the panel should be SystemResources. The title should be set to System Resources as illustrated in Figure 181.



Property	Value
Element	PANEL
Name	SystemResources
Title	System Resources
Size	
Width	326

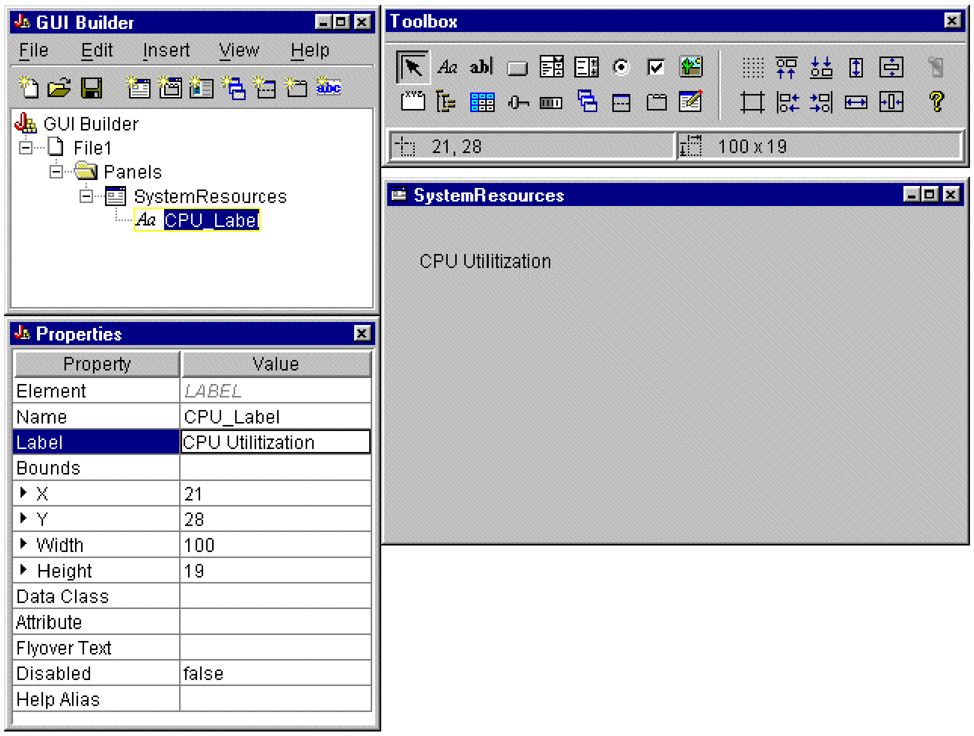
Figure 181. Setting the Properties of System Resources Panel

4. Add the required basic text labels.

   Select the label icon from the Toolbox window and click on the new panel. Then modify the label to set these properties:

   • **Name**: CPU_Label
   • **Label**: CPU Utilization

   Figure 182 shows all the various GUIBuilder windows after performing this task.



Property	Value
Element	LABEL
Name	CPU_Label
Label	CPU Utilitization
Bounds	
X	21
Y	28
Width	100
Height	19
Data Class	
Attribute	
Flyover Text	
Disabled	false
Help Alias	

Figure 182. The GUIBuilder after Adding and Modifying the First Label

5. Repeat step 4 to add the five labels listed in Table 28 and modify their properties.

Table 28. Labels to Add to the Panel

Name	Label
ASPSize_Label	System ASP Size
ASPUsed_Label	Percent System ASP Used
Addresses_Label	System Address Utilization
Perm_Label	Permanent
Temp_Label	Temporary

6. Align the labels on the left-hand side of the panel using the alignment tool. The panel should appear as shown in Figure 183. To align the labels, press the **Control** key and click on the labels with the mouse pointer. Then, release the **Control** key and click the **Alignment** button.

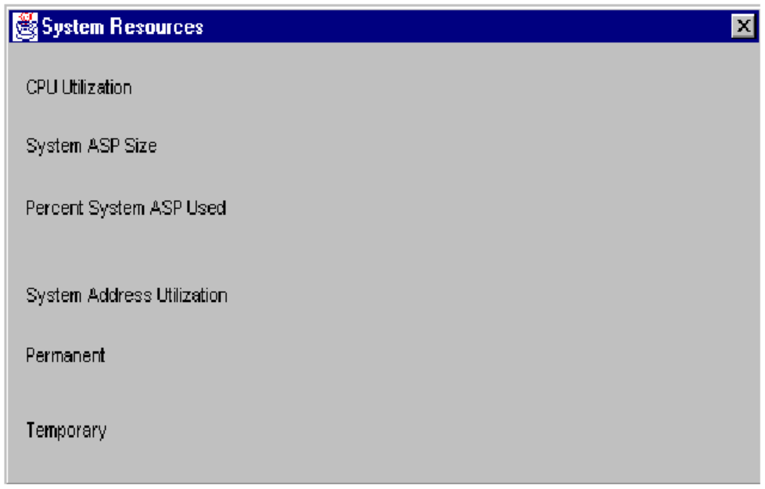


Figure 183. System Status Panel after Inserting the Labels

7. Add labels to display information from a databean.

   Since we are only displaying information we only need to use labels, instead of text fields, to present data to the end user. However, if you were to request data from the user, then you would use text fields.

   To create bound labels, simply set the DataClass and Attribute property of a label. Then, providing that the PDML file Generate DataBeans property is true, the GUI Builder builds the skeleton get and set methods for the databean. The one drawback to using a label, as opposed to a text field, is that it has no concept of any data types, except Strings. This means that your get and set methods must accept String input values and return Strings. The set methods are only called when the panel in instantiated through a programmed method invocation.

Table 29 shows the labels and associated properties that need to be added to complete this task.

Table 29. Labels Required to Show the System Status Values

Label Name	Label Value	Data Class	Attribute
CPU_Value	%	SystemStatusEngine	CpuUtilization
ASPSize_Value	MB	SystemStatusEngine	SystemASPSize
ASPUsed_Value	%	SystemStatusEngine	SystemASPPercent
Perm_Value	%	SystemStatusEngine	PermanentAddressesUsed
Temp_Value	%	SystemStatusEngine	TemporaryAddressesUsed

After we successfully complete the addition of these new labels, we align them in as shown in Figure 184.

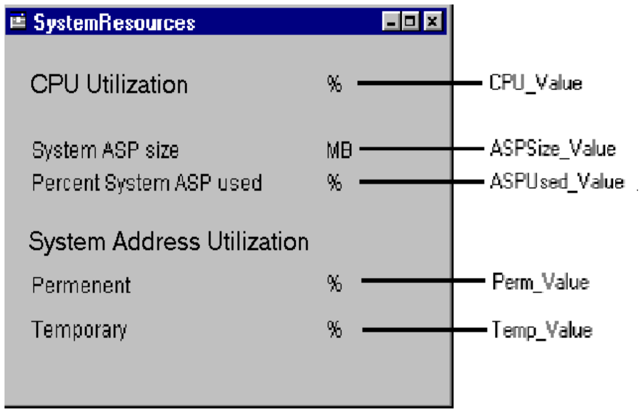


Figure 184. Completed SystemResources Panel

8. Save the PDML file.

   a. To save a PDML file, select **File—>Save** from the menu, or click on the save icon in the main window toolbar. When prompted for a filename and location, save the PDML file as SystemStatus**.**

   b. Exit the GUI Builder application.

   Table 30 shows the generated files.

Table 30. The Contents of the Directory

File name	Content
SystemStatus.pdml	The PDML definitions including the main panel, layout, and types of labels and their bindings to the databean.
SystemResources.html	A skeleton help file in HTML format. This can be edited to display context-sensitive help for the benefit of the end user.
SystemStatus.properties	The initial values for each of the labels. Other information used by the GUIBuilder is also stored in this file.
SystemStatusEngine.java	The skeleton Java code generated for the bound components
SystemStatusEngine.class	A compiled version of the SystemStatusEngine.java file.

Serialization

To increase run-time performance, a PDML file can be serialized. The panel is then constructed using the serialized file. In this case, the IBM XML parser is not used at runtime. To serialize a PDML file, select **File—>Properties** on the GUI Builder main window. Then, turn on the **Serialize** option.

5.5.4 Modifying the Databean to Retrieve Data from the AS/400 System

During this task, we modify the generated SystemStatusEngine Java code to retrieve data from the AS/400 system, so it can be displayed on the panel.

The databean needs to know about the system from which we retrieve data. To do this, we create a constructor method that accepts an AS400 object. The only other method that needs to be modified is the load() method, which retrieves the AS/400 system status data and sets the databean attributes.

We follow these steps to modify the databean:

1. Open the SystemStatusEngine.java file using the notepad editor.

   Once open, we find that the following methods were created:

   - Method
   - getSystemASPPercent()
   - getCpuUtilization()
   - getSystemASPSize()
   - getPermanentAddressesUsed()
   - getTemporaryAddressesUsed()
   - getCapabilities()
   - verifyChanges()
   - save()
   - load()

   The get methods, with the exception of getCapabilities(), are generated as a result of binding labels to databean classes. To make the data class useful, we write a SystemStatusEngine(AS400) constructor method and modify the load() method to retrieve the data from the AS/400 system.

2. Add two new import statements.

   Since we will use some of the AS/400 Toolbox for Java access classes, we import the com.ibm.as400.access classes. In addition, the com.ibm.as400.opnav.Monitor class allows us to log error messages. This is a useful way to trap errors once the extension is loaded.

   We move the cursor to the line after the first import statement and add the following import statements:

   ```
   import com.ibm.as400.access.*;
   import com.ibm.as400.opnav.Monitor;
   ```

3. Add an instance variable to the class.

   We add a private instance variable called m_sAS400 of type AS400 to the end of the list of variables:

   ```
   private AS400 m_sAS400;
   ```

4. Create a constructor method.

   We add a construct method that will accept an AS400 object as a parameter and store it in the instance variable called m_sAS400.

   ```
   public SystemStatusEngine(AS400 anAS400) {
   m_sAS400=anAS400;
   }
   ```

5. Modify the load() method.

   Internationalization

   Percentages are handled differently for different languages and countries. The technique used here may have to be changed for a different language.

   a. Locate the load() method.

   b. Enclose the existing assignments within a try block. If we catch an error, we use the logThrowable method to log it.

   ```
   public void load() {
    try {
     m_sSystemASPPercent = "";
     m_sCpuUtilization = "";
     m_sSystemASPSize = "";
     m_sPermanentAddressesUsed = "";
     m_sTemporaryAddressesUsed = "";
    } catch (Exception e) {
     Monitor.logThrowable(e);
    }
   }
   ```

   c. Before the assignments are made, create a new object called aStatus of type SystemStatus. Then, we use the appropriate constructor method to create the instance and set the system to m_sAS400.

   d. Use the AS/400 Toolbox for Java SystemStatus class to retrieve the values to display from the AS/400 system.

   e. Modify the existing assignments to extract the appropriate attributes from the newly created aStatus object, for example:

   ```
   m_sSystemASPPercent = "";
   ```

   would become

   ```
   m_sSystemASPPercent =
   newFloat(aStatus.getPercentSystemASPUsed()).toString() + " %";
   ```

   Completed Code

   See Section C.1, “SystemStatusEngine.java” on page 407, for the complete source code.

6. Save the changes and close Notepad.

7. Modify the Classpath and compile the new Java code.

   In addition to the Toolbox classes, use the class com.ibm.as400.opnav.Monitor, which is found in an archive file named jopnav.jar in the Classes subdirectory of Client Access Express. Add this archive to the classpath before compiling the Java source.

a. To add the archive file, enter:

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\Ibm\Client
Access\Classes\jopnav.jar
```

b. To compile the Java source, enter:

```
javac SystemStatusEngine.java
```

5.5.5 Testing the Application

To test the application, we write a test program. It is very small. The code is shown in Figure 185.

```
import com.ibm.as400.access.*;
import com.ibm.as400.ui.framework.java.*;
class SystemStatusTester extends java.awt.Frame {
public static void main(String[] args) {
  SystemStatusTester aSystemStatusTester = new SystemStatusTester();
  AS400 theMachine = new AS400();
  SystemStatusEngine theSystemEngine = new SystemStatusEngine(theMachine);
  theSystemEngine.load();
  DataBean[] dbeans = {theSystemEngine};
  PanelManager pm = null;
  try {
      pm = new PanelManager("SystemStatus", "SystemResources", dbeans,
           aSystemStatusTester);
  } catch (DisplayManagerException e) {
      e.displayUserMessage(aSystemStatusTester);
  }
  pm.setVisible(true);
       }
}
```

Figure 185. SystemStatusTester.java

The code follows this sequence:

1. Instantiates an AS400 object named theMachine.
2. Instantiates a new SystemStatusEngine object named theSystemEngine passing in the AS400 object as a parameter.
3. Executes the load() method of the theSystemEngine object to cause it to retrieve the system information.
4. Creates a DataBean array, which contains the theStatusEngine object.
5. Creates a PanelManager object named pm.
6. Instantiates the **pm** object using the constructor method, passing the following parameters in sequence:
   a. A String containing the name of the PDML file. The PDML document must be in a directory or JAR file in the CLASSPATH. The PanelManager first looks for a serialized panel definition before attempting to parse the PDML file.
   b. A String containing the name of the actual panel required.

   c. The array of databeans to be used by this panel.
   d. The owning frame.

7. Executes the setVisible method to show the panel.

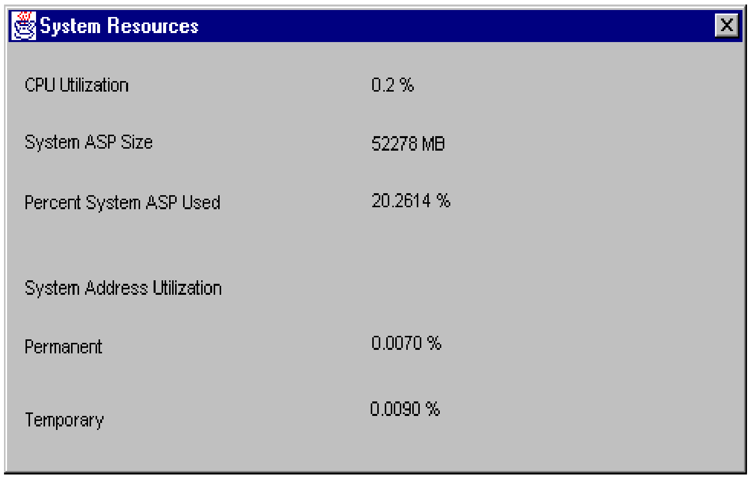
We compile the code using:

```
javac SystemStatusTester.java
```

We run the program using:

```
java SystemStatusTester
```

If everything works correctly, we are prompted to sign on to the AS/400 system. After signing on to the AS/400 system, we see the SystemStatus panel in which the system information is displayed as shown in Figure 186.



System Resources

CPU Utilization	0.2 %
System ASP Size	52278 MB
Percent System ASP Used	20.2614 %
System Address Utilization	
Permanent	0.0070 %
Temporary	0.0090 %

Figure 186. System Information

5.5.6 Adding an Operations Navigator Plug-in

In this section, we deploy the PDML application as a plug-in to Operations Navigator. The ActionsManager interface displays context menus to a user. It is the simplest interface to write to and use to extend Operations Navigator.

Once an actions manager is added into the Operations Navigator, various methods are called depending on the action of the user and the state of Operations Navigator.

When implementing the ActionsManager interface, we implement these methods:

- initialize(ObjectName[] arg1, ObjectName arg2)
- queryActions(int arg1)
- actionSelected(int arg1, java.awt.Frame arg2)

These methods are covered in detail in the following steps. The completed Java code can be found in Section C.2, “SystemStatusManager” on page 408.

1. We create a new Java source file named SystemStatusManager.java.
2. We add import statements.

We import all the classes in the following packages:

```
com.ibm.as400.opnav
com.ibm.as400.ui.framework.java
com.ibm.as400.access
```

3. We define a new Class.

   We create a public class definition named SystemStatusManager that extends the Object class and implements ActionsManager.

   We define two private class variables:

   - **initObjs** — An array of ObjectName
   - **dragDropObj** — Of type ObjectName

4. We define an initialize method.

   When Operations Navigator detects that a plug-in or extension may be required to perform an action, it calls the initialize method. The initialize method accepts two variables as parameters and returns void. The first parameter is an array of ObjectName that identifies objects on which the ActionsManager implementor may want to perform some actions. The second parameter is a single ObjectName that is used if an Operations Navigator object is dragged or dropped onto another Operations Navigator object.

   In this case, we register the SystemStatusManager as a context manager of the AS/400 Network. This way, it is called when the end user right clicks on an AS/400 system. As such, the only type of Operations Navigator ObjectName we should receive is an "AS4" type.

   The parameters that are passed should be stored in the class variables. A completed definition of this method appears is shown here:

```
public void initialize(com.ibm.as400.opnav.ObjectName[] arg1,
   com.ibm.as400.opnav.ObjectName arg2) {
   initObjs = arg1;
   dragDropObj = arg2;
}
```

5. We define a queryAction method.

   The queryAction method is used by Operations Navigator to ask the ActionsManager to report on the actions it can perform and what is displayed to the user. To inform Operations Navigator, the queryActions method returns an array of ActionDescriptor objects. An ActionDescriptor object has a number of attributes, some of which are shown in Table 31 on page 234.

Table 31. Attributes and Methods from the ActionDescription Class

Attribute	Methods	Comment
HelpText	set, get	Action help text displayed in the status field in the Operations Navigator main window.
Text	set, get	Text displayed in the menu.
Verb	set, get	A non-displayed string used to identify the action.
ID	get, set & ActionDescription(ID)	A numeric integer used to identify the action. All actions should be assigned a unique ID for any ActionManager implementor.
subActions	set, get	Enables context menus within menu items.
Default	set, isDefault()	If true, then this menu item is the default action on this context menu.

In this case, the queryActions method needs to return an array of one element, since we are only performing one action. The ActionDescription returned needs to have the attributes set that are shown in Table 32.

Table 32. ActionDescription Attributes

Property	Required Value
ID	1
Text	System Status
HelpText	Loads the ITSO System Status Plugin
Verb	ITSOSysSts

Before setting up the ActionDescription, ensure that you are defining the correct menu to display in the given context. This is achieved by checking the ObjectType that was set when the initialize method is called. In this case, we registered the context menu to exist only under an AS/400 System. This object type is AS4, and it represents an AS/400 network connection. For a complete list of the available types, see the system registry entries for HKEY_CLASSES_ROOT\IBM.AS400.Network\TYPES.

The getObjectType method can throw an ObjectNameException so we encapsulate the code within a **try** block. The following code sample illustrates all of these points:

```
public ActionDescriptor[] queryActions(int arg1) {

   ActionDescriptor[] actions = new ActionDescriptor[0];
   String objType = null;

   try {
      objType = initObjs[0].getObjectType();
      if (objType.equals("AS4")) {
      if ((arg1 & CUSTOM_ACTIONS) == CUSTOM_ACTIONS) {
       actions = new ActionDescriptor[1];
       ActionDescriptor act = new ActionDescriptor(1);
       act.setText("System Status");
```

```
        act.setHelpText("Loads the ITSO System Status Plugin");
        act.setVerb("ITSOSysSts");
        actions[0] = act;
       }
     }
   }
   catch (Exception e) {
      Monitor.logThrowable(e);
   }
   return actions;
 }
```

6. We define an actionSelected method.

   The actionSelected method is only called by Operations Navigator when the user selects one of the actions returned in queryActions. For this reason, we need to instantiate any databeans and code the majority of the plug-in within this method:

   a. We add a public method called actionSelected that accepts two arguments. The first argument should be an integer called arg1. The second should be a Frame called arg2. The method does not return a value.

   b. We use an **if** clause to verify that the action selected (arg1) is equal to 1. Since this is the only value set by queryActions, it should always be set to 1.

   c. Once the action is verified, we use a **try** block since the subsequent methods may throw an exception.

   d. Within the **try** block we extract the AS/400 system object from the first element in the initObjs variable using the getSystemObject(). We store this value in a new AS400 object called theMachine.

   e. We use theMachine to create a new instance of the SystemStatusEngine databean and call it theStatusEngine.

   f. We have the theStatusEngine load data by calling the load() method.

   g. We create an array called dBeans of type DataBean. We set the first and only element to be theSystemStatusEngine.

      Now that the databean is initialized, we need to display the SystemResource panel created in Section 5.5.3, “Creating the New Panel Definition” on page 224. To achieve this, we use the PanelManager class and construct a panel manager object using the databean and the owning frame.

   h. We declare a PanelManager object called pm and set it to null.

   i. Within a **try** block, we instantiate the pm object using the constructor method, passing the following parameters in sequence:

      i. A String containing the name of the PDML file. The PDML document must be in a directory or JAR file in the CLASSPATH. The PanelManager will first look for a serialized panel definition before attempting to parse the PDML file.

      ii. A String containing the name of the actual panel required.

iii. The array of databeans to be used by this panel.

iv. The owning frame.

j. If the PanelManager constructor method throws a PanelManagerException**,** we catch it and use the displayUserMessage() method, to display a message to the user.

k. To display the new panel and hand over control to this panel to the user, we use the setVisible(true) method.

l. We use a catch to trap all exceptions and log them to the Monitor log using the logThrowable() method.

After completing this step the defined method should appear like this example:

```
public void actionSelected(int arg1, java.awt.Frame arg2) {

   if (arg1 == 1)
   {
      try {
          AS400 theMachine = (AS400)initObjs[0].getSystemObject();
          SystemStatusEngine theSystemEngine = new
          SystemStatusEngine(theMachine);
          theSystemEngine.load();
          DataBean[] dbeans = {theSystemEngine};

          PanelManager pm = null;

          try
          {
          pm = new
          PanelManager("SystemStatus","SystemResources",dbeans,arg2);
          }
          catch (DisplayManagerException e){
          e.displayUserMessage(arg2);
          }
          pm.setVisible(true);
      }
      catch (Exception e) {
         Monitor.logThrowable(e);
      }
   }
}
```

7. We save the Java source.
8. We Compile the Java Source.

Source Code

See Section C.2, “SystemStatusManager” on page 408, for the full source code listing.

5.5.7 Modifying the Windows Registry

To reduce the possibility of incorrectly setting the registry, a file called SystemStatusPlugin.reg is used. This file was produced using REGEDIT4. An incorrectly set registry can, under extreme circumstances, cause your machine to stop functioning. Therefore, we recommend that you always backup the registry prior to any modification. Follow these steps:

1. Backup the registry.
   a. Start the regedit program
   b. Select **Registry—>Export Registry File** from the menu and export all entries to a file called Original in the C:\L04\Student directory.
2. From Windows NT Explorer, double click on the **SystemStatusPlugin.reg** file to have the system apply it to the registry.

Results

See Section C.3, "SystemStatus Registry" on page 408, for a listing of the registry file.

5.5.8 Testing the Extension

To test the extension, we perform the following steps:

1. Start Operations Navigator. If it is already active, then we close and re-open it to force it to read the registry again.
2. Select and expand the defined AS/400 system.
3. Log on to the system as prompted.
4. If the extension is registered, a scan operation takes place. Click the **Scan Now** button to confirm that the scan can occur now.
5. Right click on the system menu item to see the newly installed extension is available. Figure 187 shows the context menu with the new menu item.

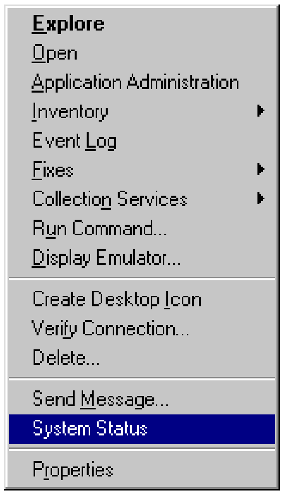
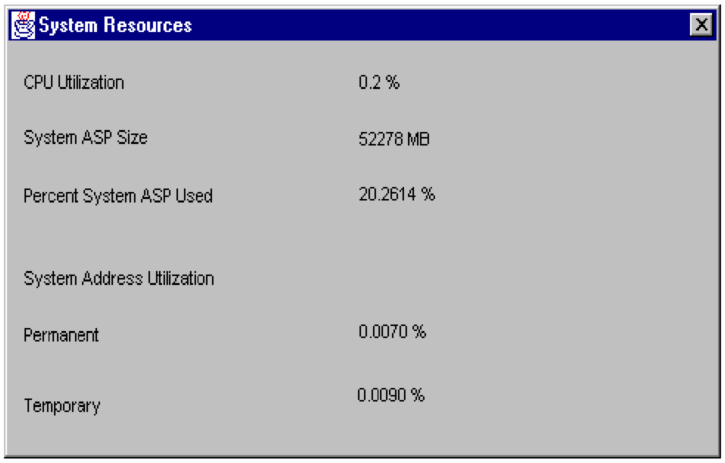


Figure 187. Correctly Installed Extension Showing the SystemStatus Menu Item

6. Select the menu option **SystemStatus**. After retrieving the data from the AS/400 system, the System Resource panel is displayed as shown in Figure 188 on page 238.


System Resources

CPU Utilization	0.2 %
System ASP Size	52278 MB
Percent System ASP Used	20.2614 %
System Address Utilization	
Permanent	0.0070 %
Temporary	0.0090 %

Figure 188. System Resources Panel

5.5.9 Adding a Second Panel to the PDML File

In this section, we modify the existing PDML file to include a second panel and embed the two panels in a tabbed pane. We follow these steps:

1. Start the GUI Builder.
2. Open the SystemStatus.pdml file that we previously created. We add a new Panel called PoolAllocation.
3. Add a table to the panel called SystemPool_Table. We set the table selection mode to "none".
4. Double-click on the table to edit its columns.
5. Modify the table to define the columns shown in Table 33. All columns have the editable property set to "false" and the Pool # column property Primary Column set to "true".

Table 33. Table Column Properties

Title	Data Class	Attribute
Pool #	SystemStatusEngine	SystemPoolNumber
Pool Name	SystemStatusEngine	SystemPoolName
Pool Size	SystemStatusEngine	SystemPoolSize
DB Faulting	SystemStatusEngine	DBFaulting
Non-DB Faulting	SystemStatusEngine	NonDBFaulting

6. We use the alignment tools so that when we preview the pane, it appears as shown in Figure 189 on page 239.

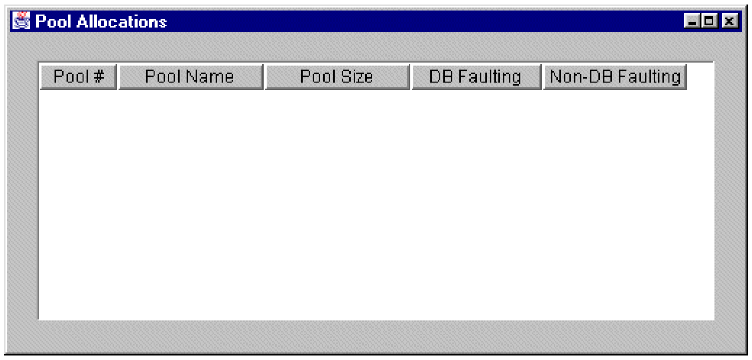


Figure 189. Preview of the PoolAllocation Panel

7. We add a new Panel to the PDML file called SystemStatus.
8. We add a Tabbed Pane to the Panel.
9. We add the two panels, SystemResources and PoolAllocation, to the tabbed pane.
10. We preview the tabbed pane and verify that it appears the same as the one in Figure 190.

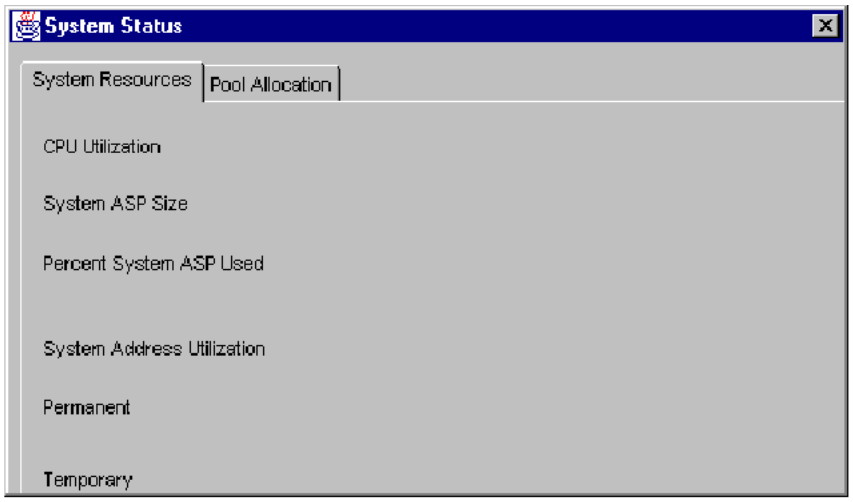


Figure 190. Preview of the Tabbed Panel

11. We save the PDML panel and exit the GUI Builder.

5.5.10 Modifying the SystemStatusEngine DataBean

Having successfully completed adding the second panel, we modify the Java source for the databean to perform the methods associated with the newly added panel.

The GUI Builder defines the instance variables shown in Table 34 on page 240.

Table 34. SystemStatusEngine Instance Variables

Name	Type
m_sSystemPoolNumber	String[]
m_idSystemPoolNumber	ItemDescriptor[]
m_sSystemPoolName	String[]
m_idSystemPoolName	itemDescriptor[]
m_sSystemPoolSize	String[]
m_idSystemPoolSize	ItemDescriptor[]
m_sDBFaulting	String[]
m_idDBFaulting	ItemDescriptor[]
m_sNonDBFaulting	String[]
m_idNonDBFaulting	ItemDescriptor[]

We need only modify the load() method so that it sets these variables correctly. The getPoolsNumber() method, used in an instance of the SystemStatus class, returns the number of system pools defined.

Using the getSystemPools() method in a SystemStatus object returns a vector of Enumeration. We can loop though the Enumeration object using the nextElement() method and cast the elements to a SystemPool object. Once cast, we use the appropriate method to extract the required information. For example, the getPoolIdentifier() method returns an integer that represents the system pool number.

Java Source Code

Section C.4, "SystemStatusEngine.java" on page 409, shows the code for the new load method.

We perform this series of steps:

1. Open the **SystemStatusEngine.java** file and locate the **load()** method.
2. Within the **try** block, add an integer variable called numOfPools and use the getPoolsNumber() method to retrieve the number of pools defined in aStatus.
3. Declare a variable named thePools of type java.util.Enumeration, and use the getSystemPools() method to retrieve the system pool objects.
4. Modify the assignments to the new class variables so that all arrays hold the same number of elements as the number of pools defined.
5. After the arrays are defined, use a while loop to cycle through all the elements of the thePools object. Extract the appropriate information, and assign it to the class variables.
6. We save, exit, and compile the Java program.

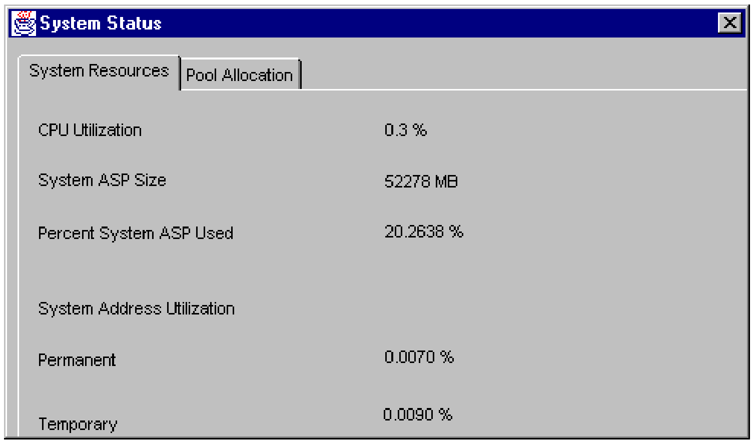
5.5.11 Modifying the SystemStatusManager

We need to modify the SystemStatusManager.java source file to display the panel that contains the tabbed pane. We only need to change one line of code:

```
pm = new PanelManager("SystemStatus","SystemStatus",dbeans,arg2);
```

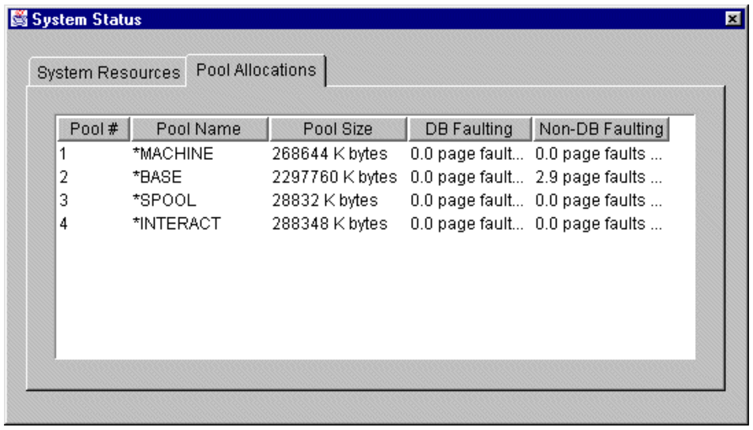
See Section C.5, “SystemStatusManager” on page 412, for the completed source listing for this task.

We compile the updated SystemStatusManager.java program. Now when we run the plug-in, it displays information as illustrated in Figure 191.



CPU Utilization	0.3 %
System ASP Size	52278 MB
Percent System ASP Used	20.2638 %
System Address Utilization	
Permanent	0.0070 %
Temporary	0.0090 %

Figure 191. System Resources Tab



Pool #	Pool Name	Pool Size	DB Faulting	Non-DB Faulting
1	*MACHINE	268644 K bytes	0.0 page fault...	0.0 page faults ...
2	*BASE	2297760 K bytes	0.0 page fault...	2.9 page faults ...
3	*SPOOL	28832 K bytes	0.0 page fault...	0.0 page faults ...
4	*INTERACT	288348 K bytes	0.0 page fault...	0.0 page faults ...

Figure 192. Pool Allocation Tab

5.6 PCML Examples

Program Call Markup Language (PCML) is a tag language that helps you call AS/400 programs, but with writing less Java code. PCML is based on the XML, a tag syntax that you write to describe the input and output parameters for AS/400 programs. PCML enables you to define tags that fully describe AS/400 programs that will be called by your Java application.

PCML was created to simplify calling AS/400 programs from Java. It performs the following functions:

- It converts data types between the AS/400 format and the Java format.
- It simplifies Java programs by handling complex relationships in AS/400 data.
  - Varying length character strings and structures
  - Varying size arrays of fields and structures and nested arrays
  - Strings with runtime CCSID tagging
- PCML is implemented as a package of Java classes:
  - com.ibm.as400.data(data400.jar)

Figure 193 shows how PCML works.

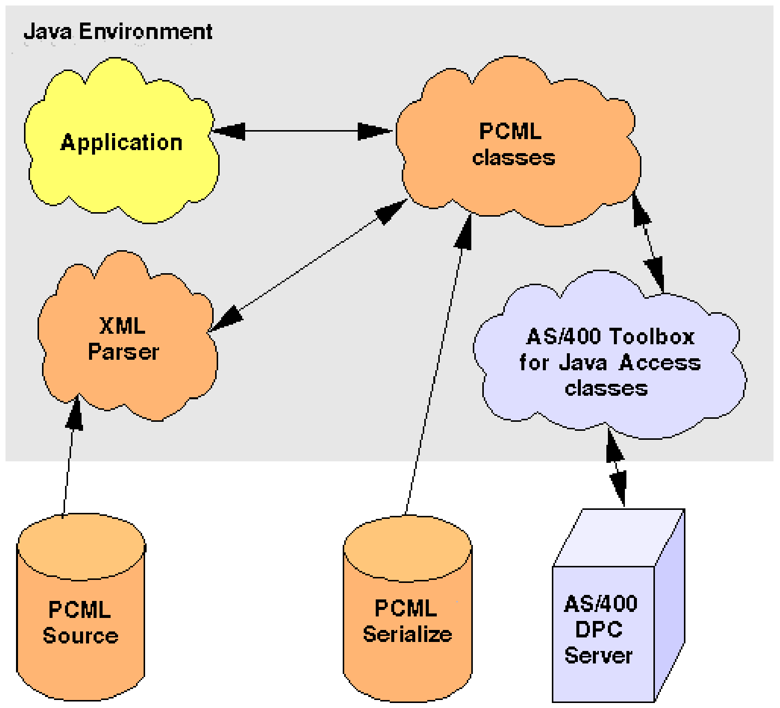


Figure 193. PCML Architecture

To build AS/400 program calls with PCML, you must start by creating:

- A Java application
- A PCML source file

Depending on your application design, you must write one or more PCML source files where you describe the interfaces to the AS/400 programs that will be called by your Java application. When your application constructs a ProgramCallDocument object, the IBM XML parser reads and parses the PCML source file. After the ProgramCallDocument class is created, the application program uses the ProgramCallDocument class methods to retrieve the necessary information from the AS/400 system through the AS/400 distributed program call (DPC) server.

To increase runtime performance, the ProgramCallDocument class can be serialized. The ProgramCallDocument is then constructed using the serialized file. In this case, the IBM XML parser is not used at runtime.

The following Java code constructs a ProgramCallDocument object:

```
AS400 as400 = new AS400();
ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400,
     "myPcmlDoc");
```

The ProgramCallDocument object looks for the PCML source in a file called myPcmlDoc.pcml. Notice that the .pcml extension is not specified on the constructor. If you are developing a Java application in a Java package, you can package-qualify the name of the PCML resource:

```
AS400 as400 = new AS400();
      ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400,
                 "com.company.package.myPcmlDoc");
```

5.6.1 A Simple PCML Example

PCML consists of the following tags, each of which has its own attribute tags:

- The **program tag** begins and ends code that describes one program.
- The **struct tag** defines a named structure, which can be specified as an argument to a program or as a field within another named structure. A struct tag contains a data or a struct tag for each field in the structure.
- The **data tag** defines a field within a program or structure.

Figure 194 shows a simple PCML file. It allows us to call a program named QZLSSTRS. It has no input parameters and one output parameter.

```
<pcml version="1.0">
<program name="StartNetServer"
         path="/QSYS.LIB/QZLSSTRS.PGM">
<data name="ErrorCode"  type="char" useage="output"/>
</program>
</pcml>
```

Figure 194. A Simple PCML File

Figure 195 shows how we call a program from Java using PCML. The Java application uses PCML by constructing a ProgramCallDocument object with a reference to the PCML source file. The ProgramCallDocument considers the PCML source file to be a Java resource. Consequently, the PCML source file is found using the Java CLASSPATH.

```
AS400 myAS400 = new AS400(...) ;
ProgramCallDocument myCall = new ProgramCallDocument
 (myAS400, "pcmlfile") ;
Boolean wasDone = myCall.callProgram("StartNetServer");
```

Figure 195. Calling a Program Using PCML

5.6.2 Calling the DPCXRPG Program Using PCML

In this section, we use PCML to call the RPG Program, DPCXRPG, described in Section 3.5.13, “Distributed Program Call (DPC) Application Example” on page 144. Figure 196 shows the PCML source file.

```
<pcml version="1.0">

 <!-- PCML source for calling "DPCXRPG" Program  -->

   <!-- Program PgmCall and its parameter list for retrieving part records -->
   <program name="PcmlPgmCall" path="/QSYS.lib/APILIB.lib/DPCXRPG.pgm">


    <data name="asFlag"       type="char"   length="1" usage="inputoutput"
          init=" "/>
    <data name="asPartNo"     type="packed" length="5" precision="0" usage="inputoutput"
          init=" "/>
    <data name="asDesc"       type="char"   length="25"  usage="inputoutput"
          init=" "/>
    <data name="asQty"        type="packed" length="5" precision="0" usage="inputoutput"
          init="0"/>
    <data name="asPrice"      type="packed" length="6" precision="2" usage="inputoutput"
          init="0.0"/>
    <data name="asDate"       type="char"   length="10"  usage="inputoutput"
          init="1999-01-01"/>

   </program>

 </pcml>
```

Figure 196. PCML Source File PcmlPgmCall.pcml

The name of the AS/400 program that we call is named DPCXRPG. It is found in the library named APILIB. DPCXRPG accepts and returns six parameters. In this example, we use it to retrieve information from rows in the Parts table. To request information from the AS/400 program, we set two input parameters:

- Flag = "S"
- Part Number = the part number (key) of the row to retrieve

The AS/400 program returns six parameters:

- Flag = "Y" if the row was found; "X" if the row was not found
- Part Number = The part number
- Description = Description of the part
- Quantity = Quantity in stock
- Price = Price
- Date= Received date

Notice that we specify the usage as "inputoutput" for all the parameters. This is because the AS/400 program also supports update, delete, and add capabilities. If we write another Java program to use these capabilities, we can use the same PCML file. Also notice that some of the parameters have a type of "packed". The AS/400 program expects and returns packed decimal parameters. The PCML support automatically handles the conversion for us.

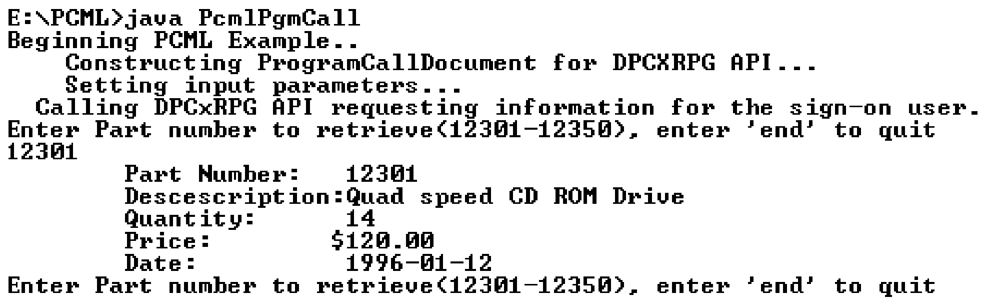
Figure 197 shows code snippets from a Java program, named PcmlPgmCall, that calls the AS/400 program using PCML. The entire program is available for download from the redbooks Web site.

```
 pcml = new ProgramCallDocument(as400System, "PcmlPgmCall");
 pcml.setValue("PcmlPgmCall.asFlag", "S");
pcml.setValue("PcmlPgmCall.asPartNo", partNo);
rc = pcml.callProgram("PcmlPgmCall");
 .
 .
 value = pcml.getValue("PcmlPgmCall.asFlag");
if (value.equals("Y")) {
value = pcml.getValue("PcmlPgmCall.asPartNo");
System.out.println("       Part Number:   " + value);
value = pcml.getValue("PcmlPgmCall.asDesc");
System.out.println("       Descescription:" + value);
value = pcml.getValue("PcmlPgmCall.asQty");
System.out.println("       Quantity:      " + value);
value = pcml.getValue("PcmlPgmCall.asPrice");
System.out.println("       Price:        $" + value);
value = pcml.getValue("PcmlPgmCall.asDate");
System.out.println("       Date:          " + value);
```

Figure 197. Java Code for PCML Program Call

We create a ProgramCallDocument object by passing it an AS/400 object and the name of the PCML source file. When an application constructs a ProgramCallDocument object, the IBM XML parser reads and parses the PCML source file. We use the setValue method of the ProgramCallDocument object to set the values for the flag and part number parameters. We use the callProgram method to actually call the AS/400 program. When the call to the AS/400 program is complete, we use the getValue method to retrieve the values of the parameters returned and write them to the console. Notice that we did not have to handle any data conversions.

Figure 198 shows an example of running the application.



```
E:\PCML>java PcmlPgmCall
Beginning PCML Example..
    Constructing ProgramCallDocument for DPCXRPG API...
    Setting input parameters...
  Calling DPCxRPG API requesting information for the sign-on user.
Enter Part number to retrieve(12301-12350), enter 'end' to quit
12301
        Part Number:   12301
        Descescription:Quad speed CD ROM Drive
        Quantity:      14
        Price:       $120.00
        Date:          1996-01-12
Enter Part number to retrieve(12301-12350), enter 'end' to quit
```

Figure 198. Running the PCML Example

5.6.3 PCML Conclusion

Calling AS/400 programs from Java applications can require a large amount of programming effort. Ordinarily, you have to write additional lines of code to construct AS/400 Toolbox for Java class objects for connecting to and retrieving information from an AS/400 program and for performing the appropriate data translation.

Using PCML, calls to AS/400 program are handled by PCML class objects. The PCML class objects are generated from PCML tags, which the PCML coded description of AS/400 programs calls. This minimizes the amount of code you need to write in order to call AS/400 programs from your application.

PCML provides a powerful way to call existing AS/400 programs. It can run in any 1.1.7 JVM that has the AS/400 Toolbox for Java Modification 2 and the PCML and XML parser archives available. You can write one PCML definition to use in all your Java programs. This allows you to reuse it and makes it easier to maintain. It can also serialize the PCML file for improved runtime performance. To serialize a PCML file, use the ProgramCallDocument class with the -serialize option:

```
Java com.ibm.ProgramCallDocument -serialize pcmlfile
```

The system will use the serialized file if it exists. Also, the IBM XML parser packages are not required at runtime.

While PCML was designed to support distributed program calls to AS/400 program objects from a Java client platform, you can also use PCML to make calls to an AS/400 program from within an AS/400 environment.

5.7 JDBC 2.0

The JDBC 2.0 API is the latest update of the JDBC API. The JDBC 2.0 API contains many new features, including scrollable result sets. There are two parts to the JDBC 2.0 API: the JDBC 2.0 Core API and the JDBC 2.0 Standard Extension API. The JDBC 2.0 Core API is included in the Java 2 (JDK 1.2) Platform release.

The JDBC 2.0 API has been factored into two complementary components. The first component, which is termed the JDBC 2.0 Core API, comprises the updated contents of the java.sql package. The second component, termed the JDBC 2.0 Standard Extension API, comprises the contents of a new package, javax.sql, which as its name implies will be delivered as a Java Standard Extension.

The java.sql package contains all of the enhancements that have been made to the existing JDBC interfaces and classes, in addition to a few new classes and interfaces. The new javax.sql package has been introduced to contain the parts of the JDBC 2.0 API that are closely related to other pieces of the Java platform. The parts are standard extensions, such as the Java Naming and Directory Interface (JNDI), and the Java Transaction Service (JTS). In addition, some advanced features that are easily separable from the JDBC Core API, such as connection pooling and rowsets, have also been added to javax.sql. Putting these advanced facilities into a standard extension, instead of into a core, helps keep the JDBC Core API small and focused.

Since the standard extensions are downloadable, it will always be possible to deploy an application that uses the features in the JDBC standard extension that will "run anywhere". If a standard extension is not installed on a client machine, it can be downloaded along with the application that uses it.

The AS/400 Toolbox for Java supports the JDBC 2.0 Core API. If you want to use any of the following JDBC 2.0 enhancements, you also need to use Java 2 (JDK 1.2):

- BLOB interface
- CLOB interface
- Scrollable result sets
- Updatable result sets
- Batch update capability with Statement, PreparedStatement, and CallableStatement objects

To use BLOB and CLOB support with the AS/400 system, the AS/400 Universal Data Base (UDB) is required. At the time of the writing of this redbook, it was anticipated that this support would be available in fourth quarter 1999.

5.7.1 JDBC Result Sets

A result set created by executing a statement may support the ability to move backward (last-to-first), as well as forward (first-to-last), through its contents. Result sets that support this capability are called *scrollable result sets*. Result sets that are scrollable also support relative and absolute positioning. *Absolute positioning* is the ability to move directly to a row by specifying its absolute position in the result set. *Relative positioning* gives the ability to move to a row by specifying a position that is relative to the current row. The definition of absolute and relative positioning in JDBC 2.0 is modeled on the X/Open SQL CLI specification.

The JDBC 1.0 API provided one result set type, forward-only. The JDBC 2.0 API provides three result set types: forward-only, scroll-insensitive, and scroll-sensitive. As their names suggest, the new result set types support scrolling, but they differ in their ability to make changes visible while they are open.

A scroll-insensitive result set is generally not sensitive to changes that are made while it is open. A scroll-insensitive result set provides a static view of the underlying data it contains. The membership, order, and column values of rows in a scroll-insensitive result set are typically fixed when the result set is created. On the other hand, a scroll-sensitive result set is sensitive to changes that are made while it is open, and provides a "dynamic" view of the underlying data. For example, when using a scroll-sensitive result set, changes in the underlying column values of rows are visible. The membership and ordering of rows in the result set may be fixed. This is implementation defined.

An application may choose from two different concurrency types for a result set: read-only and updatable. A result set that uses read-only concurrency does not allow updates of its contents. This can increase the overall level of concurrency between transactions, since any number of read-only locks may be held on a data item simultaneously. A result set that is updatable allows updates and may use database write locks to mediate access to the same data item by different transactions. Since only a single write lock may be held at a time on a data item, this can reduce concurrency. Alternatively, an optimistic concurrency control scheme may be used if conflicting access to data will be rare. Optimistic concurrency control implementations typically compare rows either by value or by a version number to determine if an update conflict has occurred.

Two performance hints may be given to a JDBC 2.0 driver to make access to result set data more efficient. Specifically, the number of rows to be fetched from the database each time more rows are needed can be specified. A direction for processing the rows—forward, reverse, or unknown—can be given as well. These values can be changed for an individual result set at any time. A JDBC driver may ignore a performance hint if it chooses. The AS/400 Toolbox for Java JDBC 2.0 driver implements the number of rows fetched, but not the direction for processing.

5.7.2 Using Scrollable and Updatable Result Sets

If a result set is created by executing a statement or prepared statement, you can move (scroll) backward (last-to-first) or forward (first-to-last) through the rows in a table. Figure 199 shows a code snippet, which uses a scrollable result set.

```
private java.sql.ResultSet rs;
private java.sql.PreparedStatement s;

s = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTQY > ? ",
            java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE,
            java.sql.ResultSet.CONCUR_READ_ONLY);
s.setFetchSize(25);
s.setInt(1,1000);
rs = s.executeQuery();
rs.next ();
.
rs.previous();
.
rs.first();
.
rs.last();
.
boolean islast = rs.isLast();
.
boolean setPosition = rs.absolute(10);
```

Figure 199. Scrollable Result Set Example

The prepareStatement method now has two additional parameters. We set them to scroll insensitive and read only. We use the setFetchSize method to set the number of rows to be fetched from the database when more rows are needed. This may be changed at any time. If the value specified is zero, then the driver will choose an appropriate fetch size.

There are a number of new methods available for moving through the result set. Some of them include:

- **Next** — Positions the cursor to the next row.
- **Previous** — Positions the cursor to the previous row.
- **First** — Positions the cursor to the first row of the result set.
- **Last** — Positions the cursor to the last row of the result set.
- **IsLast** — Indicates if the cursor is positioned on the last row.
- **Absolute** — Positions the cursor to an absolute row number. Attempting to move beyond the first row positions the cursor before the first row. Attempting to move beyond the last row positions the cursor after the last row. If the absolute row number is positive, this positions the cursor with respect to the

beginning of the result set. If the absolute row number is negative, this positions the cursor with respect to the end of result set.

Figure 200 shows a code snippet, which updates a row from a result set.

```
private java.sql.ResultSet rs;
private java.sql.PreparedStatement s;

s = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTQY > ? FOR UPDATE",
              java.sql.ResultSet.TYPE_SCROLL_SENSITIVE,
              java.sql.ResultSet.CONCUR_UPDATABLE);

s.setInt(1,1000);
rs = s.executeQuery();
rs.next ();
.
rs.previous();
String name = rs.getString("PARTNO");
int qty = rs.getInt("PARTQY");
.
.     // application logic
.
rs.updateInt("PARTQY", qty);   // Update the quantity with a new value

rs.updateRow ();  // Send the updates to the server.
```

Figure 200. Updatable Result Set Example

In this case, we do not set a fetch size because the toolbox JDBC driver will return only one row at a time from the AS/400 system. The host server will lock the row on which we are positioned. We use the updateRow method to update the row.

5.7.3 JDBC 2.0 Example

In this section, we modify the JDBCExample and JDBCExampleDisplayAll classes discussed in Section 3.5.4, “JDBCExample Class” on page 116, to use scrollable result sets.

Figure 201 on page 250 shows the connectToDB method of the JDBCExample class.

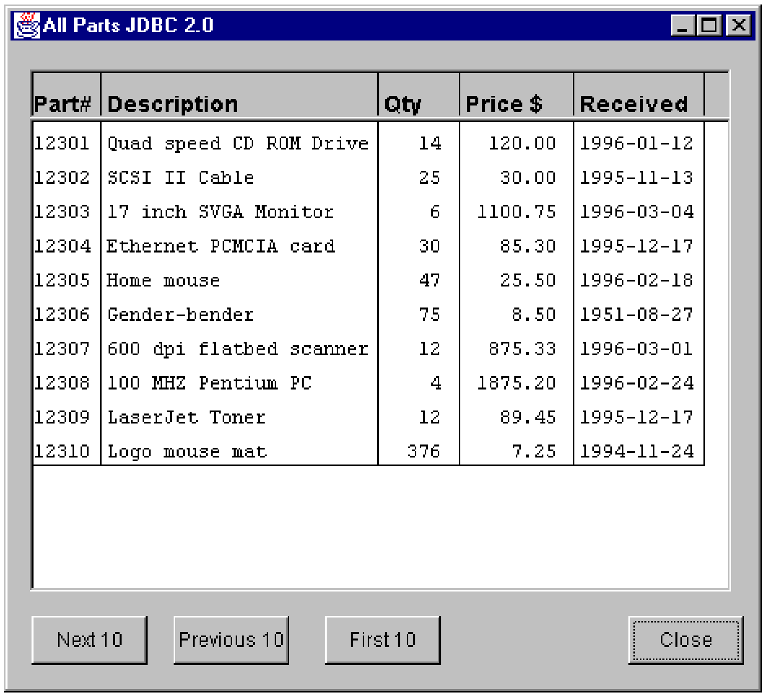
```
public String connectToDB(String systemName, String userid, String password)
{
try
{
setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.WAIT_CURSOR));
java.sql.DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName +
            "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
            "package=JDBCExa;package library=apilib", userid, password);

psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?");

psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS ORDER BY PARTNO",
                        java.sql.ResultSet.TYPE_SCROLL_SENSITIVE,
                          java.sql.ResultSet.CONCUR_READ_ONLY);
psAllRecord.setFetchSize(10);

psUpdateRecord = dbConnect.prepareStatement("UPDATE PARTS SET PARTDS = ?," + " PARTQY = ?, PARTPR = ?,
                 PARTDT = ? WHERE PARTNO = ?");
psAddRecord = dbConnect.prepareStatement("INSERT INTO PARTS (PARTDS, PARTQY," + " PARTPR, PARTDT,
                 PARTNO) VALUES(?, ?, ?, ?, ?)");
psDeleteRecord = dbConnect.prepareStatement("DELETE FROM PARTS WHERE PARTNO = ?");
}
catch (Exception e){
          e.printStackTrace();
          showException(e);
          setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
          return "Connect Failed.";
          }
setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
return "Connected to AS/400.";
}
```

Figure 201. The connectToDB Method

The prepareStatement method for the psAllRecord PreparedStatement object has two additional parameters. We set them to scroll sensitive and read only. *Scroll sensitive* means that any changes to the database will be reflected in the result set as we scroll through it. The CONCUR_READ_ONLY option allows us to only read rows from the database. We cannot update them. This setting allows the JDBC driver to function more efficiently because it can retrieve multiple rows at a time from the server. If we set this value to Concur_Updatable, the server returns only one row at a time. We use the setFetchSize method to set the number of rows to be fetched from the database (when more rows are needed) to ten. This option can be valuable when the SQL statement executed returns a very large result set. In JDBC 1.0, all the rows were returned to you. Now, you can control the number of rows returned programmatically. As you scroll through the result set, the driver retrieves the number of rows that you specify as they are needed.

Figure 202 on page 251 shows the new JDBCExampleDisplayAll display. Three new buttons have been added to demonstrate some of the new JDBC 2.0 capabilities:

- Next 10
- Previous 10
- First 10



Part#	Description	Qty	Price $	Received
12301	Quad speed CD ROM Drive	14	120.00	1996-01-12
12302	SCSI II Cable	25	30.00	1995-11-13
12303	17 inch SVGA Monitor	6	1100.75	1996-03-04
12304	Ethernet PCMCIA card	30	85.30	1995-12-17
12305	Home mouse	47	25.50	1996-02-18
12306	Gender-bender	75	8.50	1951-08-27
12307	600 dpi flatbed scanner	12	875.33	1996-03-01
12308	100 MHZ Pentium PC	4	1875.20	1996-02-24
12309	LaserJet Toner	12	89.45	1995-12-17
12310	Logo mouse mat	376	7.25	1994-11-24

Figure 202. JDBCExampleDisplayALL

Figure 203 shows the populateListBoxNextTen method, which supports the Next 10 button.

```
public void populateListBoxNextTen() {
int i = 0;
try
{
      if (!(psAllRecordRS.next())) {/* read past the end of the result set*/
                psAllRecordRS.absolute(psAllRecordRS.getRow() - 1); /*return to last row */
                return;     /* Just leave current screen up  */
              }

      ivjIMulticolumnListbox1.removeAllRows();
      do
      {
         String[] array = new String[5];
         array[0] = psAllRecordRS.getString("PARTNO");
         array[1] = psAllRecordRS.getString("PARTDS");
         array[2] = insertSpaces(Integer.toString(psAllRecordRS.getInt("PARTQY")), 5);
         array[3] = insertSpaces(psAllRecordRS.getBigDecimal("PARTPR", 2).toString(), 8);
         array[4] = psAllRecordRS.getDate("PARTDT").toString();
         ivjIMulticolumnListbox1.addRow(array, array[0]);
         i++;
      }while ((i < 10) && (psAllRecordRS.next()));
}
catch (Exception e)
{
showException(e);
}
return;
}
```

Figure 203. The populateListBoxNextTen Method

If we are already at the end of the result set, we use the absolute method to position to the last row and return. Otherwise, we use the next method to retrieve up to ten rows from the result set and add them to the list box.

Figure 204 shows the populateListBoxPrevTen method. This method supports the Previous 10 button.

```
public void populateListBoxPrevTen() {
int i = 0;

try
{
     psAllRecordRS.relative(-20);
     if(psAllRecordRS.getRow() == 0)
            psAllRecordRS.beforeFirst();   /* position before first */
     ivjIMulticolumnListbox1.removeAllRows();
     while ((i < 10) && psAllRecordRS.next())
           {String[] array = new String[5];
          array[0] = psAllRecordRS.getString("PARTNO");
          array[1] = psAllRecordRS.getString("PARTDS");
          array[2] = insertSpaces(Integer.toString(psAllRecordRS.getInt("PARTQY")), 5);
          array[3] = insertSpaces(psAllRecordRS.getBigDecimal("PARTPR", 2).toString(), 8);
          array[4] = psAllRecordRS.getDate("PARTDT").toString();
          ivjIMulticolumnListbox1.addRow(array, array[0],i);
          i++;
     }
}
catch (Exception e)
{
showException(e);
}
return;
}
```

Figure 204. The populateListBoxPrevTen Method

We move the cursor backwards by 20 rows with the psAllRecordRS.relative(-20) statement. Notice that the getRow method returns the absolute row number. It returns zero if we are not positioned on a row. We can also point directly to a specific row with the absolute method. We can point the cursor to before the first row with the beforeFirst method.

Figure 205 on page 253 shows the populateFirstTen method. This method supports the First 10 button.

```
public void populateFirstTen() {
int i = 0;

try
{
        psAllRecordRS.first();
        ivjIMulticolumnListbox1.removeAllRows();
        do
        {
         String[] array = new String[5];
         array[0] = psAllRecordRS.getString("PARTNO");
         array[1] = psAllRecordRS.getString("PARTDS");
         array[2] = insertSpaces(Integer.toString(psAllRecordRS.getInt("PARTQY")), 5);
         array[3] = insertSpaces(psAllRecordRS.getBigDecimal("PARTPR", 2).toString(), 8);
         array[4] = psAllRecordRS.getDate("PARTDT").toString();
         ivjIMulticolumnListbox1.addRow(array, array[0]);
         i++;
         }while ((i < 10) && (psAllRecordRS.next()));
}
catch (Exception e)
{
showException(e);
}
return;
}
```

Figure 205. The populateFirstTen Method

Notice that the first() method positions the cursor to the first row of the result set. We then use the next() method to retrieve nine more rows from the result set and add them to the list box.

5.8 Additional Classes

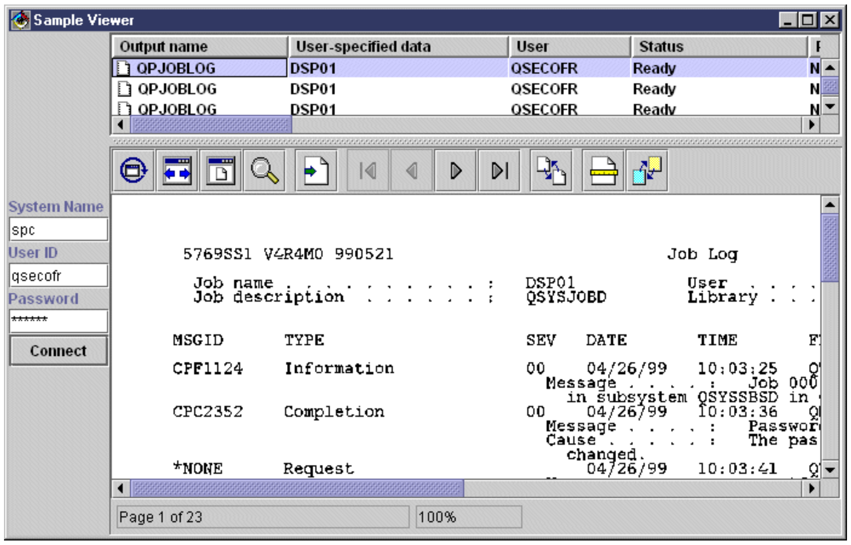
This section describes and demonstrates some of the new or enhanced classes available with Modification 2 of the AS/400 Toolbox for Java. All sample applications were written using IBM VisualAge for Java 2.0 with rollup2, enterprise update, and Modification 2 of the AS/400 Toolbox for Java loaded in the workspace. For instructions on how to load these features, see Section 5.1, “Upgrading the AS/400 Toolbox Contained in VisualAge for Java 2.0” on page 213.

5.8.1 SpooledFileViewer

As the class name implies, the SpooledFileViewer class displays AS/400 spooled files on a client workstation. There are two AS/400 requirements for this class:

- The spooled file must reside on a V4R4 or later AS/400 system.
- The AFP Utilities licensed product must be installed on the AS/400 system.

The SpooledFileViewer class is a subclass of the com.sun.java.swing.JComponent. It can be added to suitable swing classes (such as JFrame). Figure 206 on page 254 illustrates a possible use of this class. We use a JSplitPane to hold an AS400DetailPane on top and a SpooledFileViewer on the bottom. The AS400DetailPane is populated with a list of AS/400 spooled files. Selecting a file in the AS400DetailPane causes its contents to be displayed in the bottom pane.



Output name	User-specified data	User	Status
QPJOBLOG	DSP01	QSECOFR	Ready
QPJOBLOG	DSP01	QSECOFR	Ready
QPJOBLOG	DSP01	QSECOFR	Ready

```
 5769SS1 V4R4M0 990521                                    Job Log

  Job name . . . . . . . . . . :   DSP01            User  . . . .
  Job description  . . . . . . :   QSYSJOBD         Library . . .

MSGID       TYPE                    SEV   DATE        TIME        F
CPF1124     Information             00    04/26/99    10:03:25    Q
                                       Message . . . . :    Job 000
                                         in subsystem QSYSSBSD in
CPC2352     Completion              00    04/26/99    10:03:36    Q
                                       Message . . . . :    Passwor
                                       Cause . . . . . :    The pas
                                         changed.
*NONE       Request                       04/26/99    10:03:41    Q
```

Figure 206. Application Using the SpooledFileViewer Class

5.8.1.1 Building the Spooled Application

For this example, it is necessary to extend the com.ibm.as400.vaccess.SpooledFileViewer class to add two methods. We do this to enable an application to pass the AS400DetailsPlane selected object and the VPrinterObject object to the extended SpooledFileViewer class. It converts them to the required parameters for the setSpooledFile method provided with the base SpooledFileViewer class. The base SpooledFileViewer does not directly accept AS400DetailsPanes and VPrinterObjects. The sample application is built by extending the existing SpooledFileViewer class to provide two additional setSpooledFile methods. The extended class is used inside a JSplitPanel to create the application. The following steps show how to build the application:

1. Start IBM VisualAge for Java 2.0
2. From the Workspace window, create a new project called New Toolbox Samples.
3. Create a package called SpooledViewer inside the previously created project.
4. Create a new class called ESpooledFileViewer that extends com.ibm.as400.vaccess.SpooledFileViewer and imports the com.ibm.as400.access and com.ibm.as400.vaccess packages.
5. Create a new void method called setSpooledFile that accepts two parameters:

   - **system** of type AS400
   - **splf** of type VOutput

   The method should also be capable of throwing the exception java.beans.PropertyVetoException.

   The following code snippet forms the body of the previously created method. It uses the VOutput and AS400 objects to create a SpooledFile object that is used to call the inherited setSpooledFile(SpooledFile) method.

   ```
   String splfString = splf.toString();
   String splfName = splfString.substring(0, splfString.indexOf(" "));
   splfString = splfString.substring(splfString.indexOf(" ")).trim();
   ```

```
int splfNum = new Integer(splfString.substring(0,splfString.indexOf("
"))).intValue();
splfString = splfString.substring(splfString.indexOf(" ")).trim();
String splfUser = splfString.substring(0, splfString.indexOf(" "));
splfString = splfString.substring(splfString.indexOf(" ")).trim();
String splfJobNum = splfString.substring(0, splfString.indexOf(" "));
String splfJobName = splfString.substring(splfString.indexOf(" ")).trim();
setSpooledFile(new SpooledFile(system ,splfName, splfNum, splfJobName,
splfUser, splfJobNum));
```

6. Create another new void method called setSpooledFile that accepts two slightly different parameters:
   - **system** of type AS400
   - **splf** of type VObject

   Again, the method should be capable of throwing the java.beans.PropertyVetoException exception.
7. This method simply calls the previously created method by casting the VObject to a VOutput object, as shown in the following code snippet:

   ```
   setSpooledFile(system, (VOutput) splf)
   ```

Figure 207 shows the complete ESpooledFileViewer class.

```
import com.ibm.as400.vaccess.*;
import com.ibm.as400.access.*;
public class ESpooledFileViewer extends SpooledFileViewer{
public void setSpooledFile(VOutput aVSplf,VPrinterOutput aVSplfList )
        throws java.beans.PropertyVetoException {
  String splf = aVSplf.toString();
  String splfName = splf.substring(0, splf.indexOf(" "));
  splf = splf.substring(splf.indexOf(" ")).trim();
  int splfNum = new Integer(splf.substring(0,splf.indexOf(" "))).intValue();
  splf = splf.substring(splf.indexOf(" ")).trim();
  String splfUser = splf.substring(0, splf.indexOf(" "));
  splf = splf.substring(splf.indexOf(" ")).trim();
  String splfJobNum = splf.substring(0, splf.indexOf(" "));
  String splfJobName = splf.substring(splf.indexOf(" ")).trim();
  setSpooledFile(new SpooledFile(aVSplfList.getSystem(),splfName, splfNum,
          splfJobName, splfUser, splfJobNum));
}
public void setSpooledFile (VObject aVSplf, VObject aVList) throws
java.beans.PropertyVetoException {
if (aVSplf!=null) {
setSpooledFile((VOutput) aVSplf, (VPrinterOutput) aVList);
}
}
}
```

Figure 207. The ESpooledFileViewer Class

Next, we use the VisualAge for Java Visual Composition Editor to create an end-user application that uses the ESpooledFileViewer class to display spooled files. Figure 208 on page 256 shows the JavaBeans that make up the new application.

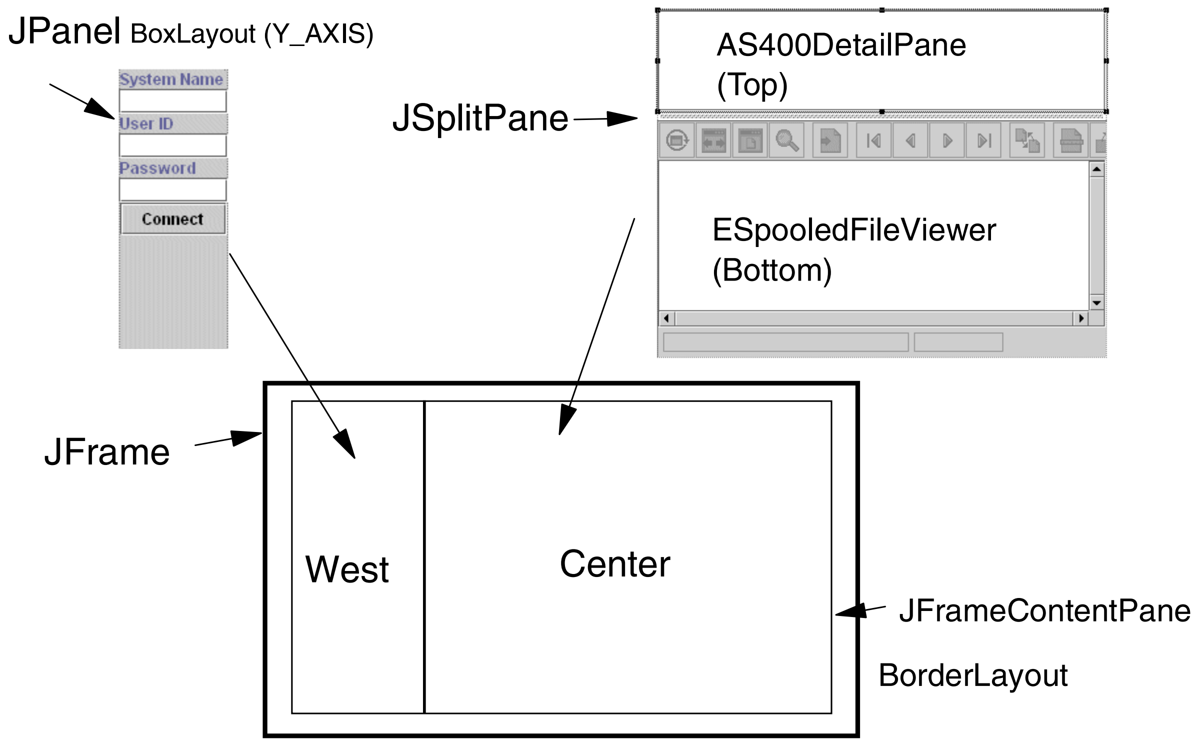


Figure 208. The SampleViewer Class in the IDE

We follow this process:

1. In the VisualAge for Java IDE, create a new class named SampleViewer that extends com.sun.java.swing.JFrame and select the Compose the class visually radio box.
2. Click on **Finish** to generate the class and start the Visual Composition Editor.
3. Set the JFrameContentPane, inside the JFrame, to use the Border layout.
4. Add a new JPanel to the free form space. It should not be added to the JFrameContentPane yet.
5. Add the Swing components (see Table 35) in sequence to the JPanel.

Table 35. Swing Components

Component	Text Property
JLabel	System Name
JTextField	
JLabel	User ID
JTextField	
JLabel	Password
JPasswordField	
JButton	Connect

6. Change the panel layout manager to BoxLayout (with the BoxLayout axis property set to" Y_AXIS:). It should appear the same as shown in Figure 209.

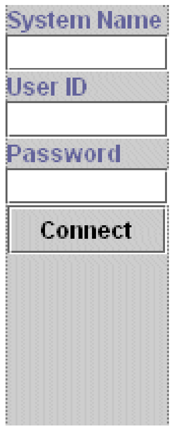


Figure 209. System and User Details Panel

7. Add the panel to the JFrameContentPane's west side.
8. Create a new JSplitPane in the free form space. Again, be careful not to add it to the JFrameContentPane yet. Set the orientation property of the JSplitPane to VERTICAL_SPLIT.
9. Use the Choose a bean tool to add a ESpooledFileViewer to the bottom portion of the JSplitPane object.
10. Add an AS400DetailsPane to the top portion of the JSplitPane object. Set the JSplitPane dividerLocation property to 30. The JSplitPane should appear similar to Figure 210.

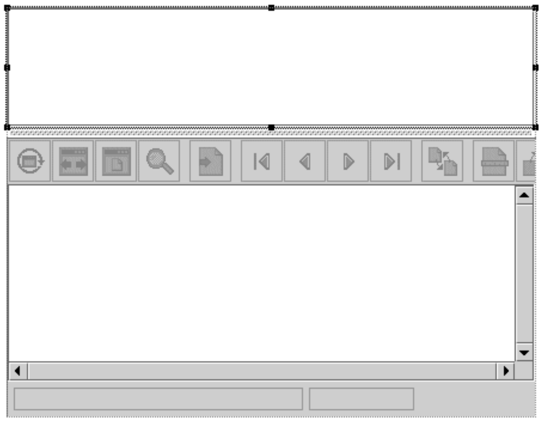


Figure 210. The JSplitPane

11. Add the JSplitFrame to the center of the JFrameContentPane. Again, at this time, you may need to change the JSplitPane dividerLocation property to display both panels correctly.

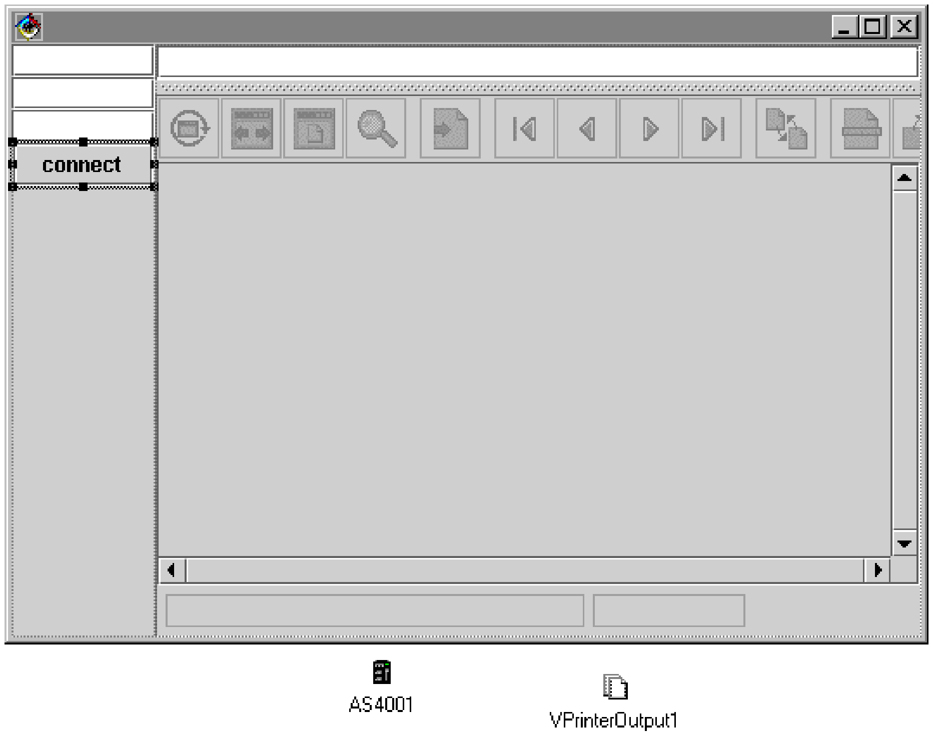


Figure 211. Adding AS400 and VPrinterOutput Objects

12.As shown in Figure 211, add an AS400 System and a VPrinterOutput object to the free space of the Visual Composition Editor.

13.Add the connections in sequence as shown in Table 36.

Table 36. VCE Connections

From Object	From Event or Property	To Object	To Property or Method
JButton1	actionPerformed	AS4001	setSystemName()
Previous connection	arg1	JTextField1	text
JButton1	actionPerformed	AS4001	setUserId()
Previous connection	arg1	JTextField2	text
JButton1	actionPerformed	AS4001	setPassword
Previous connection	arg1	JPasswordField1	text
JButton1	actionPerformed	VPrinterOutput1	setSystem
Previous connection	arg1	AS4001	this
JButton1	actionPerformed	VPrinterOutput1	load()
JButton1	actionPerformed	AS400DetailsPane1	setRoot()
JButton1	actionPerformed	AS400DetailsPane1	load()
Previous connection	arg1	VPrinterOutput1	this
AS400DetailsPane1	listSelection	MySpooledFileViewer1	setSpooledFile(AS400, VObject)

From Object	From Event or Property	To Object	To Property or Method
Previous connection	system	AS4001	this
Same connection as the previous connection	splf	AS400DetailsPane1	getSelectedObject()
AS400DetailsPane1	listSelection	MySpooledFileViewer1	load()

Figure 212 shows the resulting connections within the VCE.

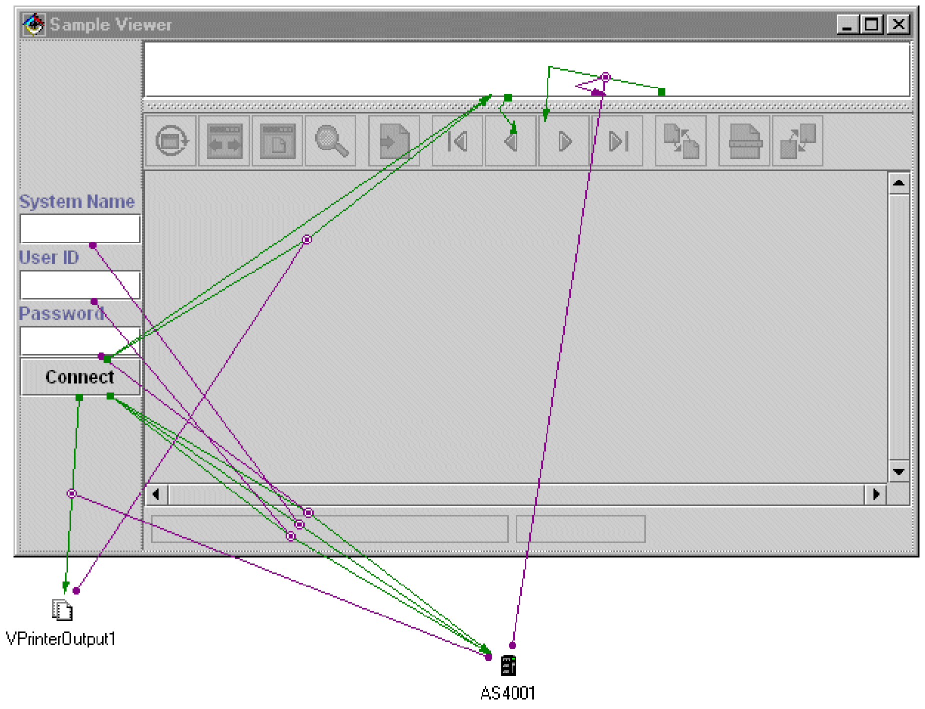
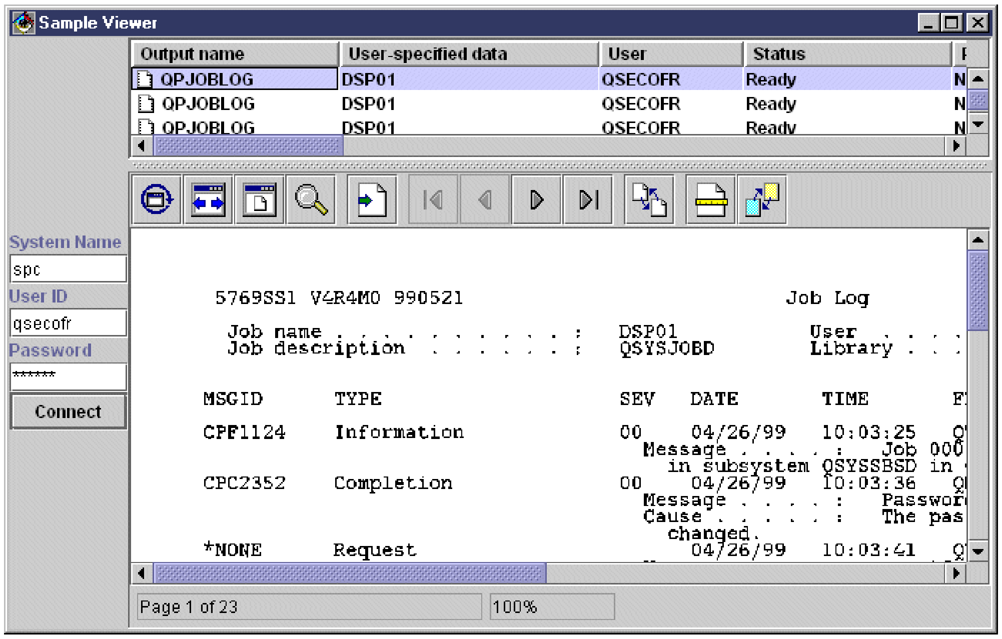


Figure 212. SampleViewer Class with Completed Connections

14. Run the application.

Once loaded, maximize the window so you can see both subcomponents of the JSplitPane correctly.

Verify that it will connect to an AS/400 system. Once the list of available spooled files is obtained from the server, selecting a spooled file in the details pane should produce the MySpooledFileViewer display as shown in Figure 213 on page 260.



System Name: spc
User ID: qsecofr
Password: ******

Output name	User-specified data	User	Status	F
QPJOBLOG	DSP01	QSECOFR	Ready	N
QPJOBLOG	DSP01	QSECOFR	Ready	N
QPJOBLOG	DSP01	QSECOFR	Ready	N

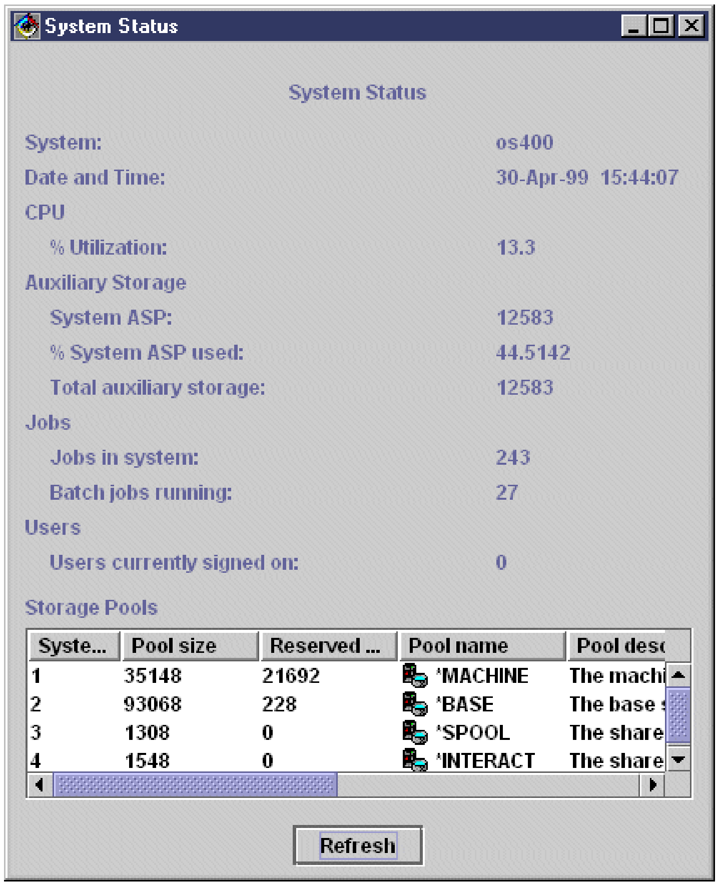
```
   5769SS1 V4R4M0 990521                               Job Log

    Job name . . . . . . . . . . :   DSP01            User  . . . .
    Job description  . . . . . . :   QSYSJOBD         Library . . .

  MSGID      TYPE                    SEV   DATE        TIME        F
  CPF1124    Information             00    04/26/99    10:03:25    Q
                                       Message . . . . :    Job 000
                                         in subsystem QSYSSBSD in
  CPC2352    Completion              00    04/26/99    10:03:36    Q
                                       Message . . . . :    Passwor
                                       Cause . . . . . :    The pas
                                         changed.
  *NONE      Request                       04/26/99    10:03:41    Q
```

Page 1 of 23   100%

Figure 213. Running the Application

5.8.2 VSystemStatusPane

The VSystemStatusPane is another JComponent that can be added to a suitable container (such as a JFrame). As the name implies, the VSystemStatusPane is used to display information about the current system status on an AS/400 system. Adding a refresh button to the panel allows the user to monitor the AS/400 status at various points in time. An example application is shown in Figure 214 on page 261.


System Status

System Status

System:	os400
Date and Time:	30-Apr-99 15:44:07
CPU	
% Utilization:	13.3
Auxiliary Storage	
System ASP:	12583
% System ASP used:	44.5142
Total auxiliary storage:	12583
Jobs	
Jobs in system:	243
Batch jobs running:	27
Users	
Users currently signed on:	0

Storage Pools

Syste...	Pool size	Reserved ...	Pool name	Pool desc
1	35148	21692	*MACHINE	The machi
2	93068	228	*BASE	The base s
3	1308	0	*SPOOL	The share
4	1548	0	*INTERACT	The share

Figure 214. Example Using the VSystemStatusPanel Class

5.8.2.1 Building the Sample Application

The sample application uses a bean factory to overcome a missing null constructor method for the VSystemStatusPane class. The following steps outline how to build this sample application:

1. Start VisualAge for Java 2.0
2. Within the **New Toolbox Samples** project, add a new package named StatusExample.
3. Create a new class in this package called Status. This class should extend com.sun.java.swing.JFrame. Select the **Compose the class visually** radio box.
4. Click on the **Finish** button to generate the class and start the Visual Composition Editor.
5. Set the JFrameContentPane to use the Border layout manager.
6. Add a JPanel component to the Center of the JFrameContentPane.
7. Set the JPanel to use FlowLayout.
8. Add a button to the JPanel, and set the text property to Refresh.
9. Set the JPanel constraints property to South.
10. Add an AS400 object to the free form area of the VCE.

11. Within the VCE, select the **Factory** bean ( ), from the **Other** palette. Add it to the free form area of the VCE.

12. Right click on the **Factory** bean and select the **Change Type...** option. Set the type to com.ibm.as400.vaccess.VSystemStatusPane.

13. Connect the Status initialize event to the Factory1 constructor method.

    To do this, right-click on the JFrame and select the **Connect—>Connectable Features—>initialize** event. Then, click on the **Factory1** object and select the **VSystemStatusPane(com.ibm.as400.access.AS400)** option.

14. Pass the AS4001 object as the parameter for the previous connection.

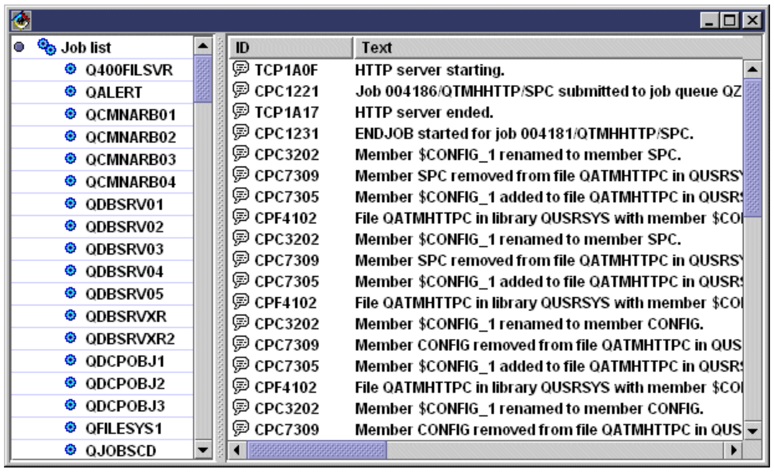
15. Connect the Status initialize event to the add(Component) method for the JFrameContentPane. Pass in the Factory as the component parameter.

16. Connect the actionPerformed event of the JButton1 object to the load() method for the Factory1 object.

17. Save and run the sample application. Click on the **Refresh** button to cause the sign on dialog to appear.

5.8.3 Jobs and Job Logs

The VJobList class is very simple to use. Once added to an AS400ExplorerPane, it can be used to display and select jobs running on an AS/400 system. Selecting a job triggers the joblog messages to be displayed in the right-hand pane of the AS400ExplorerPane. Figure 215 shows the output generated once the VJobList loads data from an AS/400 system.



```
Job list
    Q400FILSVR
    QALERT
    QCMNARB01
    QCMNARB02
    QCMNARB03
    QCMNARB04
    QDBSRV01
    QDBSRV02
    QDBSRV03
    QDBSRV04
    QDBSRV05
    QDBSRVXR
    QDBSRVXR2
    QDCPOBJ1
    QDCPOBJ2
    QDCPOBJ3
    QFILESYS1
    QJOBSCD
```

```
ID        Text
TCP1A0F   HTTP server starting.
CPC1221   Job 004186/QTMHHTTP/SPC submitted to job queue QZ
TCP1A17   HTTP server ended.
CPC1231   ENDJOB started for job 004181/QTMHHTTP/SPC.
CPC3202   Member $CONFIG_1 renamed to member SPC.
CPC7309   Member SPC removed from file QATMHTTPC in QUSRS
CPC7305   Member $CONFIG_1 added to file QATMHTTPC in QUSR
CPF4102   File QATMHTTPC in library QUSRSYS with member $CO
CPC3202   Member $CONFIG_1 renamed to member SPC.
CPC7309   Member SPC removed from file QATMHTTPC in QUSRS
CPC7305   Member $CONFIG_1 added to file QATMHTTPC in QUSR
CPF4102   File QATMHTTPC in library QUSRSYS with member $CO
CPC3202   Member $CONFIG_1 renamed to member CONFIG.
CPC7309   Member CONFIG removed from file QATMHTTPC in QUS
CPC7305   Member $CONFIG_1 added to file QATMHTTPC in QUSR
CPF4102   File QATMHTTPC in library QUSRSYS with member $CO
CPC3202   Member $CONFIG_1 renamed to member CONFIG.
CPC7309   Member CONFIG removed from file QATMHTTPC in QUS
```

Figure 215. A VJobList Example

5.8.3.1 Building the Sample

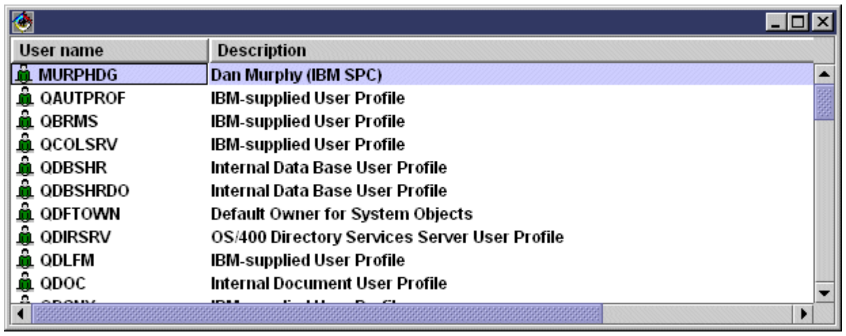
To build the VJobList sample, perform these steps:

1. Start IBM VisualAge for Java 2.0.
2. Within the **New Toolbox Samples** project, add a new package named UsersExample.
3. Create a new class in this package called Jobs. This class should extend com.sun.java.swing.JFrame.

4. Select the **Compose the class visually** radio box and click on the **Finish** button.
5. Within the VCE, set the JFrameContentPane to use the Border layout manager.
6. Add an AS400ExplorerPane to the Center of the JFrameContentPane.
7. Add an AS400 object to the free form area of the VCE.
8. Add a com.ibm.as400.vaccess.VJobList to the free form area of the VCE.
9. Connect the Jobs initialize event to the VJobList.setSystem(AS400) method, and pass the AS4001 object as the parameter to this method.
10. Connect the Jobs initialize event to the VJobList.load() method.
11. Connect the Jobs initialize event to the AS400ExplorerPane.setRoot(VNode) method, and pass the VJobList as the parameter to this method.
12. Save and run the sample application.

5.8.4 Users and Groups

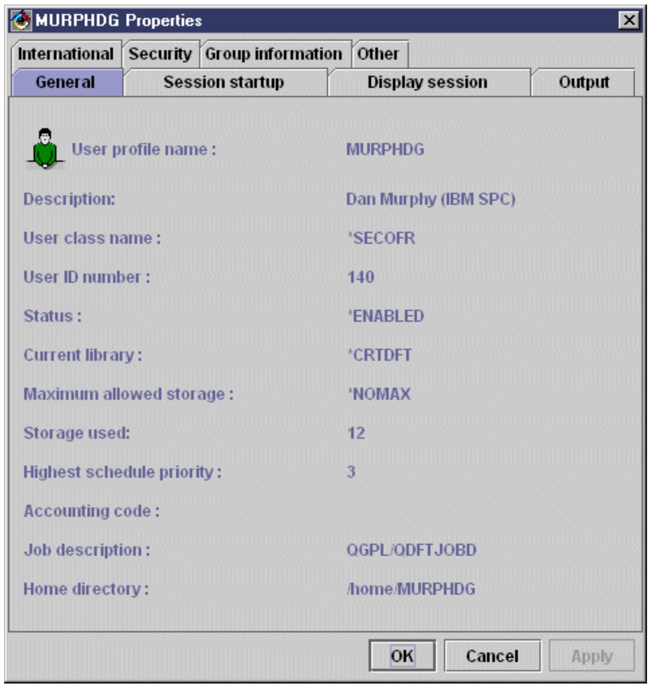
The VUserList and VUserAndGroup classes provide a simple and efficient way to manage user profiles from a Java client. Simply adding a VUserList to an AS400DetailsPane allows the user to list AS/400 user profiles.



User name	Description
MURPHDG	Dan Murphy (IBM SPC)
QAUTPROF	IBM-supplied User Profile
QBRMS	IBM-supplied User Profile
QCOLSRV	IBM-supplied User Profile
QDBSHR	Internal Data Base User Profile
QDBSHRDO	Internal Data Base User Profile
QDFTOWN	Default Owner for System Objects
QDIRSRV	OS/400 Directory Services Server User Profile
QDLFM	IBM-supplied User Profile
QDOC	Internal Document User Profile

Figure 216. A VUserList Example

To inspect user profiles in more detail, right-click the selected user profile and select the property pop-up option.



User profile name :	MURPHDG
Description:	Dan Murphy (IBM SPC)
User class name :	*SECOFR
User ID number :	140
Status :	*ENABLED
Current library :	*CRTDFT
Maximum allowed storage :	*NOMAX
Storage used:	12
Highest schedule priority :	3
Accounting code :	
Job description :	QGPL/QDFTJOBD
Home directory :	/home/MURPHDG

Figure 217. The Properties for a Selected User

An additional class, VUserAndGroup, allows a Java program to list user groups, all users, or users that are not in a group.

5.8.4.1 Building the Sample Application

The following steps create a the program used to generate the displays shown in Figure 216 on page 263 and Figure 217:

1. Start IBM VisualAge for Java 2.0.
2. Within the **New Toolbox Samples** project add a new package called UsersExample.
3. Create a new class in this package called Users. This class should extend com.sun.java.swing.JFrame.
4. Select the **Compose the class visually** radio box.
5. Within the VCE, set the JFrameContentPane to use the Border layout manager.
6. Add an AS400DetailsPane to the Center of the JFrameContectPane.
7. Add an AS400 object into the free form area.
8. Add a VUserList object to the free form area of the VCE.
9. Connect the User object initialize event to the VUserList1 object setSystem(AS400) method.
10. Pass the AS4001 object as the parameter to the setSystem() method call generated in the previous step.
11. Connect the User object initialize event to the VUserList1 object load() method.
12. Connect the User object initialize method to the setRoot(VNode) method of the AS400ListPane1 object.

13.Set the VUserList1 object as the parameter to the setRoot(VNode) method generated in the previous step.

14.Run the program and connect to an AS/400 system.

Note

As with all the examples in the section, the amount of time required to retrieve information from the AS/400 system depends on many factors. During this time, the user is not given any indication that processing is taking place. Also, little error handling is performed, the addition of an ErrorDialogAdaptor would improve the application's error handling. See Chapter 4, "AS/400 Toolbox for Java — GUI Classes" on page 181, for examples of using the ErrorDialog Adapter class.

5.8.5 IFS File Access

The IFSJavaFile class represents a file in the AS/400 integrated file system. IFSJavaFile extends the java.io.File class and allows programs to be written for the java.io.File interface and still access the AS/400 integrated file system. IFSFile should be considered as an alternate to this class.

Here are some considerations to help you decide when IFSJavaFile should be used:

- IFSJavaFile should be used when a portable interface, compatible with java.io.File, is needed. For example, you have written code that accesses the native file system. Now you want to move the design to a networked file system. More particularly, you need to move the code to the AS/400 integrated file system. When a program is being ported and needs to use the AS/400 integrated file system, IFSJavaFile is a good choice. IFSJavaFile also provides the SecurityManager features defined in java.io.File.
- If you need to take full advantage of the AS/400 integrated file system, IFSFile is more useful. IFSFile is written to handle more of the specific AS/400 integrated file system details.
- java.io.File can be used to access the AS/400 file system if you use a product like Client Access/400 to map a local drive to the AS/400 integrated file system.

IFSJavaFile is used in conjunction with IFSFileInputStream and IFSFileOutputStream. It does not support java.io.FileInputStream and java.io.FileOutputStream. Despite IFSJavaFile being based on the IFSFile class, its interface is more like java.io.File than IFSFile. Using this class enables your Java application to access remote AS400 integrated file systems in the same manner as it would access a local system file. The following code snippet accesses an AS400 called SystemX and writes an array of bytes to a file in the named myFile.txt in the /home directory of the AS/400 system IFS:

```
IFSJavaFile file = new IFSJavaFile(new AS400("SystemX"), "/home/MyFile.txt");
try {
   IFSFileOutputStream os = new IFSFileOutputStream(file.getSystem(),
         file,IFSFileOutputStream.SHARE_ALL,false);
   byte[] data = new byte[256];
   for (int i=0; i < data.length; i++) {
         data[i] = (byte) i;
```

```
            os.write(data[i]);
    }
    os.close();
}
catch (Exception e) {
     System.err.println ("Exception: " + e.getMessage());
}
```

Chapter 6. Enterprise Access Builder for Data (DAX)

The VisualAge for Java Enterprise edition includes the following Access Builder components:

- Enterprise Access Builder for Data

  This component allows access to any relational database that supports either an ODBC driver or a JDBC driver.

- Enterprise Access Builder for Java to C++

  This component allows access to C++ services by generating JavaBeans and C++ code to allow interoperability between Java and C++.

- Enterprise Access Builder for RMI

  This component is used for creating distributed Java applications. Remote Method Invocation (RMI) allows a Java object running on one virtual machine to send messages to another Java object running on a another Java virtual machine. These objects can even be on different systems.

- Enterprise Access Builder for SAP R/3 using SAP R/3 BAPI business objects
- Enterprise Access Builder for Persistence Enterprise Access Builder

  This component is used for transforming relational schemas into Enterprise JavaBeans components.

- Enterprise Access Builder for interacting with existing applications

This chapter focuses entirely on the Enterprise Access Builder for Data component of the VisualAge Java Enterprise edition.

6.1 Overview

Enterprise Access Builder for Data (referred to as Data Access Builder or DAX) is part of the VisualAge Java Enterprise edition. It allows you to generate data access classes based on existing relational database tables.

You use Data Access Builder to generate the Java source code (classes) to access data. These generated Java classes, which are JavaBeans, can be used directly in your Java programs or within the VisualAge for Java Visual Composition Editor. Some of the key features of Data Access Builder are:

- **JDBC access to data**

  Data Access Builder generates classes that use JDBC to access databases. You can use the JDBC driver that is part of the AS/400 Toolbox for Java to access the databases.

- **RAD but still object-oriented**

  Data Access Builder can generate Java source code in a matter of minutes that allows you to add, update, delete, and retrieve rows from a database. Data Access Builder generates the code in a consistent, extendable, and object-oriented fashion enabling the benefits of object-oriented programming.

- **Generated JavaBeans**

  Data Access Builder generates JavaBeans. JavaBeans are a standard Java class architecture that allows generated classes to be used in any JavaBeans compliant IDE or utility.

- **Stored procedures**

  You can use Data Access Builder to generate code that calls JDBC stored procedures. Stored procedures often provide better performance than JDBC data access.

- **Commitment control and connection**

  Services are provided for connecting to databases. In addition, commit and rollback methods are also generated for transactions.

6.2 Building an Application Using the Data Access Builder (DAX)

This section describes how to create an application using the VisualAge for Java Data Access Builder.

6.2.1 Application Requirements

The application we build is for the ABC Parts Supply Company, a fictitious parts wholesaler. The application allows ABC employees to enter orders by selecting a customer, the part being ordered, and the quantity of the part being ordered.

The application uses the three DB2/400 database tables that are described in Table 37.

Table 37. ABC Database Tables

Table/File Name	Description	Comments
Parts	Contains the parts that can be ordered	Keyed by part id (IID)
Customer	Contains the company's customers	Keyed by customer id (CID)
Orders	Contains order information	keyed by the order timestamp (ORDERTMSP) customer id (CUSTID).

The layout of the three DB2/400 database tables are described in Table 38 through Table 40 on page 269.

Table 38. Database Tables Layout (Customer)

Field/Column Name	Description	Type	Length
CID	Uniquely identifies a customer	CHAR	4
CFNAME	First Name	CHAR	20
CLNAME	Last Name	CHAR	20
CADDRESS	ADDRESS	CHAR	30
CCITY	City	CHAR	30
CSTATE	State	CHAR	2
CZIPCODE	ZipCode	CHAR	15
CBAL	Customer Balance (not used)	PACKED	8,2
CPHONE	Phone Number	CHAR	20

Table 39. Database Tables Layout (Parts)

Field/Column Name	Description	Type	Length
IID	Uniquely identifies a part	CHAR	4
INAME	Part Name	CHAR	20
ICOMMENT	Comment About the Part	CHAR	30
IPRICE	Price of Part	PACKED	6,2
ICOST	Cost of Part	PACKED	6,2
IIMAGE	Image of Part (Not Used)	CHAR	40
ISOUND	Sound File of Part (Not Used)	CHAR	40
IQTY	Quantity in Inventory	BINARY	4
ISOLD	Quantity of Part Sold	BINARY	4

Table 40. Database Tables Layout (Orders)

Field/Column Name	Description	Type	Length
ORDERTMSP	Timstamp When Order Was Created	TIMESTAMP	N/A
CUSTID	Customer Id Field	CHAR	4
PARTID	Part Id Field	CHAR	4
QUANTITY	Quantity of Part Ordered	BINARY	4

The application we create is shown in Figure 218. It allows the user to view a list of parts and customers and to create orders. The user must sign on and connect to the database using the Connection menu option before any processing can occur.

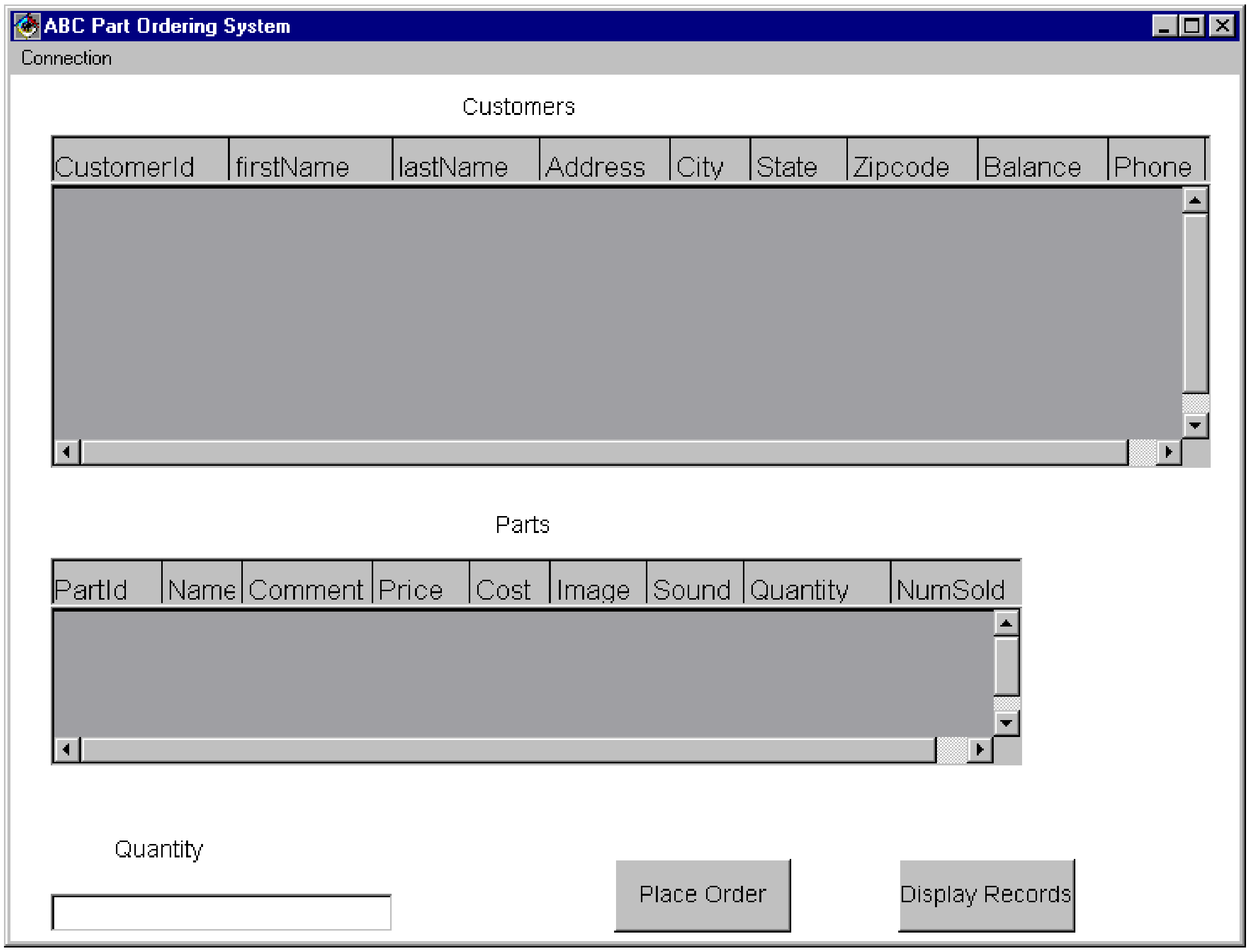


Figure 218. Parts Order Management Window

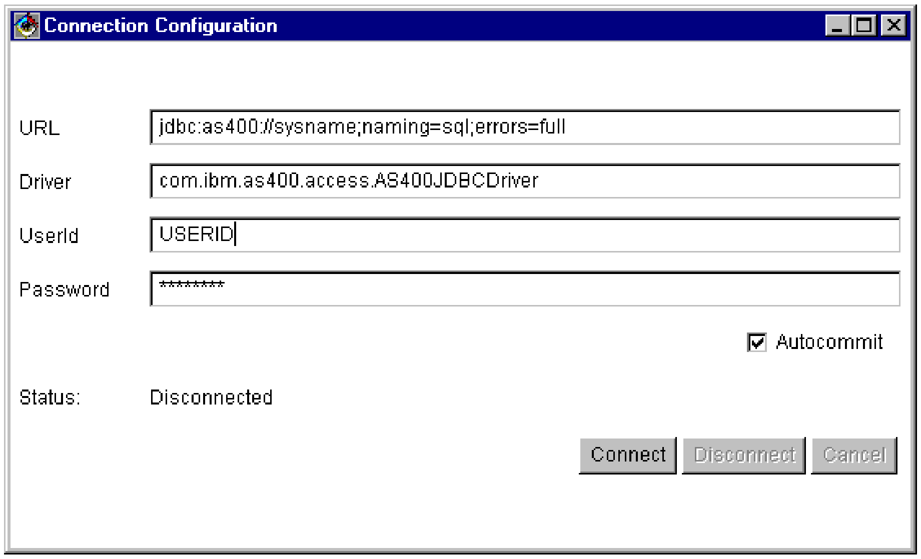
The following process occurs when the Display Records button is clicked:

1. All of the customer records are read from the database and placed in the multi-column Customer list box.
2. All of the parts records are read from the database and placed in the multi-column Parts list box.

The following process occurs when the Place Order button is clicked:

1. The parts record field IQTY is reduced by the quantity ordered. The ISOLD field is incremented by the quantity sold.
2. The parts record is updated in the database file.
3. A new order, which includes the customer ID, part ID, and quantity ordered, is inserted into the Order database.

The Parts Ordering Application "Connection Configuration" frame is shown in Figure 219 on page 271. It allows the user to specify a URL, JDBC driver, user ID, password, and commitment control option. The user uses this window to connect to the AS/400 system.


Connection Configuration

URL jdbc:as400://sysname;naming=sql;errors=full

Driver com.ibm.as400.access.AS400JDBCDriver

UserId USERID

Password ********

[✓] Autocommit

Status: Disconnected

Figure 219. Parts Configuration Window

6.3 Generating the Application Using DAX

In this section, we build the complete ABC Part Ordering System application using DAX.

6.3.1 Understanding the Software Design

A key feature of Java is its support of object-oriented programming (OOP). Please refer to Chapter 1, “Object-Oriented Technology Overview” on page 1, for a more detailed discussion of object-oriented programming. Here, we use and discuss elementary elements of OOP. Figure 220 on page 272 illustrates the use of Unified Methodology Language (UML) to describe our object model. UML is basically a diagramming language used to describe object data properties, actions, and relationships with other objects. For more information on UML, refer to the site on the Web at: http://www.rational.com

An object model is produced with UML through object-oriented analysis and design (OOA OOD).

The goal of OOP is to increase programmer productivity and the quality of the software produced. To achieve this goal, we design software that:

- Models the real world

  OOP allows us to create objects and classes that are the same as their real-world counterparts. This makes software more simple and easier to understand.

- Promotes re-usability

  Objects can be created in an abstract way, and sub-classed or extended using inheritance. This allows object properties and operations to be reused.

The UML object model shown in Figure 220 illustrates the software design we use for constructing our sample application. UML uses a rectangle with three compartments to describe a class. The top compartment simply contains the class name. The middle compartment contains the attributes or properties of the class. The bottom compartment contains the operations or methods that this class can perform.

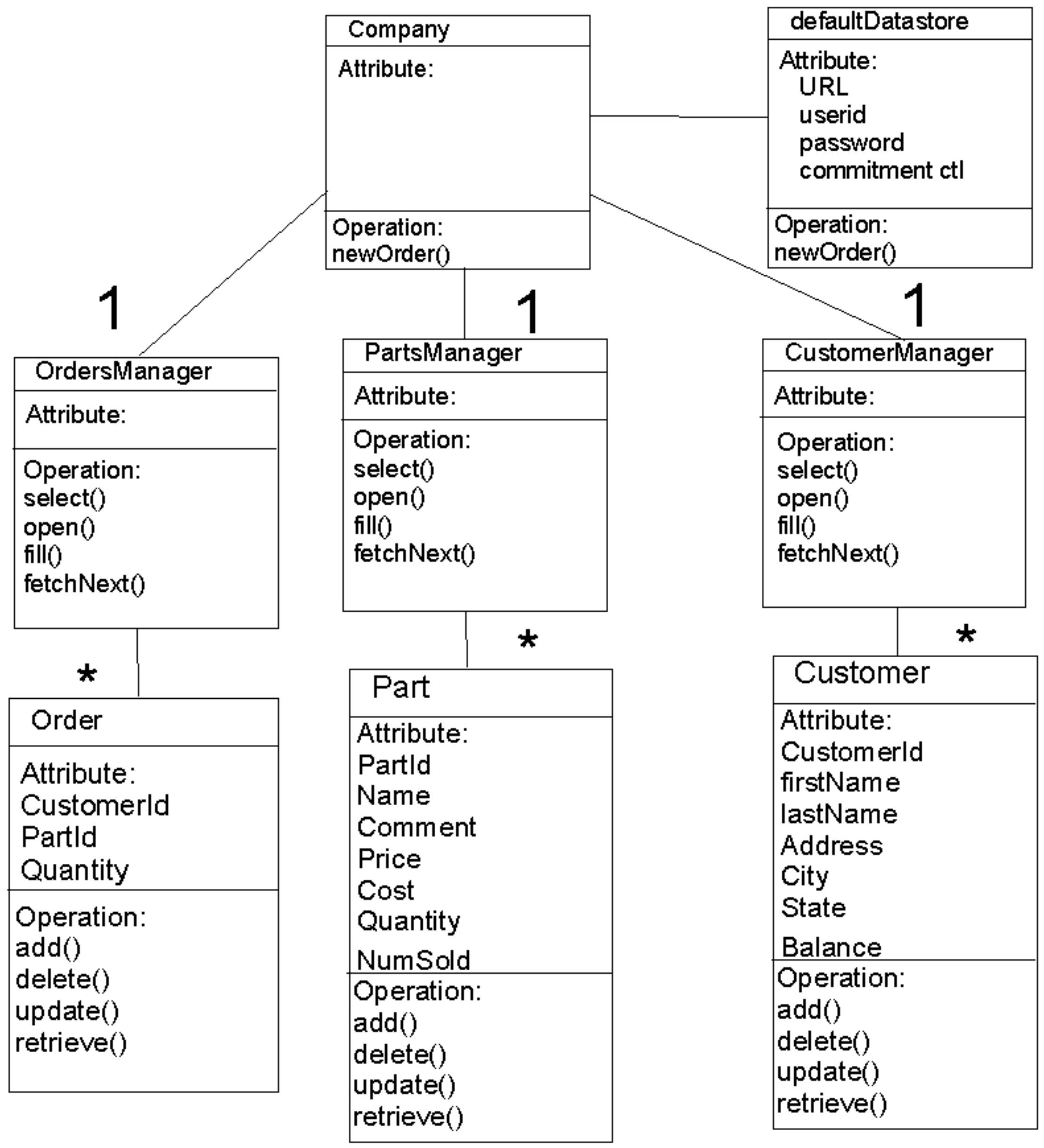


Figure 220. UML Object Model

Table 41 describes all of the classes previously illustrated in the UML diagram.

Table 41. Application Classes

Class	Description
Company	Company is the main integrating class. It contains all of the manager classes and is responsible for creating new orders.
OrderManager	OrderManager is responsible for selecting and managing a collection of order objects. The OrderManager class is contained within the Company class.
CustomerManager	CustomerManager is responsible for selecting and managing a collection of customer objects. The CustomerManager class is contained within the Company class.
PartsManager	PartsManager is responsible for selecting and managing a collection of parts objects. The PartsManager class is contained within the Company class.
Order	An order represents one single order. The order contains data properties such as customer id, part number, quantity, and the timestamp of the order.
Parts	A parts object represents one single part that can be ordered from the company. The parts object contains data properties such as Part id, name, price, and cost.
Customer	A customer object represents one single customer. The customer object contains data properties such as Customer id, firstName, lastName, address, and so on.
defaultDatastore	This object represents the connection to a server. It contains data properties such as URL, user id, password, and commitment control.

6.3.2 Building the Application

Without using the Data Access Builder, we must manually create all of the classes, as well as code all of the logic to select, update, and insert records into the database. Using the Data Access Builder, much of the code is generated for us. First, we start by creating a VisualAge project and package to hold the classes that we are about to create. We can start the Data Access Builder by choosing the Selected menu option and then selecting Tools—>Data AccessBuilder—>Create Data Access Builder Beans from the VisualAge for Java Workbench menu. This brings up a Data Access Builder session window.

Data Access Builder uses ODBC to access the remote databases. You must define an ODBC data source for it to use. Once the JavaBeans are generated, JDBC is used for database access and ODBC is no longer used.

Selecting Map Schema from the file menu starts the database-to-Java object mapping process. Selecting the ODBC Data Source that represents the target system identifies the location of the data source. From this point, clicking the Get Tables button retrieves the available tables and views from the target system. Selecting a particular file such as the CUSTOMER file results in a window similar to the one shown in Figure 221.

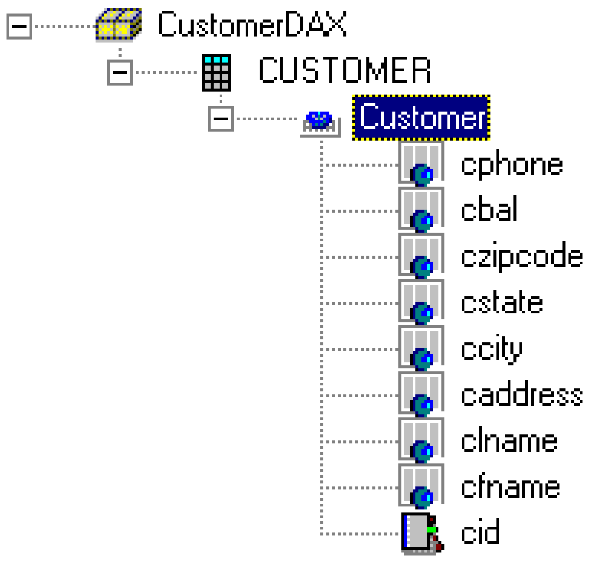


Figure 221. DAX Generation Window

The Data Access Builder has, at this point, accessed the database that was specified and retrieved all the fields (or columns) available in the database file. DAX creates a Java class named Customer and adds variables for each field in the file, as well as the Java methods to retrieve and set the variables value. For example, the database contains a field named czipcode. DAX generates an instance variable named czipcode and a method named getczipcode to get the value. It also generates a method called setczipcode that is capable of setting the value of this instance variable. You can change the names of the instance variables to something more descriptive such as zipCode instead of czipcode. This can be done from the attribute settings of the Customer class as shown in Table 42 on page 275. The data identifier, which is the field or fields that identify a record, can also be specified in this window.

We make the changes shown in Table 42 on page 275 by selecting Attributes from the Customer pop-up menu.

Table 42. Customer Table

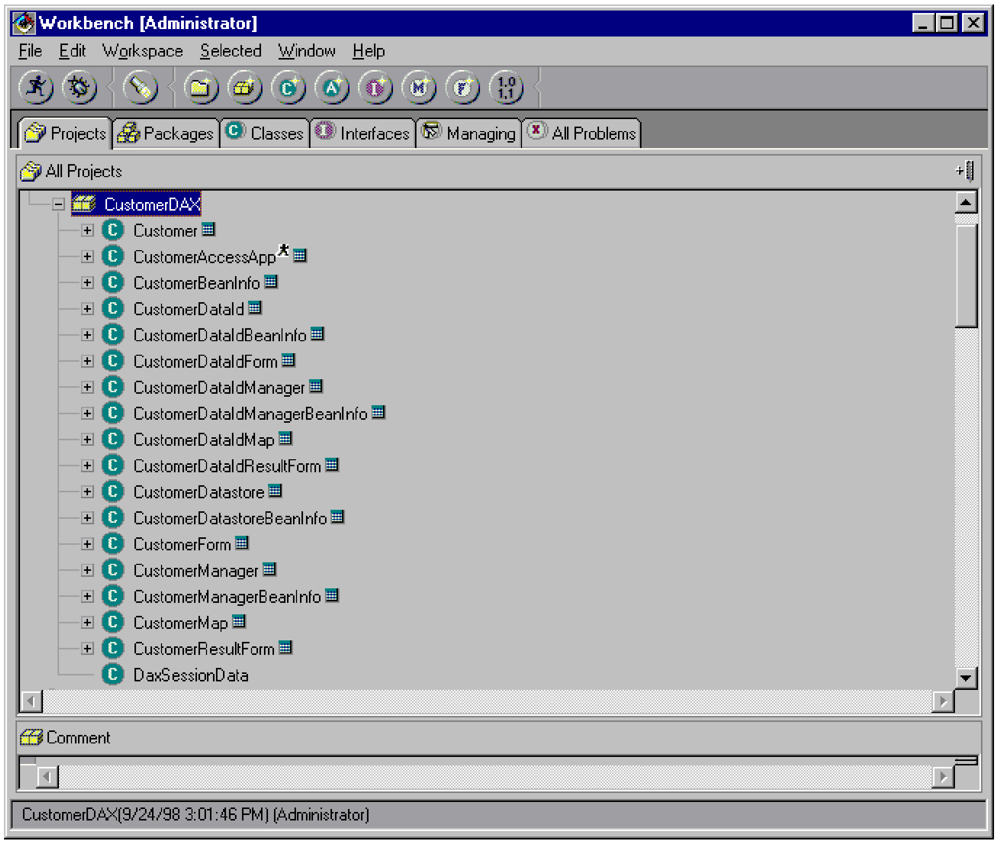
Database Field Name	JavaBean Attribute Name	Data Identifier	Comments
cid	CustomerId	**YES**	ID Number of Customer
cfname	firstName	No	Customer's First Name
clname	lastName	No	Customer's Last Name
caddress	Address	No	Customer's Address
ccity	City	No	Customer's City
cstate	State	No	Customer's State
czipcode	Zipcode	No	Customer's Zipcode
cbal	Balance	No	Customer's Balance
cphone	Phone	No	Customer's Telephone Number

Note: We change CustomerId to be the data identifier. The data identifier is taken from the primary key of the table. If the table does not have a primary key specified, you may have to manually specify the data identifier as we do here. Having a data identifier allows DAX to generate, delete, update, and retrieve methods.

In addition to the generated Customer class, several additional classes are generated including a CustomerManager. The CustomerManager class is capable of retrieving and instantiating a collection of Customer objects.

The DAX generation process starts when you specify Save and Generate from the file menu pull-down. This takes a few minutes while DAX actually creates the classes with the appropriate methods and variables.

Upon completion of the generation process, the classes shown in Figure 222 on page 276 are generated by DAX and placed in the Java package.



Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
Comment

CustomerDAX
Customer
CustomerAccessApp
CustomerBeanInfo
CustomerDataId
CustomerDataIdBeanInfo
CustomerDataIdForm
CustomerDataIdManager
CustomerDataIdManagerBeanInfo
CustomerDataIdMap
CustomerDataIdResultForm
CustomerDatastore
CustomerDatastoreBeanInfo
CustomerForm
CustomerManager
CustomerManagerBeanInfo
CustomerMap
CustomerResultForm
DaxSessionData

CustomerDAX(9/24/98 3:01:46 PM) (Administrator)

Figure 222. Dax Generated Customer Window

Table 43 on page 277 describes each generated class (bean).

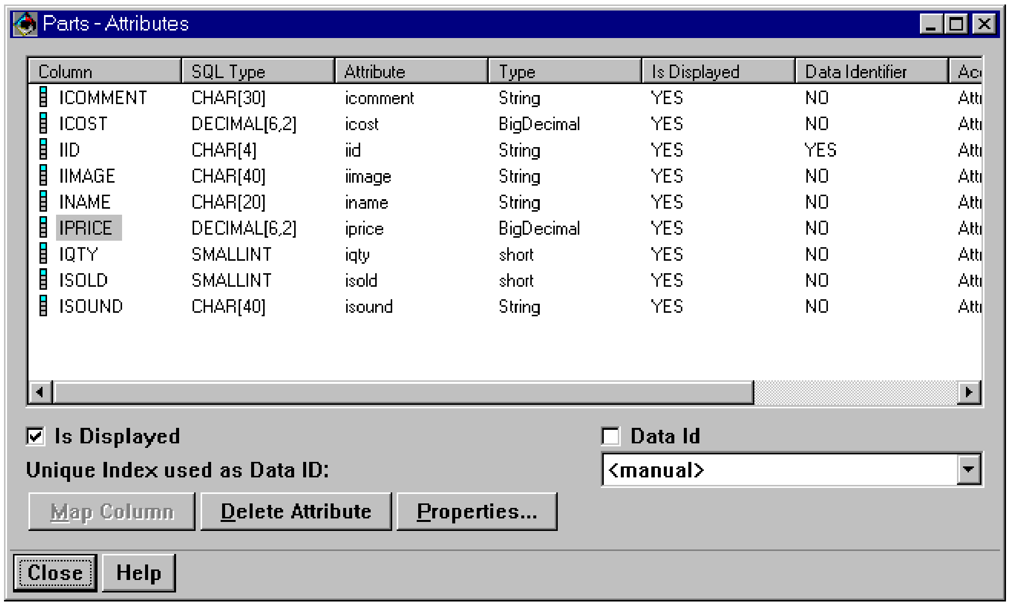
Table 43. Generated Classes

DAX Generated Class	**Description**
Customer	This represents a single instance of a customer and maps to one record from the customer database file.
CustomerAccessApp	This is a sample application that can be used to test the other classes. It has notebook pages for viewing all records, updating, deleting, and inserting records.
CustomerBeanInfo	All the classes generated are JavaBeans. This is the JavaBeans information file associated with the Customer class.
CustomerDataId	This object is the same as the Customer object, but only contains the key fields or data identifier variables for the customer object. Data identifiers can be specified in the attributes table within DAX. For the Customer object, the data identifier is the cid (customer id) field.
CustomerDataIdBeanInfo	This is the JavaBean information for the CustomerDataId class.
CustomerDataIdForm	This is a GUI panel for displaying the CustomerDataId fields.
CustomerDataIdManager	This class is responsible for selecting, updating, deleting, and creating new CustomerDataId objects. This class "manages" CustomerDataId objects.
CustomerDataIdManagerBeanInfo	This is the JavaBean information for the CustomerDataIdManager class.
CustomerDataIdMap	This is an internal DAX object that is used to automatically retrieve fields from the SQL/JDBC cursor and puts them into a CustomerDataId object.
CustomerDataIdResultForm	This is a GUI container that contains a multi-column list box for displaying a list of keys. This can be used in a case where you wanted to allow a user to select a customer based on an id. When the id is selected, a select operation can retrieve the full customer object for viewing of detailed customer information.
CustomerDatastore	This class is responsible for handling the connection to a data source such as DB2/400. It contains such properties as URL, user id, password, and connection status.
CustomerDatastoreBeanInfo	This is the JavaBean information for the CustomerDatastore class.
CustomerForm	This is a GUI panel for displaying the Customer fields.

Note: The xxxDataIdxxx classes are only generated if a data identifier field is specified during the generation process. These generated classes are the reusable elements that we use to build the application.

We follow the same process to generate classes for the Order file and the Parts file. This results in classes such as Order, OrderManager, OrderResultForm, Parts, PartsManager, and PartsResultForm.

We set the attributes for the Parts class as shown in Figure 223.



Column	SQL Type	Attribute	Type	Is Displayed	Data Identifier	Ac
ICOMMENT	CHAR[30]	icomment	String	YES	NO	Att
ICOST	DECIMAL[6,2]	icost	BigDecimal	YES	NO	Att
IID	CHAR[4]	iid	String	YES	YES	Att
IIMAGE	CHAR[40]	iimage	String	YES	NO	Att
INAME	CHAR[20]	iname	String	YES	NO	Att
IPRICE	DECIMAL[6,2]	iprice	BigDecimal	YES	NO	Att
IQTY	SMALLINT	iqty	short	YES	NO	Att
ISOLD	SMALLINT	isold	short	YES	NO	Att
ISOUND	CHAR[40]	isound	String	YES	NO	Att

[✓] Is Displayed

Unique Index used as Data ID:

[ ] Data Id

<manual>

Figure 223. The Parts Attributes Window

We set the attributes for the Parts table and the Orders table as shown in Table 44 and Table 45 on page 279.

Table 44. Parts Table

Database Field Name	Java Bean Attribute Name	Data Identifier	Comments
iid	PartId	YES	Part ID Number
iname	Name	No	Name of Part
icomment	Comment	No	A description of the part
iprice	Price	No	Price of part
icost	Cost	No	Cost of part
iqty	Quantity	No	Quantity in Stock
isold	NumSold	No	Number of parts Sold

Table 45. Orders Table

Database Field Name	Java Bean Attribute Name	Data Identifier	Comments
ordertmsp	Ordertmsp	YES	Timestamp of Order
custid	CustomerId	YES	Customer ordering part
partid	PartId	No	ID of part ordered
quantity	Quantity	No	Number of parts ordered

6.4 Building the Company Class

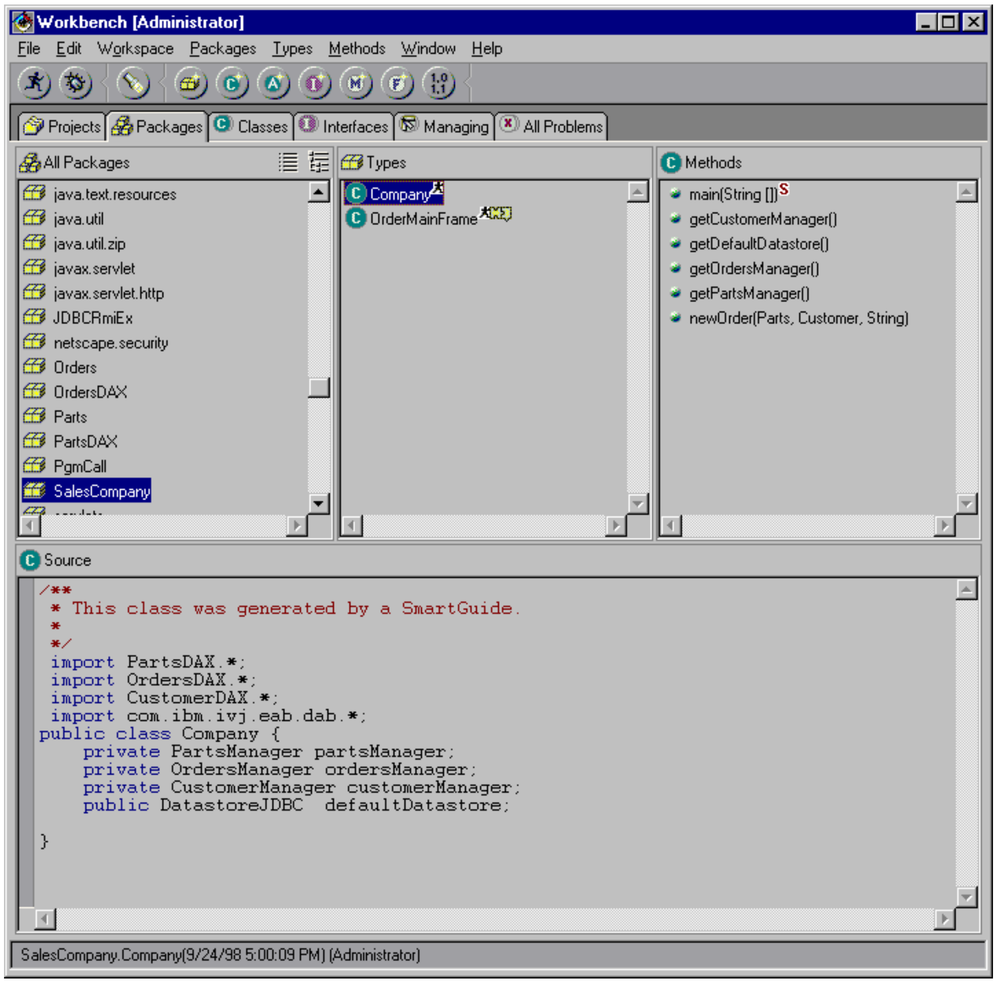
The Company class handles the processing for a new order. It contains the following objects:

- PartsManager object
- OrderManager object
- CustomerManager object

The Company class integrates the xxxManager classes and has the ability to create orders. The xxxManager classes are generated for us by DAX. We must create the Company class because it is part of our object model that DAX knows nothing about. To create the Company class, we simply create a class within one of the Java packages.

Figure 224 on page 280 shows the Company class created within a package called SalesCompany. It shows partsManager, orderManager, and customerManager instance variables created as private variables. Along with these variables are methods to get the values of these variables. For example, the customerManager variable has a getCustomerManager method that returns the value of the variable. The returned value is an instance of a CustomerManager object. These variables do not have associated set methods because there is no need in this application to set these variables.

The import statements allow you to use classes and objects that exist in a different package. The defaultDatastore variable is used to hold our connection object. This connection object, which is generated by DAX, contains the URL, connection status, and methods to connect and disconnect from the database. The defaultDatastore variable has a getter method to return the datastore object.



```
/**
 * This class was generated by a SmartGuide.
 *
 */
 import PartsDAX.*;
 import OrdersDAX.*;
 import CustomerDAX.*;
 import com.ibm.ivj.eab.dab.*;
public class Company {
     private PartsManager partsManager;
     private OrdersManager ordersManager;
     private CustomerManager customerManager;
     public DatastoreJDBC  defaultDatastore;

}
```

Figure 224. DAXProject Window

The method shown in Figure 225 is used to acquire the customerManager object.

```
public CustomerManager getCustomerManager() {
          if (customerManager == null) {
          customerManager = (new
         CustomerManager(getDefaultDatastore()));
          }
          return customerManager;
         }
```

Figure 225. The getCustomerManager Method

This code uses a technique called *lazy initialization*. Lazy initialization tests and sets the value of a variable when it is accessed. In this case, if the customerManager variable is null, it is set to a new instance of the CustomerManager class. The getDefaultDatastore method is used to assign the defaultDatastore as the datastore of the CustomerManager instance. The methods for getOrderManager and getPartsManager are the same except they return instances of OrderManager and PartsManager respectively.

The getDefaultDatastore() method, shown Figure 226, can return any of the xxxxxxdatastore objects. This is because DAX generated three datastore objects called PartsDatastore, OrderDatastore, and CustomerDatastore. Since they all reference the same URL and datastore information, they are all the same. We simply use PartsDatastore.

```
public DatastoreJDBC getDefaultDatastore() {
            if (defaultDatastore == null) {
            defaultDatastore = (new
           PartsDatastore());
            }
            return defaultDatastore;
           }
```

Figure 226. The getDefaultDatastore Method

The *newOrder (Parts aPart, Customer aCustomer, String aQuantity)* method is our most important method. It is called when the user clicks the Place Order button after selecting a customer, a part, and specifying a quantity.

```
public void newOrder(Parts aPart, Customer aCustomer, String aQuantity) {

/* convert the input string aQuantity to a short value.  It comes in a string because
   it comes from the user interface */
short quantitySold = (new Short(aQuantity)).shortValue();

/** Create a new order and populate the order data */
Orders newOrder = new Orders();
newOrder.setQuantity(quantitySold);
newOrder.setCustomerId(aCustomer.getCustomerId());
newOrder.setPartid(aPart.getPartId());
newOrder.setOrdermsp(new java.sql.Timestamp(System.currentTimeMillis()));
/** Reset the part inventory quantity.  Need to do a lot of conversion since
   coming from a short  */

int newQuantityInt = (aPart.getQuantity() - quantitySold); //short - short gives a int
String newQuantityString = (new String()).valueOf(newQuantityInt);
short newQuantity = ((new Short(newQuantityString)).shortValue());
aPart.setQuantity(newQuantity);
int newNumSoldInt = (aPart.getNumSold() + quantitySold); //short - short gives a int
String newNumSoldString = (new String()).valueOf(newNumSoldInt);
short newNumSold = ((new Short(newNumSoldString)).shortValue());
aPart.setNumSold(newNumSold);
try {
aPart.update();
} catch (Exception e) {
System.out.println("Error updating part " + e);
}
try {
newOrder.add();
} catch (Exception problem) {
System.out.println("error adding order " + problem);
}
 }
```

Figure 227. The newOrder Method

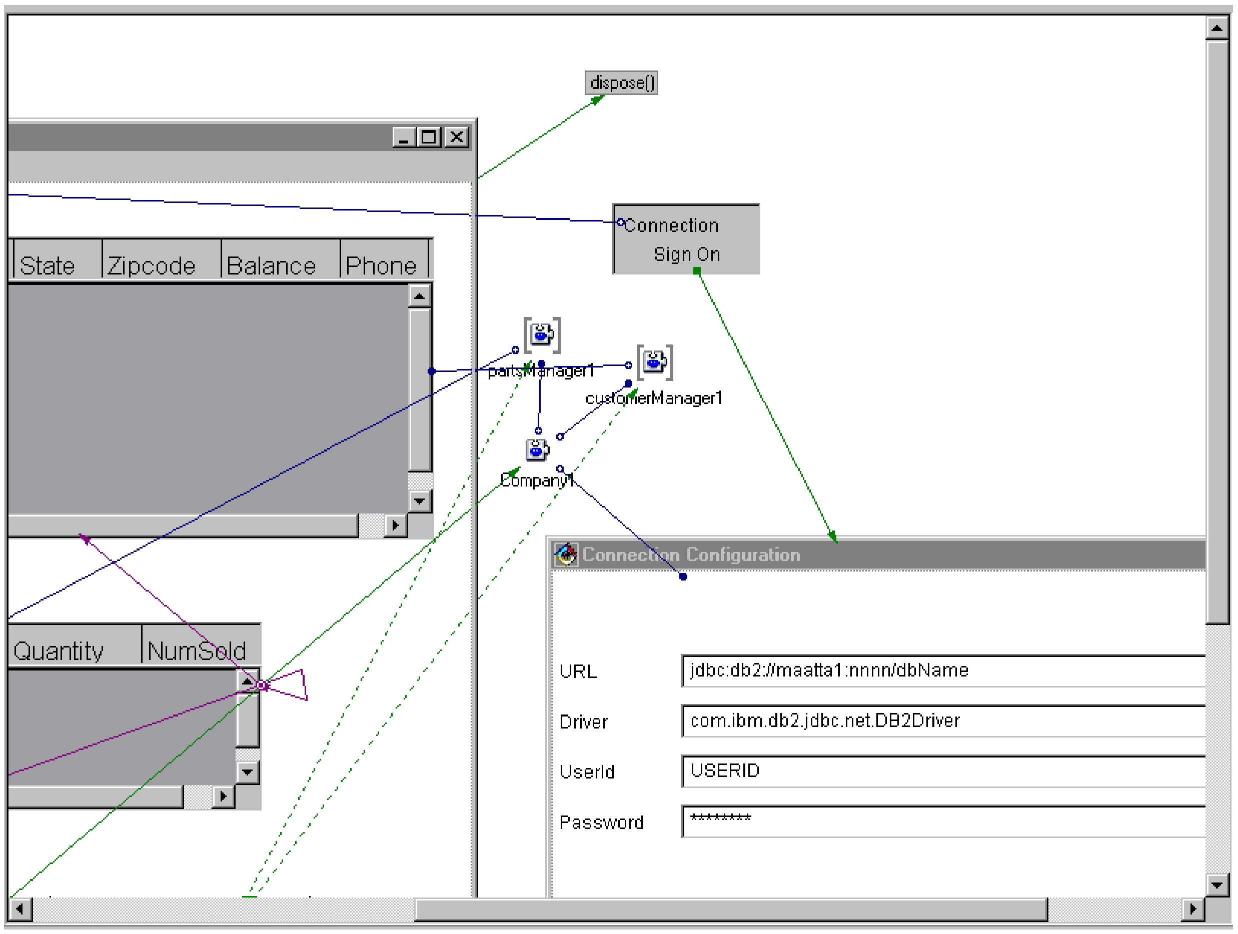
This method accepts the three objects in the parameter list. The method performs these actions:

1. Creates a new order object and sets the appropriate data in it.
2. Updates the part object by reducing the inventory and incrementing the number sold.
3. Adds the order record to the database.

6.5 Building a Custom GUI Using DAX Objects

The last task is to create a user interface for the ABC parts ordering system. Basically, we assemble and connect the classes that DAX created for us, along with our custom Company class.

As shown in Figure 228, we first create a new class named OrderMainFrame, which extends from java.awt.Frame, that we use to compose our GUI windows.



URL jdbc:db2://maatta1:nnnn/dbName
Driver com.ibm.db2.jdbc.net.DB2Driver
UserId USERID
Password ********

Figure 228. OrderMainApp in the Composition Editor Window

We then use the Visual Composition Editor to:

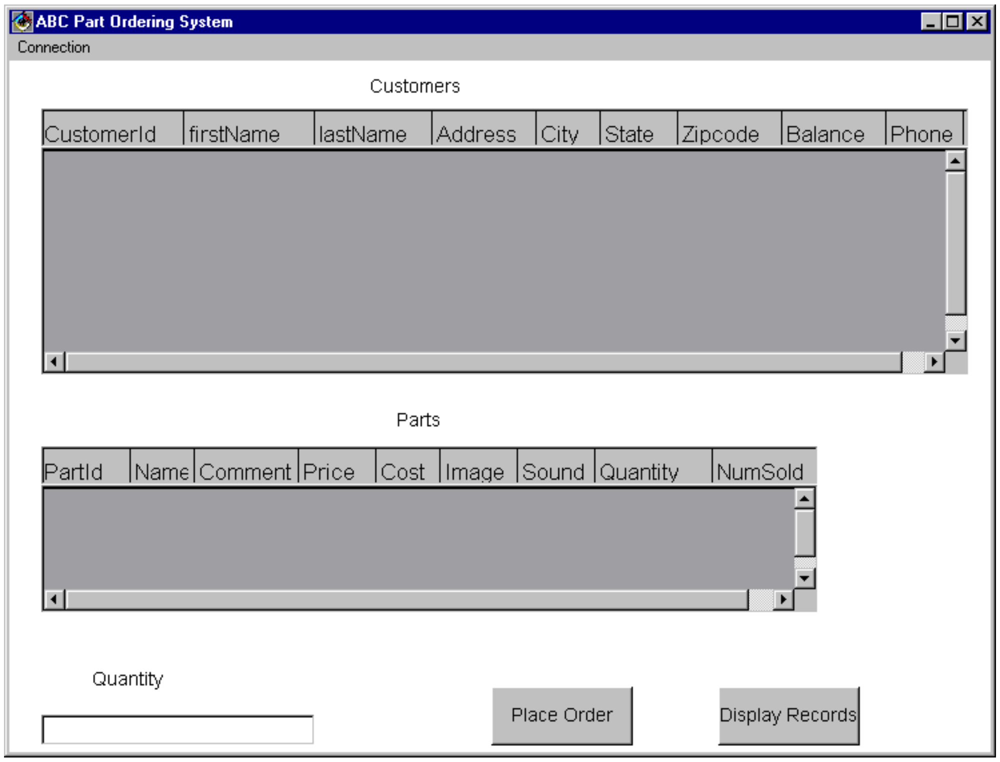
- Add the visual parts
- Add the non-visual parts
- Add the connections

The parts or classes listed in Table 46 are used.

Table 46. Application Parts

Classes to Be Added	**Comments**
Company	This is main object and is composed of the datastore, order, customer, and parts managers objects.
Display Records and Order Push Buttons	Display Records display both customers and parts records.
PartsResultForm	This is the table that displays the parts. This was generated for us by DAX.
CustomerResultsForm	This is the table that displays the customers. This was generated for us by DAX.
Quantity label and entry field.	Allows entry of a order quantity.
"Connection" menu	When clicked brings up the "OrderConfiguration" window.
"Sign On" menuitem	Added under the Connection menu.
"Order Configuration " Frame	Frame that allows entry of configuration info as shown in the following figure.
Connection Panel (IConnectPanel)	This is a VisualAge generated reusable connection panel bean as shown in the following figure. Add this to the preceding Frame.

After assembling the previously built classes, the OrderMainFrame appears similar to the example in Figure 229 on page 284.

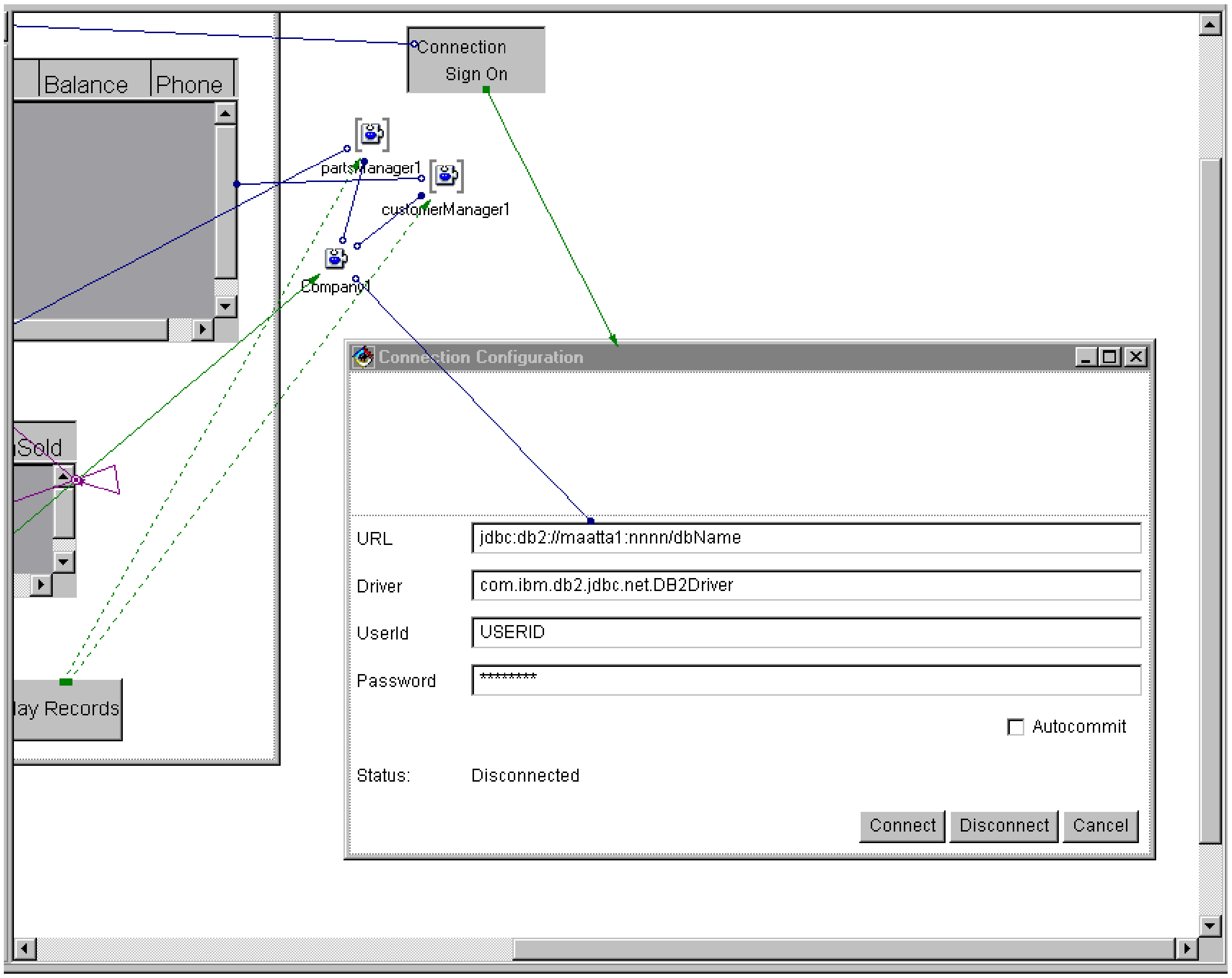


CustomerId	firstName	lastName	Address	City	State	Zipcode	Balance	Phone

PartId	Name	Comment	Price	Cost	Image	Sound	Quantity	NumSold

Figure 229. OrderMainFrame Window

Figure 230 on page 285 shows how the Order Configuration window appears in the VisualAge for Java Visual Composition Editor.



URL jdbc:db2://maatta1:nnnn/dbName
Driver com.ibm.db2.jdbc.net.DB2Driver
UserId USERID
Password ********
Autocommit
Status: Disconnected

Figure 230. Order Configuration in Visual Composition Editor

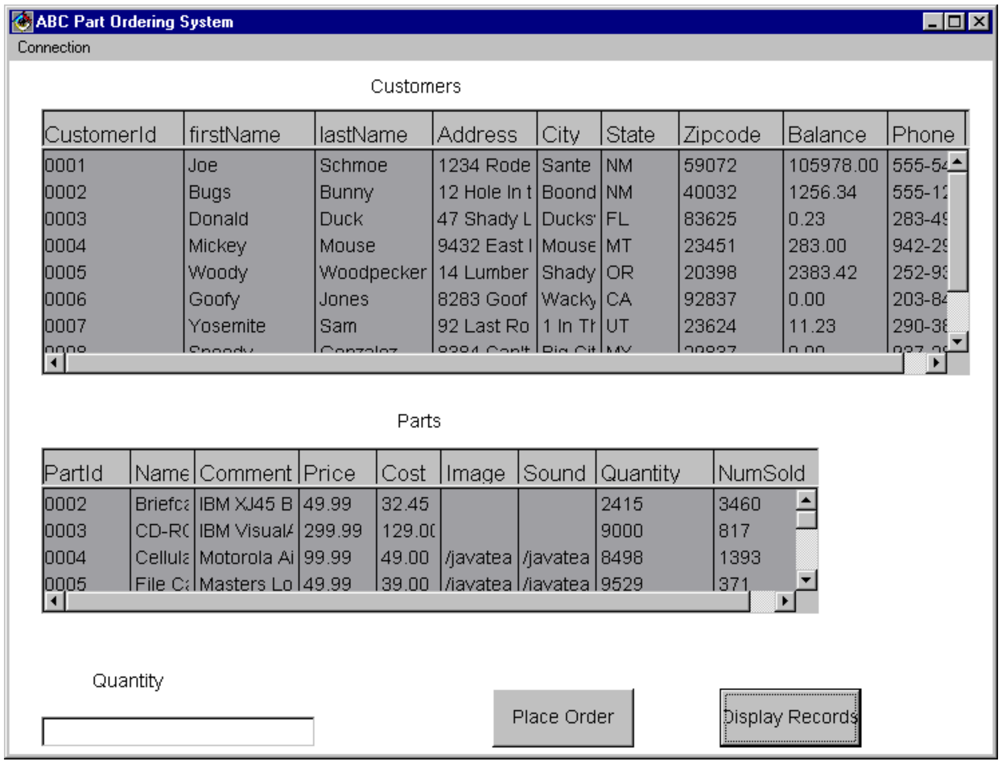
Since the Company class contains the PartsManager and CustomerManager attributes, we use the Tear off Property pop-up menu item from the Company object. This allows connections to be made to the Parts and Customer Manager objects that are contained within the Company object. The connections listed in Table 47 on page 286 complete the application.

Table 47. Application Connections

From Object	From Feature	Target Object	Target Feature
Company	defaultDatastore	IConnectPanel	dataStore
Sign On Menu Item	actinoPerformed	Order Configuration Frame	show()
Display Records Push Button	actionPerformed	PartsManager	select() method. Ignore the dotted line. In this case, the select method optionally accepts a parameter but does not require it.
Display Records Push Button	actionPerformed	CustomerManager	select() method. Ignore the dotted line. In this case, the select method optionally accepts a parameter but does not require it.
PartsManager	items	PartsResultForm	elements()
CustomerManager	items	CustomerResultFrom	elements()
Order pushbutton	actionPerformed	Company	newOrder(), use the quantity, SelectedObjects from the PartsResultForm and CustomerResultForm as the parameters.

6.6 The Completed Application

Figure 231 on page 287 shows the completed application. Please refer to the VisualAge for Java online documentation for further information about DAX.



CustomerId	firstName	lastName	Address	City	State	Zipcode	Balance	Phone
0001	Joe	Schmoe	1234 Rode	Sante	NM	59072	105978.00	555-54
0002	Bugs	Bunny	12 Hole In t	Boond	NM	40032	1256.34	555-12
0003	Donald	Duck	47 Shady L	Ducks'	FL	83625	0.23	283-49
0004	Mickey	Mouse	9432 East I	Mouse	MT	23451	283.00	942-29
0005	Woody	Woodpecker	14 Lumber	Shady	OR	20398	2383.42	252-93
0006	Goofy	Jones	8283 Goof	Wacky	CA	92837	0.00	203-84
0007	Yosemite	Sam	92 Last Ro	1 In Th	UT	23624	11.23	290-38

PartId	Name	Comment	Price	Cost	Image	Sound	Quantity	NumSold
0002	Briefca	IBM XJ45 B	49.99	32.45			2415	3460
0003	CD-R(	IBM VisualA	299.99	129.0(			9000	817
0004	Cellula	Motorola Ai	99.99	49.00	/javatea	/javatea	8498	1393
0005	File Ca	Masters Lo	49.99	39.00	/javatea	/javatea	9529	371

Figure 231. Completed Application

6.7 Summary

In summary, the benefit of using DAX over custom coding data access classes is the significant time savings. DAX can generate, in minutes, what can take several days to create with custom coding. In addition, DAX generated classes can be extended and customized by the programmer. Many advanced capabilities, such as asynchronous processing through threads, are also generated for your use. Consider DAX for any serious programming efforts. The DAX support is only available with the Enterprise Edition of VisualAge for Java.

Chapter 7. Remote Method Invocation

This chapter discusses building AS/400 client/server applications that use Remote Method Invocation (RMI) to communicate between a client program and a server program.

7.1 What RMI Is

Distributed systems require computations running in different address spaces, potentially on different hosts, to communicate. For a basic communication mechanism, the Java language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in application-level protocols to encode and decode messages for exchange. The design of such protocols is cumbersome and can be error prone.

An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure. In fact, the arguments of the call are packaged and shipped to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR. However, RPC does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. To match the semantics of object invocation, distributed object systems require remote method invocation or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

The Java remote method invocation system described in this specification has been specifically designed to operate in the Java environment. While other RMI systems can be adapted to handle Java objects, these systems fall short of seamless integration with the Java system due to their interoperability requirement with other languages. For example, CORBA presumes a heterogeneous, multi-language environment, and therefore, must have a language-neutral object model. In contrast, the Java language RMI system assumes the homogeneous environment of the Java Virtual Machine. Therefore, the system can take advantage of the Java object model whenever possible.

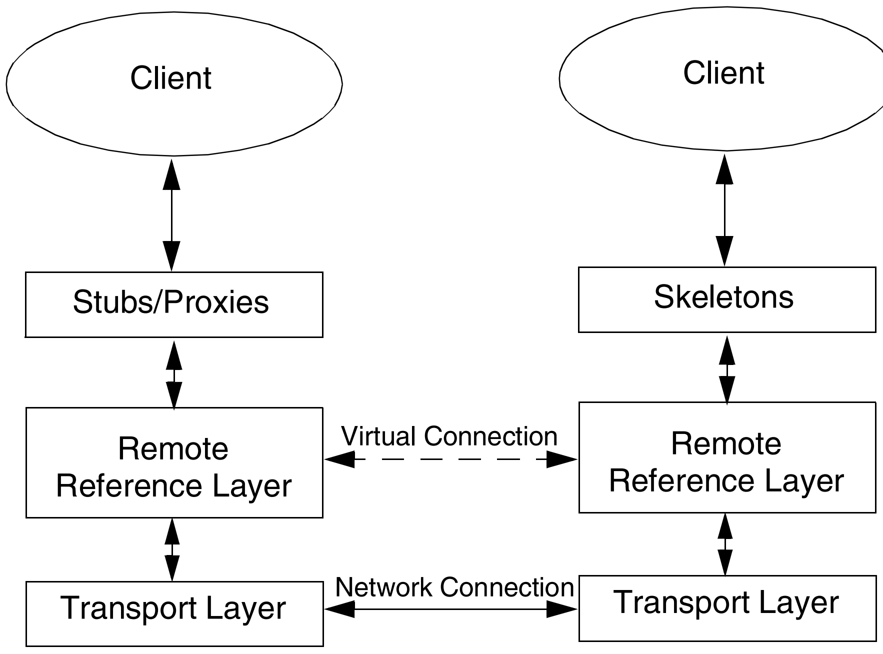


Figure 232. RMI Architecture

As shown in Figure 232, a remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport. Then, it travels up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object through the remote reference layer. Stubs are generated using the rmic compiler.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics, whether the server is a single object or is a replicated object requiring communications with its replicas.

Also handled by the remote reference layer are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring to objects that are implemented in servers that are always running on some machine, and servers that are run only when some method invocation is made on them (activation). These differences are not seen at the layers above the remote reference layer.

The transport layer is responsible for connection setup, and connection management. Plus, it keeps track of and dispatching to remote objects (the targets of remote calls) residing in the transport address space.

To dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to occur before handing off the request to the server-side skeleton. The skeleton for a remote object makes a call up to the remote object implementation that carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer, and transport on the server side. Then, it travels up through the transport, remote reference layer, and stub on the client side.

7.2 Building an RMI Application

Building an RMI application is a five step process that follows this sequence:

1. Define the interfaces to your remote server objects.

   A Java Interface is like an abstract class. It allows us to define methods without actually implementing them. We can implement the interface in a class.

2. Implement the remote server objects.

   The remote class can implement any number of remote interfaces. The class can extend another remote implementation class. The class can define methods that do not appear in the remote interface. Those methods can only be used locally and are not available remotely.

3. Run the `rmic` command on remote implementation classes.

   The rmic command creates stubs (proxies) and skeletons. It is available as part of JDK1.1. It is also available a part of many IDEs including VisualAge for Java.

4. Implement the client.

   The client invokes the remote interfaces defined by the server.

5. Make the server code network accessible.

   The server code is made network accessible by starting the RMI registry and starting and registering the server objects.

7.3 Building a Simple AS/400 Application Using RMI

This section shows you how to build a simple AS/400 client/server application using RMI. This example helps you understand the basic requirements of RMI. Later, in this chapter, we build a more complex example. This first example allows a client program to invoke a remote method, which increments a parameter passed in and returns the result. We build this application following the previously discussed five-step process.

7.3.1 Defining Interfaces

A Java interface, as shown in Figure 233 on page 292, defines a set of methods, but does not implement them. The class that implements the interface agrees to implement all methods defined in the interface. An interface exposes the programming interface of an object without revealing its class.

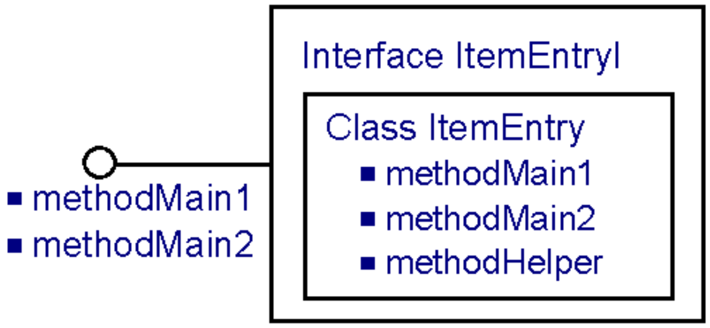


Figure 233. Java Interfaces

To use RMI, the interface must:

- Be a subclass of the Remote class
- Describe each public method
- Throw a RemoteException for each public method

Figure 234 shows an interface definition.

```
package TestRMI;

import java.rmi.*;//for Remote, RemoteException

public interface AddOneServerInterface
extends java.rmi.Remote {

public int addOne(int iNum) throws RemoteException;
}
```

Figure 234. Defining the Interface

7.3.2 Implementing the Remote Server Objects

The remote server objects follow these rules:

- The Class must extend UnicastRemoteObject and implement the interface.

  The UnicastRemoteObject class defines the remote object as a unicast object, which means that only a single instance of the object can exist on a single server. This is distinguished from a MultiCastRemoteObject, which can replicate across multiple servers. The class must implement an interface that describes the public methods.

- The constructor must throw an Exception.
- Worker methods must throw a RemoteException.
- The remote object must make its services available by:
  - Registering with an RMI security manager
  - Binding an instance of the server object to the host
  - The TCP/IP port number used must match the number used on the rmiregistry command

Figure 235 on page 293 shows the host code.

```
package TestRMI;

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class AddOneServer extends UnicastRemoteObject
                         implements AddOneServerInterface{

public AddOneServer() throws Exception {}

public int addOne(int iNum) throws RemoteException {
try {
System.out.println("addOne - someone calling us...");
System.out.println("iNum = " + iNum);
return (iNum + 1);  // Complex business logic here!
} catch (Exception e) {e.printStackTrace(); return 0;}
}
```

Figure 235. Host RMI Code

The host RMI code extends UnicastRemoteObject and implements the public interface. The public methods that it implements (addOne) throw a RemoteException.

Before the client can use the remote methods, the remote server must register with the RMI security manager. We use the setSecurityManager method to do this. The host program must bind itself as a service with an appropriate name. We use the Naming.rebind method to do this. It requires a URL as an input parameter. The format of the URL is:

```
//  +  the name of the host system + the the TCP port number + the name of the
host program ("//sysname:port/AddOne")
```

We show the host program registering with the RMI security manager in Figure 236 on page 294.

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

try {
AddOneServer myAddOneServer = new AddOneServer();
System.out.println("Main: Attempting to registerAddOneServer");
   System.out.println("Main: Before security mgr");
System.setSecurityManager(new RMISecurityManager());
System.out.println("Main: After security mgr");
System.out.println("Main: After new AddOneServer");
Naming.rebind("//sysname:6666/AddOne", myAddOneServer);
System.out.println("Main: after rebind");
System.out.println("Main: Successfully registered with the security
manager");
} catch(Exception e) {e.printStackTrace();}

return;
}
```

Figure 236. Registering with the RMI Security Manager

7.3.3 Running rmic on Remote Implementation Classes

The rmic command automatically creates stub and skeleton code from the interface and implementation class definitions.

To make the remote class ready to use, you must:

- Compile server classes using **javac** or an IDE.
- Run `rmic` or an equivalent function from an IDE.
  - In the JDK, the command is: `rmic AddOneServer`
  - From the VisualAge for Java IDE, select **Tools—>Remote Method Invocation—>Generate Proxies**

The output from the rmic command is two new compiled Java files:

- `AddOneServer_`skel.class
- `AddOneServer_`stub.class

There is no need to modify these files. Some IDE tools also create .java files.

7.3.4 Implementing the Client

For the client to use the methods of the remote object, it must:

- Import the package of server classes
- Register with the RMI security manager
- Use Naming.lookup to find the remote server object, which:
  - Uses the server interface object
  - Must match the port number used by the server

The client can make method calls on the remote server object like any other Java method call. Figure 237 on page 295 shows the client code.

```
package TestRMI;

import java.rmi.*;
public class UseAddOne {

public static void main(java.lang.String[] args){
int myNum = 0;

try {
System.setSecurityManager(new RMISecurityManager());
System.out.println("Main: After UseAddOne security mgr");

AddOneServerInterface myAddOne =
  (AddOneServerInterface)Naming.lookup("//sysname:6666/AddOne");

for (int i=0; i<10; i++)
{
   System.out.println("myNum = " + (myNum = myAddOne.addOne(myNum)));
}
} catch (Exception e) {e.printStackTrace();}
return;
}
}
```

Figure 237. Client Program

Before the client can use the remote methods, it must register with the RMI security manager. We use the setSecurityManager method to do this as shown in Figure 237:

```
System.setSecurityManager(new RMISecurityManager());
```

The client must obtain a reference to the remote object itself. Use the Naming.lookup method to do this. It requires a URL as an input parameter. The format of the URL is:

```
//  +  the name of the host system + the the TCP port number + the name of the
host program
```

Figure 237 shows the following code:

```
AddOneServerInterface myAddOne =
   (AddOneServerInterface)Naming.lookup("//sysname:port/AddOne");
```

7.3.5 Making the Server Code Network Accessible

To make the server code network accessible, you must complete these steps:

1. Start the RMI Registry (rmiregistry) on the server:
   - Pass in the TCP/IP port as the first parameter.
   - The CLASSPATH must provide access to all required server objects.
2. Start the server objects in a new (second) server process, which binds to the registry.
3. Invoke the client.

To run the application on the AS/400 system, start the host application. Since we are using RMI support, start the RMI registry. The registry must run in the QShell environment. Before starting the QShell environment, set the Java Environment CLASSPATH information. The registry must be able to find the application that we are running. To start the RMI registry, use the following command:

```
rmiregistry 6666
```

Next, start the application. First, set the Java Environment CLASSPATH information. There are a number of ways to do this. Set the CLASSPATH so we can find the application class and the AS/400 Toolbox for Java classes (if we are using them). Then, start the host application using this command:

```
java AddOneServer
```

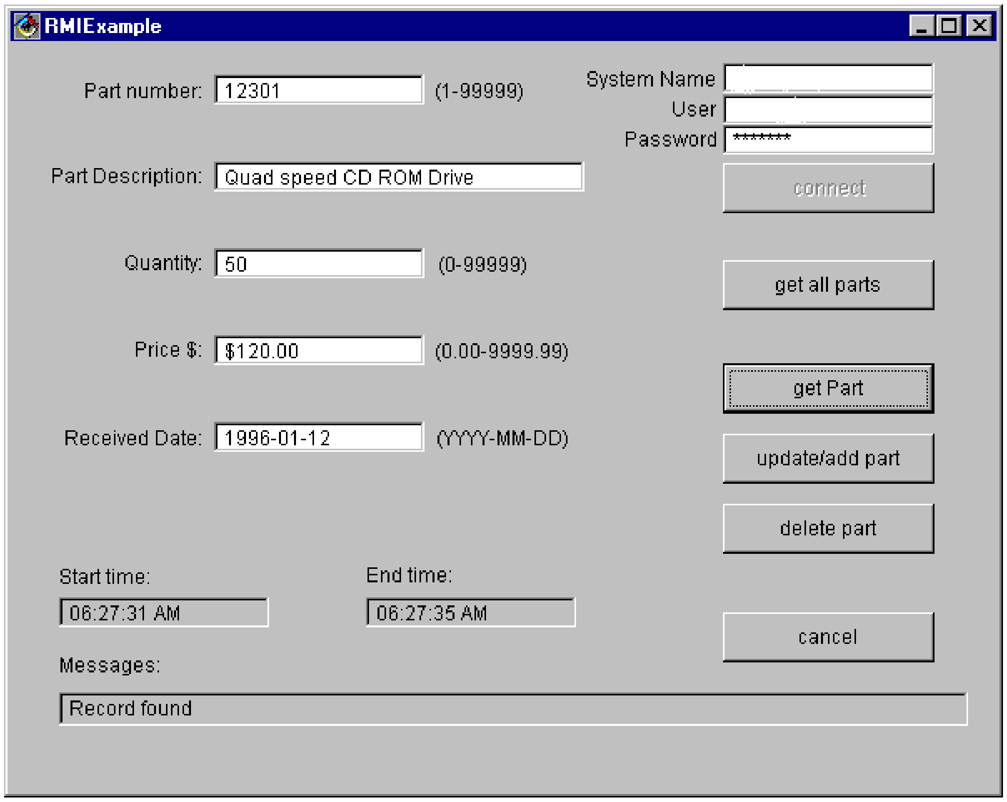
We are now ready to invoke the client. It can use the remote method supplied by the host:

```
java UseAddOne
```

7.4 RMI JDBC Example

This section explains how to implement an AS/400 client/server application using RMI. This is a "thin" client implementation. The client handles all the graphical user interface support while all the logic and database access is performed on the server. The server uses JDBC to access the AS/400 database.

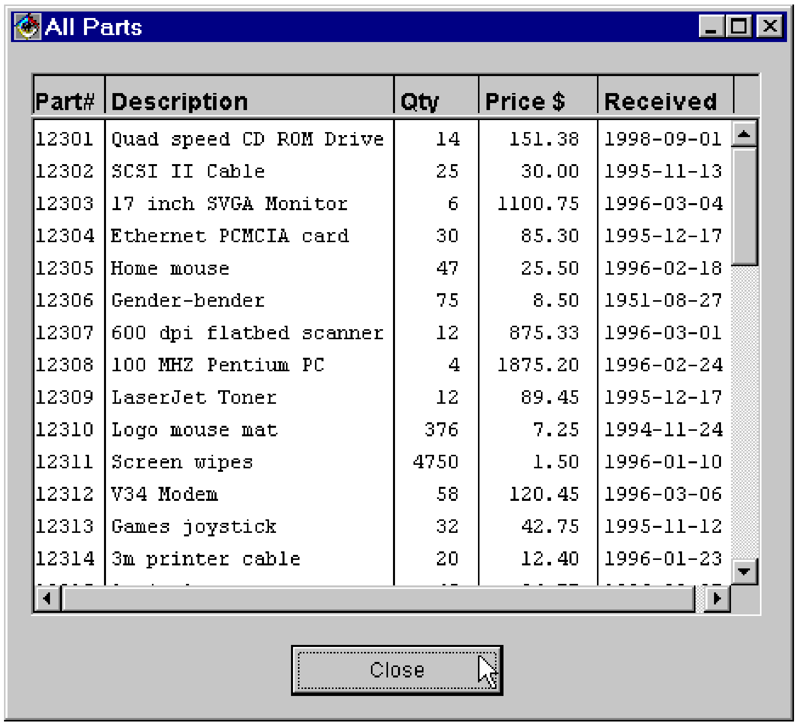
Figure 238 on page 297 shows the main window of the RMI example. To run the example, enter the name of the AS/400 system and click on the connect button. This causes the connectToDB method to run. If you successfully establish an RMI connection to the AS/400 system, you receive the message `Connected to AS/400`. This program can retrieve information about one part or all of the parts in the database.



Part number: 12301 (1-99999)
System Name
User
Password *******
Part Description: Quad speed CD ROM Drive
Quantity: 50 (0-99999)
Price $: $120.00 (0.00-9999.99)
Received Date: 1996-01-12 (YYYY-MM-DD)
Start time: 06:27:31 AM
End time: 06:27:35 AM
Messages:
Record found

Figure 238. JDBC RMI Application

Figure 239 on page 298 shows the result of clicking on the Get All Parts button. All part numbers are retrieved from the PART database and displayed in a multi-column listbox.



Part#	Description	Qty	Price $	Received
12301	Quad speed CD ROM Drive	14	151.38	1998-09-01
12302	SCSI II Cable	25	30.00	1995-11-13
12303	17 inch SVGA Monitor	6	1100.75	1996-03-04
12304	Ethernet PCMCIA card	30	85.30	1995-12-17
12305	Home mouse	47	25.50	1996-02-18
12306	Gender-bender	75	8.50	1951-08-27
12307	600 dpi flatbed scanner	12	875.33	1996-03-01
12308	100 MHZ Pentium PC	4	1875.20	1996-02-24
12309	LaserJet Toner	12	89.45	1995-12-17
12310	Logo mouse mat	376	7.25	1994-11-24
12311	Screen wipes	4750	1.50	1996-01-10
12312	V34 Modem	58	120.45	1996-03-06
12313	Games joystick	32	42.75	1995-11-12
12314	3m printer cable	20	12.40	1996-01-23

Figure 239. RMI Example — Get All Parts

Figure 240 shows the RMI example application design.

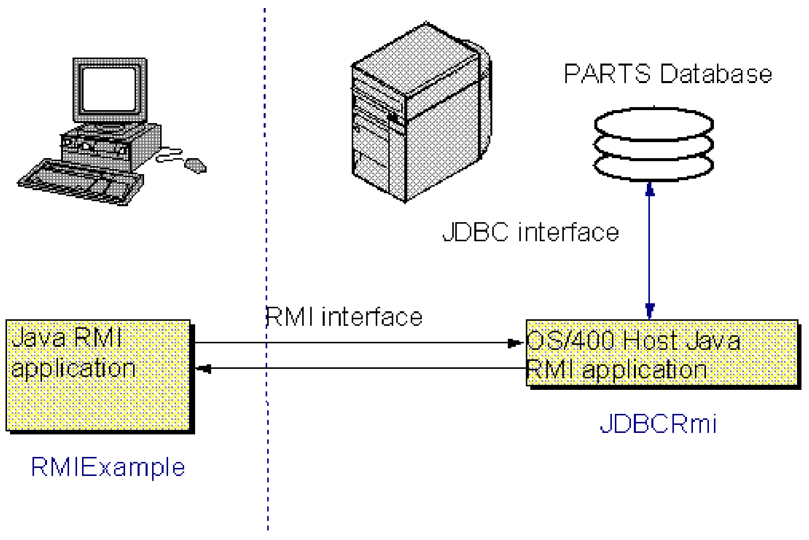


Figure 240. AS/400 RMI Example

The client Java program requests data from the AS/400 database by sending requests to the Host Java program. The Host Java program uses JDBC to access the PARTS database. Two SQL statements are used:

- `Select * from apilib.parts` to retrieve all columns for all records
- `Select * from apilib.parts where partno =?` to retrieve all columns for a given part number

The entire application is kept in a package named JDBCRmi as shown in Figure 241.

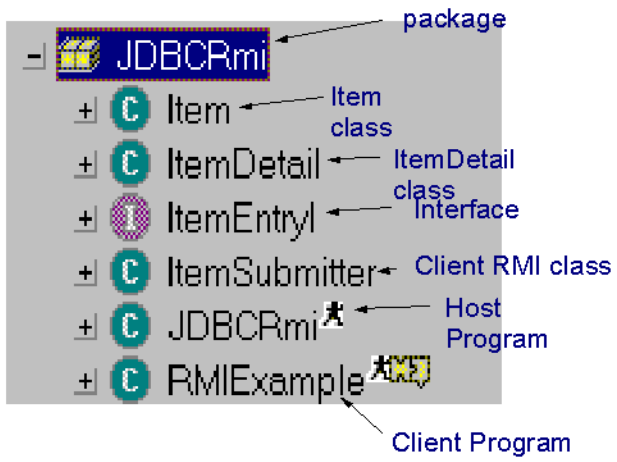


Figure 241. Java Package for the RMI Example

This package contains the following classes:

- **Item** — Used to create Item objects
- **ItemDetail** — Used to create Item Detail objects
- **ItemEntryI** — Contains the interface implemented to support RMI
- **ItemSubmitter** — Used to create an RMI support object for the client
- **JDBCRmi** — The AS/400 Host Java program
- **RMIExample** — The client Java program

Figure 242 on page 300 shows the program interface. RMIExample is the Java client program. It creates an instance of the ItemSubmitter class. The ItemSubmitter class contains all the RMI support for the client side. ItemEntryI is the interface, which describes the public methods. It is used by ItemSubmitter and JDBCRmi (the host Java program).

The Item class is used to pass information about an item between the host java program (JDBCRmi) and the client Java program (RMIExample). If a request is made for all the items in the database, an item object is created that contains ItemDetail objects for each record in the database.

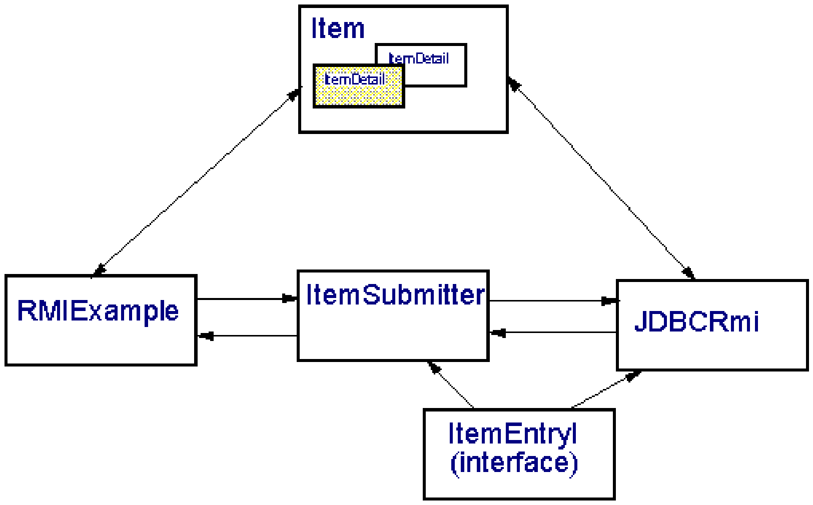


Figure 242. RMI Application Design

Figure 243 shows how the public methods are used. ItemEntryI is the interface that describes the public methods.

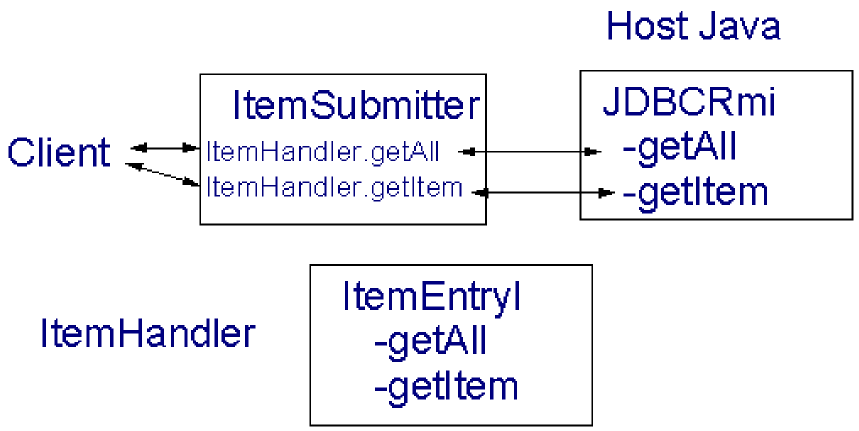


Figure 243. RMI Example Public Methods

There are two public methods:

- public Item getAll() throws RemoteException
- public Item getItem(String anItem) throws RemoteException

The getAll method takes no input parameters and returns an Item object. The getItem method takes a String input parameter, which contains the part number requested and returns an Item object, which contains the details about an Item.

To separate the RMI logic from the application logic, the ItemSubmitter class is used by the client program. The client program instantiates an instance of the ItemSubmitter class. It is called ItemHandler, and the client program uses the ItemHandler object to interface to the public methods.

On the host side, JDBCRmi contains the actual application logic for the public methods.

The getAll method uses the JDBC interface to retrieve all records from the PARTS database. The getItem method uses JDBC to retrieve the requested item from the PARTS database. In both methods, an Item object is returned to the client program.

7.4.1 Item Class

The Item class is used to pass information about a particular item between the client program and the server program. It contains the Item identification, Item description, Item price, Item quantity, and Item date. It also contains an array that can contain up to 100 ItemDetail objects. The ItemDetail objects are used when a request for all items (or parts) is made. In this case, one Item object is returned that contains an ItemDetail object for each item in the PARTS database. The Item class is shown in Figure 244.

```
import java.io.Serializable;
public class Item implements Serializable
{
private StringBuffer ItemId = new StringBuffer(5);
private String ItemDesc;
private java.math.BigDecimal ItemPrice;
private int ItemQuantity;
private String ItemDate;
private ItemDetail[] entryArray = new ItemDetail[100];
}
```

Figure 244. Item Class

The Item class also provides a number of methods for accessing or changing information in the Item object(getters/setters). Methods are also provided for determining the number of ItemDetail objects contained in the Item object and for retrieving the ItemDetail objects.

The Item class implements Serializable. This is required to allow objects to be passed as parameters over a communication network.

The ItemDetail class is used to pass information about all items between the client program and the server program. It contains the Item identification, Item description, Item price, Item quantity, and the Item date. The ItemDetail objects are used when a request for all items (or parts) is made. In this case, one Item object is returned that contains an ItemDetail object for each item in the PARTS database.

The ItemDetail class also provides a number of methods for accessing information in the ItemDetail object(getters). The ItemDetail class implements Serializable. This is required to allow objects to be passed as parameters over a network. The ItemDetail class is shown in Figure 245 on page 302.

```
public class ItemDetail implements Serializable {
StringBuffer itemId = new StringBuffer(6);
StringBuffer itemDsc = new StringBuffer(24);
String itemPrice;
String itemQty;
     String itemDate;
}
```

Figure 245. ItemDetail Class

7.4.2 Defining the Interface

When using Java RMI support, the public methods must be described in an interface. The interface must extend the Remote class. The public methods must throw a RemoteException. The interface is named ItemEntryI and is shown in Figure 246.

```
import java.rmi.*;
public interface ItemEntryI extends Remote {
public Item getAll() throws RemoteException;
public Item getItem(String anItem) throws RemoteException;
}
```

Figure 246. ItemEntryI Interface

7.4.3 Implementing the Remote Server Objects

The code example in Figure 247 shows the class description for the host Java program. We import a number of support classes, including the RMI support classes. To use RMI, we must extend the UnicastRemoteObject class and implement the ItemEntryI interface.

```
import com.ibm.as400.access.*;   //Toolbox classes
import java.math.*; // for BigDecimal class
import java.sql.*; // for JDBC classes
import java.util.*; // for Properties class
import java.text.*; // for DateFormat class
import java.rmi.*; // for Remote Method Invocation
import java.rmi.registry.*;
import java.rmi.server.*;
public class JDBCRmi extends UnicastRemoteObject implements ItemEntryI
{
private static final String SYSTEM = "SysName";  //AS/400 system name
private static final String USER = "*current";
private static final String PASSWORD = "*current";
private static final String DATA_LIBRARY = "TEAMxx";
//  global connection and prepared statements
private Connection dbConnect = null;
private PreparedStatement psAllRecord;
private PreparedStatement psSingleRecord;

}
```

Figure 247. JDBCRmi Class

The UnicastRemoteObject class defines the remote object as a unicast object, which means that only a single instance of the object can exist on a single server. This is distinguished from a MultiCastRemoteObject, which can replicate across multiple servers.

The class must implement an interface that describes the public methods. In the example used here, the interface is named ItemEntryI. It does not need to be called ItemEntryI, but such a naming convention helps keep the links between the classes clear. The Interface class must import the java.rmi package to use the RMI classes.

We declare some global variables that are used in the application. The value *current means to use the information for the current user that is signed on. We also declare an SQL Connection object and two PreparedStatement objects that are used to access the AS/400 database through the JDBC interface.

Each remote class must be capable of registering its services with an RMI registry that provides brokering services between the client and the server. We do this by adding a main method that performs the registration. The RMI classes throw exceptions so we must wrap our use of these classes in a try{} catch{} block. The host main method is shown in Figure 248.

```
public static void main(String[] parameters)
{
// main must be invoked with 1 parameter: the port number
// this should be the same port number on which the
// particular Registry has been started
if(parameters.length<1)
{
System.err.println("Must pass port number when invoking.");
System.exit(1);
  }
     String port = ":"+parameters[0];
  // Set up the server
try
{
System.out.println("Main: Attempting to register JDBCRmi");
System.setSecurityManager(new RMISecurityManager());
JDBCRmi oeJDBC = new JDBCRmi();
Naming.rebind("//"+SYSTEM+port+"/JDBCRmi", oeJDBC);
System.out.println("Main: Successfully registered with the security
manager");
} catch(Exception e) {e.printStackTrace();}
return;
}
```

Figure 248. JDBCRmi Main Method

The host initialize method shown in Figure 249 on page 304 demonstrates how the JDBC environment is set up. It shows using the AS/400 Native JDBC driver to access the AS/400 system database.

```
private void initialize () throws Exception
{
   // Create a properties object for JDBC connection
Properties jdbcProperties = new Properties();
// Set the properties for the JDBC connection
jdbcProperties.put("user", USER);
jdbcProperties.put("password", PASSWORD);
jdbcProperties.put("naming", "sql");
jdbcProperties.put("errors", "full");
jdbcProperties.put("date format", "iso");

Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");
// Connect using the properties object
dbConnect =
      DriverManager.getConnection("jdbc:db2:"+SYSTEM+"/"+DATA_LIBRARY,
             jdbcProperties);
psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS
              WHERE PARTNO = ?" );
psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS order
              by partno");
```

Figure 249. Initializing the JDBC Connection

To connect to the AS/400 database, we use the DriverManager.getConnection() method. The DriverManager.getConnection() method takes a URL string as an argument. The JDBC driver manager attempts to locate a driver that can connect to the database represented by the URL. When using the native AS/400 Java driver, we use the following syntax for the URL:

```
jdbc:db2:systemName/defaultLibrary;listOfProperties
```

An SQL statement can be compiled and stored in a PreparedStatement object. The Java program can use this object to run the statement multiple times since the statement is compiled only once. This is more efficient than running the same statement multiple times using a Statement object, which compiles the statement each time it is run. We use the Connection.prepareStatement() method to create PreparedStatement objects.

Next, we create the code for the public method getItem. The getItem method is shown in Figure 250 on page 305. We first create a new Item object that we return to the caller if we successfully find the item in the database. A ResultSet object provides access to a table of data generated by running a statement.

```
public Item getItem (String anItem) throws RemoteException
{
Item theItem = new Item(anItem);
try
{
java.sql.ResultSet rs = null;
 psSingleRecord.setInt(1, Integer.parseInt(anItem));
     rs = psSingleRecord.executeQuery();
          if (rs.next()) {
         theItem.setItemDesc(rs.getString("PARTDS"));
         theItem.setItemQty(rs.getInt("PARTQY"));
         theItem.setItemDate(rs.getDate("PARTDT").toString());
         theItem.setItemPrice(rs.getBigDecimal("PARTPR", 2));
 }
    else {
    return(null);
         }
  } catch  (Exception e) {e.printStackTrace(); return null; }
   return(theItem);
 }
```

Figure 250. The getItem remote Method

The table rows are retrieved in sequence. Within a row, column values can be accessed in any order:

- **java.sql.ResultSet rs = null;**

  Declares a variable, rs, to reference a ResultSet object.

- **psSingleRecord.setInt(1, Integer.parseInt(partNo));**

  Uses the PreparedStatement method, setInt, to set the value of parameter 1 to the integer value of the part number passed on the parameter list.

- **rs = psSingleRecord.executeQuery();**

  Executes the SQL defined by the psSingleRecord PreparedStatement object and places the table of resulting records in a ResultSet object referenced by rs.

- **if (rs.next())**

  The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. Since this is the first method read from the ResultSet, the method positions to the first record from the ResultSet and returns true. If there are no records to retrieve, the method returns a false value.

  The following lines retrieve values of database fields and place them in the Item object that is returned:

  – theItem.setItemDesc(rs.getString("PARTDS"));
  – theItem.setItemQty(rs.getInt("PARTQY"));
  – theItem.setItemDate(rs.getDate("PARTDT").toString());
  – theItem.setItemPrice(rs.getBigDecimal("PARTPR", 2));

  We use the setter provided by the Item class to do this. ResultSet objects have getter methods for many Java data types.

Here we use:

– *getString* — Returns the value of the column PARTDS as a String object
– *getInt* — Returns the value of the column PARTQY as an integer
– *getBigDecimal* — Returns the value of the PARTPR field as a BigDecimal object
– *getDate* — Returns the value of column PARTDT as a Date

If we successfully find the item number in the database, we return the Item object. Otherwise, we return null. Next, we write the code for the getAll method.

The getAll method is shown in Figure 251. To get all the records, we execute the Prepared Statement object named psAllRecord. It does not require any parameters. In this case, we have multiple rows returned in the result set. We use the result set, Next method, to retrieve each row from the result set. For each row returned, we create a ItemDetail object in the Item object that we return to the caller.

```
public Item getAll () throws RemoteException
{
Item theItem = new Item("all");
java.sql.ResultSet rs = null;
String[] detailRow = new String[5];
try {
rs = psAllRecord.executeQuery();
while (rs.next()) {
                             ItemDetail detail = new ItemDetail
(rs.getString("PARTNO"),
                        rs.getString("PARTDS"),
                        Integer.toString(rs.getInt("PARTQY")),
                        "$" + rs.getBigDecimal("PARTPR", 2).toString(),
                         rs.getDate("PARTDT").toString());
                        theItem.addEntry(detail);
                        }
  } catch     (Exception e) {e.printStackTrace(); return null;}
 return(theItem);
}
```

Figure 251. The getAll Method

7.4.4 Creating the Stubs and Skeletons

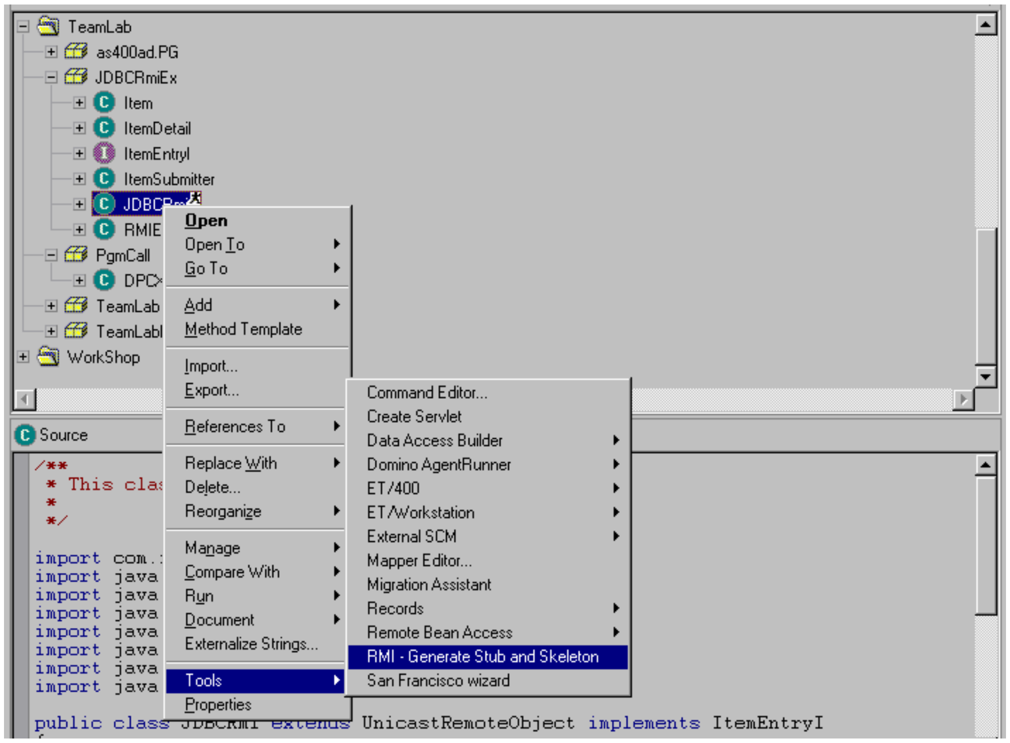
To make the remote class ready to use, you must complete this process:

1. Compile server classes using **javac** or an IDE.
2. Run `rmic` or its equivalent function from an IDE.
    - In the JDK, the command is: `rmic JDBCRmi`
    - From VisualAge for Java, follow this sequence **Tools—>Remote Method Invocation—>Generate Proxies** (as shown in Figure 252 on page 307).

The output from the rmic command is two new compiled Java files:

- **JDBCRmi**_skel.class
- **JDBCRmi**_stub.class

There is no need to modify these files. Some IDE tools also create .java files.



```
/**
 * This clas
 *
 */

import com.:
import java
import java
import java
import java
import java
import java
import java

public class JDBCRmiI extends UnicastRemoteObject implements ItemEntryI
```

Figure 252. Creating the Stubs and Skeletons in VisualAge for Java

After running RMI Create Stub and Skeleton, our package now contains all of the required host remote classes as shown in Figure 253 on page 308.

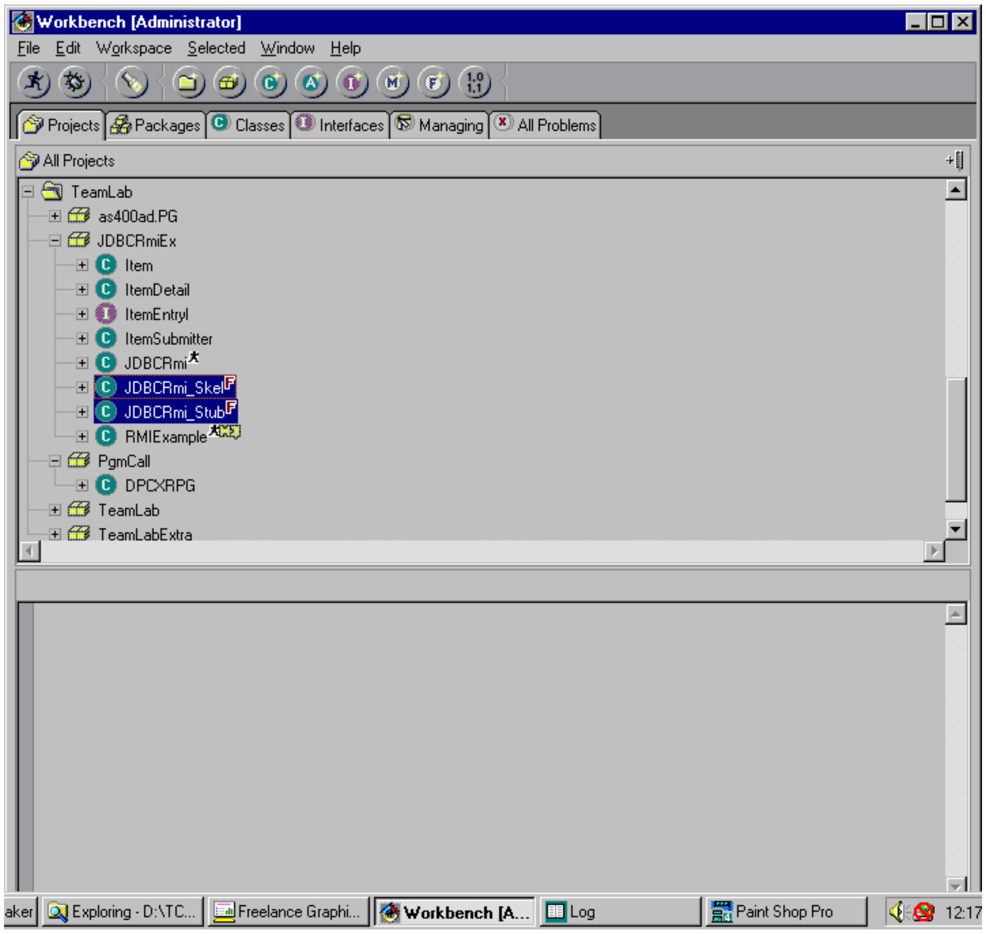
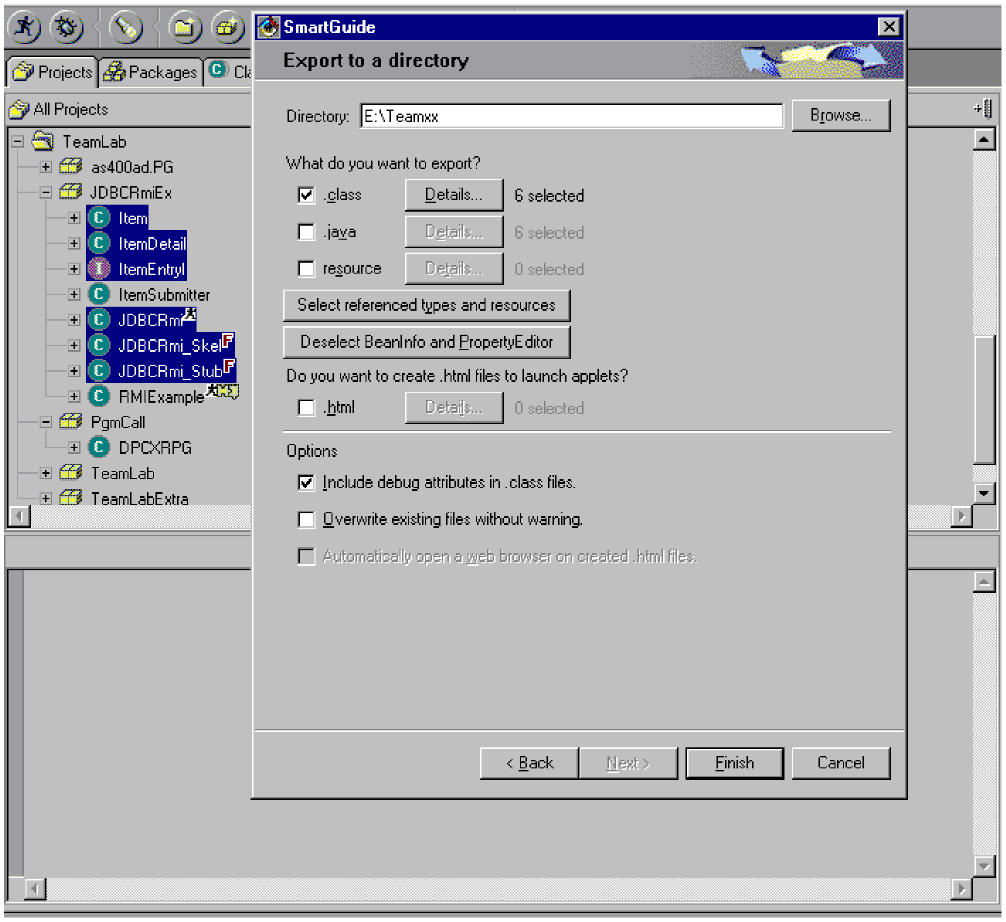


Figure 253. Completed Host Remote Application

Before we can run the application on the AS/400 system, we have to move it there. If the AS/400 Integrated File System (IFS) is available as a network drive, we can export the host classes to the AS/400 system. We can also use the ET/400 export function. Please see Section 8.6, “Support for Export, Compile, Run, and Debug AS/400 Programs” on page 330, for details about using ET/400 and the export function.

In this case, we have a network drive assigned to the AS/400 system, so we directly export the classes using the VisualAge for Java export function. When we run the application on the AS/400 system, we have to set the CLASSPATH environment variable to include the AS400 IFS file where we stored the class files. In Figure 254 on page 309, we export the classes to the AS/400 system using a network drive.


SmartGuide

Export to a directory

Directory: E:\Teamxx

What do you want to export?

- [✓] .class — 6 selected
- [ ] .java — 6 selected
- [ ] resource — 0 selected

Do you want to create .html files to launch applets?

- [ ] .html — 0 selected

Options

- [✓] Include debug attributes in .class files.
- [ ] Overwrite existing files without warning.
- [ ] Automatically open a web browser on created .html files.

Figure 254. Exporting the Class Files

7.4.5 Implementing the Client

This section explains how to build the client class of the RMI application. We use the ToolboxGUI class to create a new class that is a subclass of java.awt.Frame and implements the PartsContainer interface. For details about the ToolboxGUI class and the PartsContainer interface, please refer to Section 3.5.6, “Reusable GUI Part” on page 126.

The name of the client program is RMIExample. As shown in Figure 255, we create global variables for the port name and system name and create a new instance of the ItemSubmitter class, which we call remoteRequestor. We use the remoteRequestor object for all of our RMI work.

```
public class RMIExample extends java.awt.Frame implements
java.awt.event.WindowListener, WorkShop.PartsContainer
 {

static String port = null;
static String systemName = null;
ItemSubmitter remoteRequestor = new ItemSubmitter();
}
```

Figure 255. Creating the Client Class

The RMIExample main method was created by VisualAge for Java.

```
public static void main(java.lang.String[] args) {
try {
JDBCRmi.RMIExample aRMIExample = new
                  JDBCRmi.RMIExample();
try {
Class aCloserClass = Class.forName("uvm.abt.edit.WindowCloser");
Class parmTypes[] = { java.awt.Window.class };
Object parms[] = { aRMIExample };
port = args[0];
java.lang.reflect.Constructor aCtor =
                aCloserClass.getConstructor(parmTypes);
aCtor.newInstance(parms);
} catch (java.lang.Throwable exc) {};
aRMIExample.setVisible(true);
} catch (Throwable exception) {
System.err.println("Exception occurred in main() of java.awt.Frame");
}
}
```

Figure 256. RMIExample Main Method

When the application is run, main instantiates a new instance of the RMIExample class. We only have to modify it to use the first argument of the parameter list to set the port variable.

The connectToDB method shown in Figure 257 is executed when the user clicks on the Connect button.

```
public void connectToDB(String systemName, String userid,
        String password) throws Exception {
this.systemName = systemName;
remoteRequestor.linked(systemName,port);
return;
}
```

Figure 257. RMIExample CcnnectToDB Method

We use the system name from the screen TextField as a parameter to set the systemName variable. We call the linked method of the remoteRequestor object, which was instantiated from the ItemSubmitter class to establish the RMI connection.

The getRecord method, shown in Figure 258 on page 311, is called when the user clicks on the Get Part button.

```
public String getRecord(String partNo, java.awt.TextField partDesc,
     java.awt.TextField partQty, java.awt.TextField partPrice,
     java.awt.TextField partDate) throws Exception {

Item rtnItem = new Item(partNo);

rtnItem = remoteRequestor.submit(partNo);
 if ((rtnItem) != null)
  {
partDesc.setText(rtnItem.getItemDesc());
partDate.setText(rtnItem.getItemDate());
partQty.setText((Integer.toString(rtnItem.getItemQuantity())));
partPrice.setText("$" + (rtnItem.getItemPrice()).toString());
  }
else {

     partDesc.setText("");
 partDate.setText("");
 partQty.setText("");
     partPrice.setText("");
     return "Record not found";
   }
return "Record found";
}
```

Figure 258. RMIExample getRecord Method

The getRecord method calls the submit method of the remoteRequestor object passing the part number as an input parameter. An Item object is returned that contains the details about the Item (or Part).

If an Item object is returned, the item requested was found in the database. We use the getter methods provided by the Item object to retrieve the information and display it on the screen.

The populateListBox method, shown in Figure 259, is called when the user clicks on the Get All Parts button. It calls the submitAll method of the remoteRequestor object passing no parameters.

```
public void populateListBox(com.ibm.ivj.eab.dab.IMulticolumnListbox
aListBox) throws Exception {
Item rtnItem = new Item("all");
ItemDetail rtnDetail = new ItemDetail();
rtnItem = remoteRequestor.submitAll();
```

Figure 259. RMIExample populateListBox Method

An Item object is returned that contains an array of ItemDetail objects, which contain the details about each Item (or Part) in the database. If an Item object is returned, we use the getNumEntries to determine how many ItemDetail objects were returned.

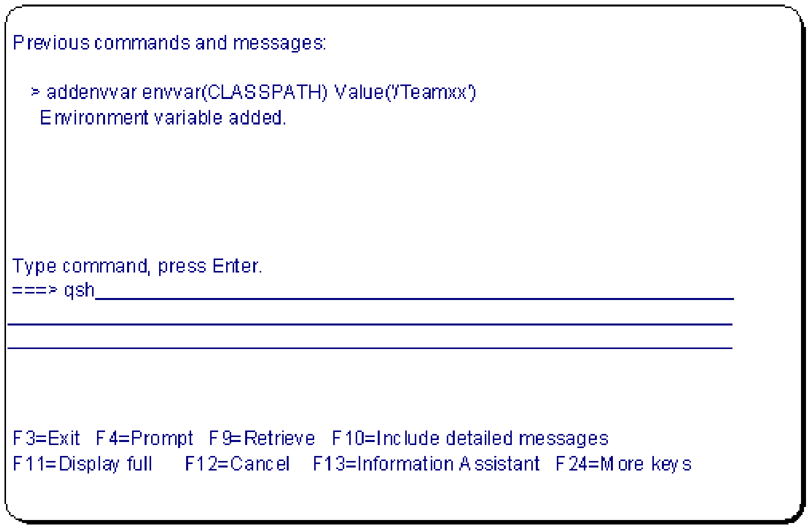
```
 if ((rtnItem) != null) {
System.out.println("back from host");
rtnDetail = rtnItem.getFirstEntry();
for (int i = 0; i < rtnItem.getNumEntries(); i++) {
String[] array = new String[5];
array[0] = rtnDetail.getItemId();
array[1] = rtnDetail.getItemDsc();
array[2] = rtnDetail.getItemQty();
array[3] = rtnDetail.getItemPrice();
array[4] = rtnDetail.getItemDate();
aListBox.addRow(array, array[0]);
rtnDetail = rtnItem.getNextEntry();
}
} else {
return;
}
return;
 }
```

Figure 260. Populating the ListBox

We use the getter methods provided by the ItemDetail object to retrieve the information and add it to the array, which is used to populate the multi-column listbox.

7.4.6 Making the Server Code Network Accessible

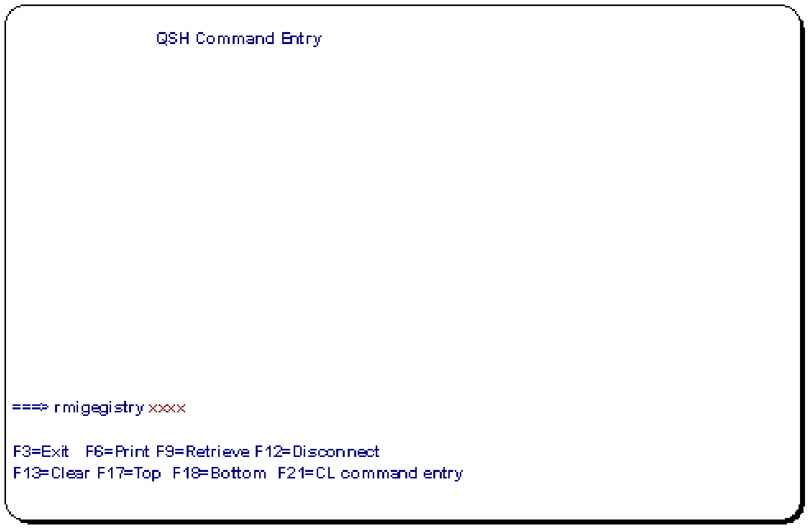
To run the application, we first start the host application on the AS/400 system. Since we are using RMI support, we start the RMI registry. The registry must run in the QShell environment. Before starting the QShell environment, we set the Java Environment CLASSPATH information. There are a number of ways to do this. Here, we use the add environment variable (addenvvar) command. The registry must be able to find the application that we are running. We start the RMI registry using the AS/400 rmiregistry command. Setting the CLASSPATH environment variable and starting the QShell environment is shown in Figure 261 on page 313.



```
Previous commands and messages:

  > addenvvar envvar(CLASSPATH) Value('/Teamxx')
   Environment variable added.




Type command, press Enter.
===> qsh



F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
F11=Display full     F12=Cancel   F13=Information Assistant   F24=More keys
```

Figure 261. Setting the CLASSPATH Environment Variable

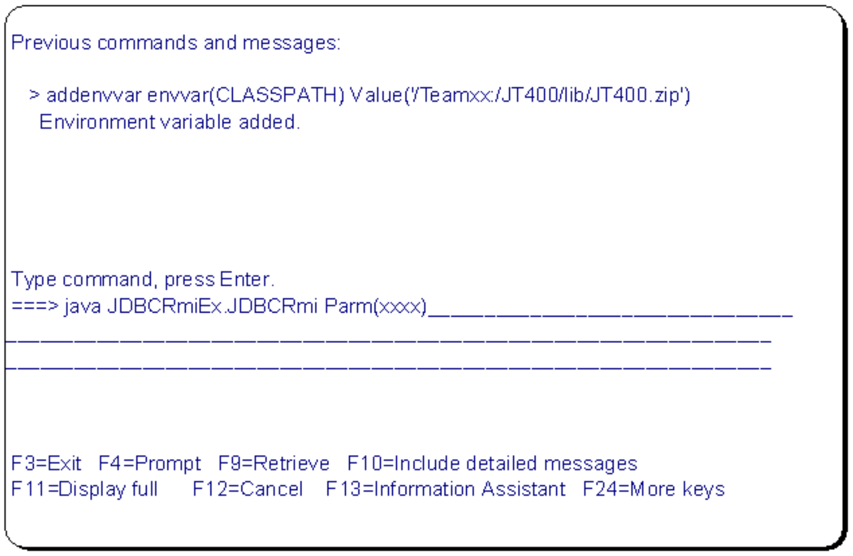
In Figure 262, we start the RMI registry using the rmiregistry CL command. We must pass in the TCP port number we are using as a parameter. The rmiregistry command does not support the verbose parameter. In this case, we start the RMI registry using a 5250 emulation session. We can also submit this as a batch job and not tie up a 5250 session.



```
                  QSH Command Entry









===> rmigegistry xxxx

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

Figure 262. Starting the RMI Registry

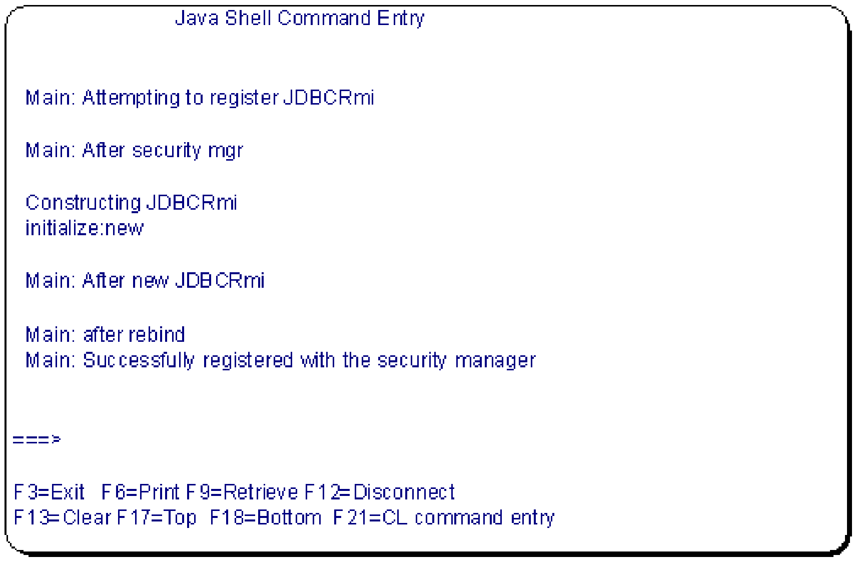
We next start the host application. As shown in Figure 263 on page 314, we first set the Java Environment CLASSPATH information. There are a number of ways to do this. Here, we use the add environment variable (addenvvar) command. We must set the CLASSPATH so we can find the application class and if we are using them, the AS/400 Toolbox for Java classes. Then, we use the java CL command to start the application passing in the port number as a parameter.



```
Previous commands and messages:

  > addenvvar envvar(CLASSPATH) Value('/Teamxx:/JT400/lib/JT400.zip')
    Environment variable added.







Type command, press Enter.
===> java JDBCRmiEx.JDBCRmi Parm(xxxx)____________________________
__________________________________________________________________
__________________________________________________________________



F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
F11=Display full      F12=Cancel    F13=Information Assistant   F24=More keys
```

Figure 263. Starting the Host Application

Figure 264 shows the host application screen after a successful start. The messages are output by the application writing to a standard out using System.out.println.



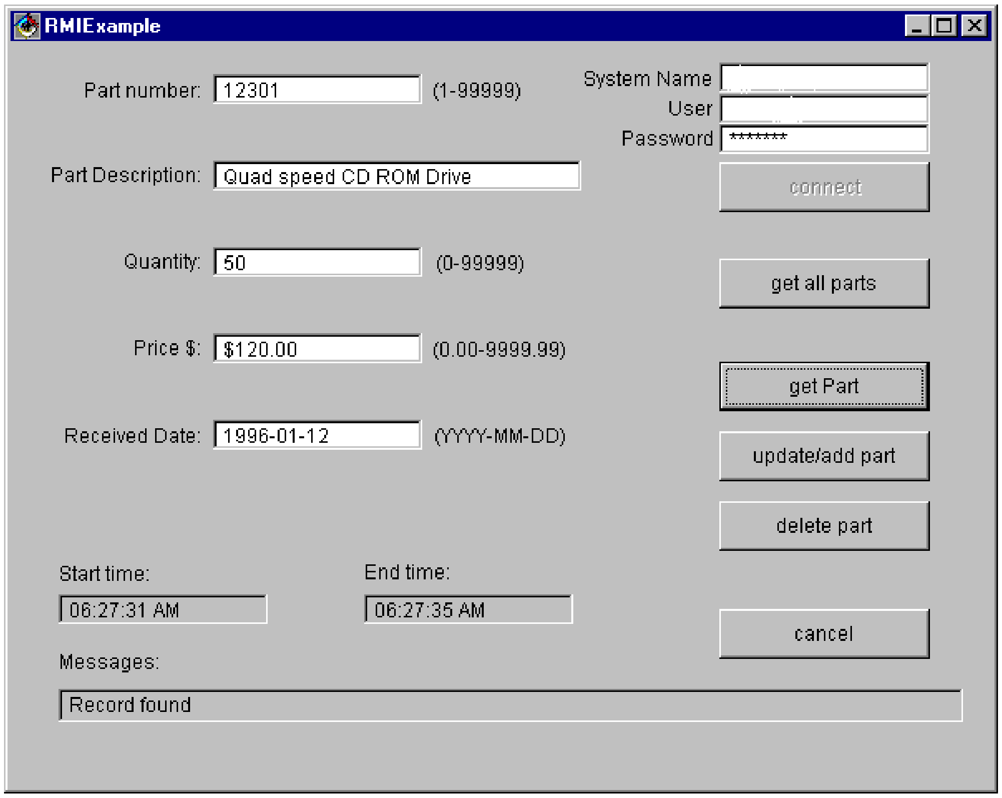
```
                    Java Shell Command Entry


  Main: Attempting to register JDBCRmi

  Main: After security mgr

  Constructing JDBCRmi
  initialize:new

  Main: After new JDBCRmi

  Main: after rebind
  Main: Successfully registered with the security manager


===>

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

Figure 264. Host Application Successful Start

We are now ready to run the client application. We have the registry started and the host application started.

After starting the client application, perform the following tasks:

1. Enter the AS/400 system name, user ID, and password. Click on **Connect**.
2. After a `Connected to AS/400` message is received, complete the following steps:
   a. Enter a part (item) number and click on the **Get Part** button. Valid numbers are 12301 through 12350.
   b. Click on the **Get All Parts** button to display all the records in a listbox.


RMIExample

Part number: 12301 (1-99999)

Part Description: Quad speed CD ROM Drive

Quantity: 50 (0-99999)

Price $: $120.00 (0.00-9999.99)

Received Date: 1996-01-12 (YYYY-MM-DD)

System Name

User

Password ********

Start time: 06:27:31 AM

End time: 06:27:35 AM

Messages:

Record found

Figure 265. Running the JDBC RMI Application

7.5 Conclusion

In this chapter, we showed you how to build AS/400 client/server applications using the Java remote method invocation (RMI) support. The advantages of using RMI are:

- Both the client and server application are written in Java. The programmer only needs to work in one language.
- The Java code is platform independent. It is easy to write Java code that can run on either the client or server.
- Calling the remote method is transparent. Calling a method using RMI is the same as calling a local method. This makes it easy to extend or modify programs by moving the actual location of the methods being used.
- Objects can be passed between the client and server code. Even though a method may reside on a remote platform, objects can be passed to it and received back from it. This makes it much easier to interface with programs running on other platforms than passing parameters. When passing parameters, we have to implement platform-unique solutions to pass the parameters from the client to the host.

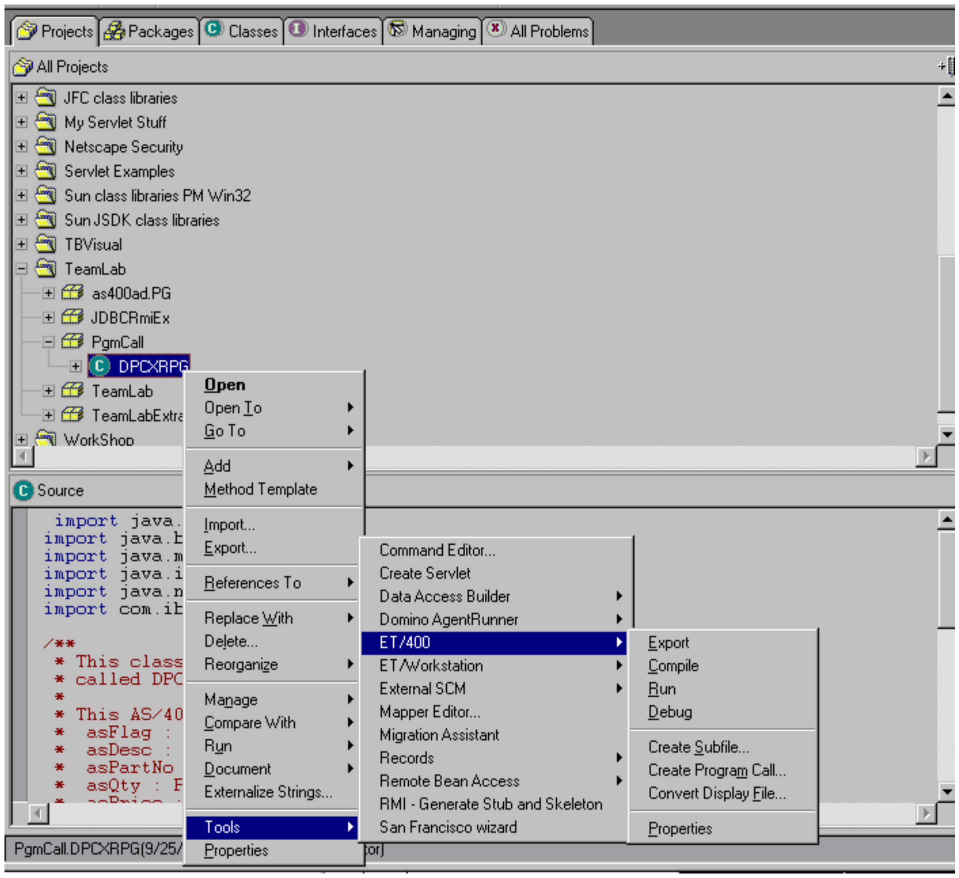
Chapter 8. IBM Enterprise Toolkit for AS/400

The VisualAge for Java 2.0 Enterprise edition includes the IBM Enterprise Toolkit for AS/400 (ET/400). ET/400 provides AS/400 unique Java support. It provides SmartGuides that allow you to use AS/400 Toolbox classes easier (Program Call SmartGuide). Plus, SmartGuides to make it easy to develop and implement AS/400 Java programs using VisualAge for Java. ET/400 also offers the ability to debug Java programs running on the AS/400 system from your workstation. In previous versions of VisualAge for Java, this support was available as VisualAge for Java - AS/400 Feature. AS/400 Feature was external to VisualAge for Java. The new ET/400 support is integrated into the VisualAge for Java 2.0 Integrated Development Environment. ET/400 includes:

- The AS/400 Toolbox for Java classes that can be used to access AS/400 resources and services. These are the same classes shipped with the AS/400 Toolbox for Java, product 5763-JC1.
- A SmartGuide to generate Java classes and beans for remote program calls to AS/400 programs written in RPG, COBOL or C.
- A SmartGuide to convert AS/400 display file records to Java Abstract Windowing Toolkit (AWT) classes.
- A SmartGuide to create a subfile from an AS/400 database file.
- Support to export, compile, run, and debug AS/400 Java applications from the VisualAge for Java Integrated Development Environment (IDE).

8.1 Using ET/400

The ET/400 tools are available from the VisualAge for Java Integrated Development Environment. To access the ET/400 tools, select **ET/400** from the Tools pop-up menu option as shown in Figure 266 on page 318.



```
 import java.
import java.b
import java.m
import java.i
import java.n
import com.ib

/**
 * This class
 * called DPC
 *
 * This AS/40
 *   asFlag :
 *   asDesc :
 *   asPartNo
 *   asQty : P
```

Figure 266. ET/400 Tools

8.2 AS/400 Toolbox for Java Classes

If you want to use the AS/400 Toolbox for Java classes inside the VisualAge for Java Integrated Development Environment, you must import these classes inside the IDE. ET/400 simplifies this process. After you install VisualAge for Java 2.0 Enterprise edition, the Toolbox classes are available in the repository as part of the IBM Enterprise Toolkit for AS/400 project. If you want to use the Toolbox classes, perform these steps:

1. From the workbench, click on **File** and **Quick Start**.
2. Click on **Features**, **Add Feature**, and **OK**.
3. Select **IBM Enterprise Toolkit** and **OK**.

This adds the toolbox classes to your workspace. The IBM Enterprise Toolkit for AS/400 is listed under All Projects.

The alternative is to perform these tasks:

1. Install LPP 5763-JC1 on an AS/400 system.
2. Download the classes to your workstation.
3. Import the classes into the VisualAge for Java IDE.

8.3 Distributed Program Call SmartGuide

The AS/400 Toolbox for Java Distributed Program Call class allows you to call AS/400 programs from a client Java application. The Create Program Call SmartGuide simplifies using this interface. With the Create Program Call SmartGuide, it is not necessary to code the program call. You just fill in the AS/400 name, the name of the AS/400 program your Java program will call, and the parameters you want to pass. The SmartGuide generates the Java code for you. The data conversions between the AS/400 data types and the Java data types are handled for you through the AS/400 Toolbox for Java classes.

Program Call SmartGuide input includes:

- AS/400 Program to Call (any *PGM object)
- AS/400 library where the program is stored
- Parameter list of data types, lengths, and decimals
- Name of the JavaBean to generate
- Name of the Java package
- Name of the Java project

For the output, JavaBean calls the AS/400 program that uses the AS/400 Toolbox for Java program call class. The output usage is:

- As is — Manually code to Java generated code
- In the Visual Composition Editor as a non-visual part
- Used on a client or server

The limitations include:

- The number of parameters are currently limited to 35 in a program call.
- Program and library names cannot contain the "." character.
- The Save Settings as: field cannot start with a "#" character.

For a complete example of using the Program Call SmartGuide, please refer to Section 9.4, “Creating a Program Call JavaBean” on page 345.

8.4 SmartGuide to Convert AS/400 Display Files to Java

The Convert Display File SmartGuide allows you to quickly convert existing 5250 screens to Java. It converts existing AS/400 display file records to Java Abstract Windowing Toolkit (AWT) code. You can use the generated Java code as a bean in the Visual Composition Editor, or runnable code as an application or applet.

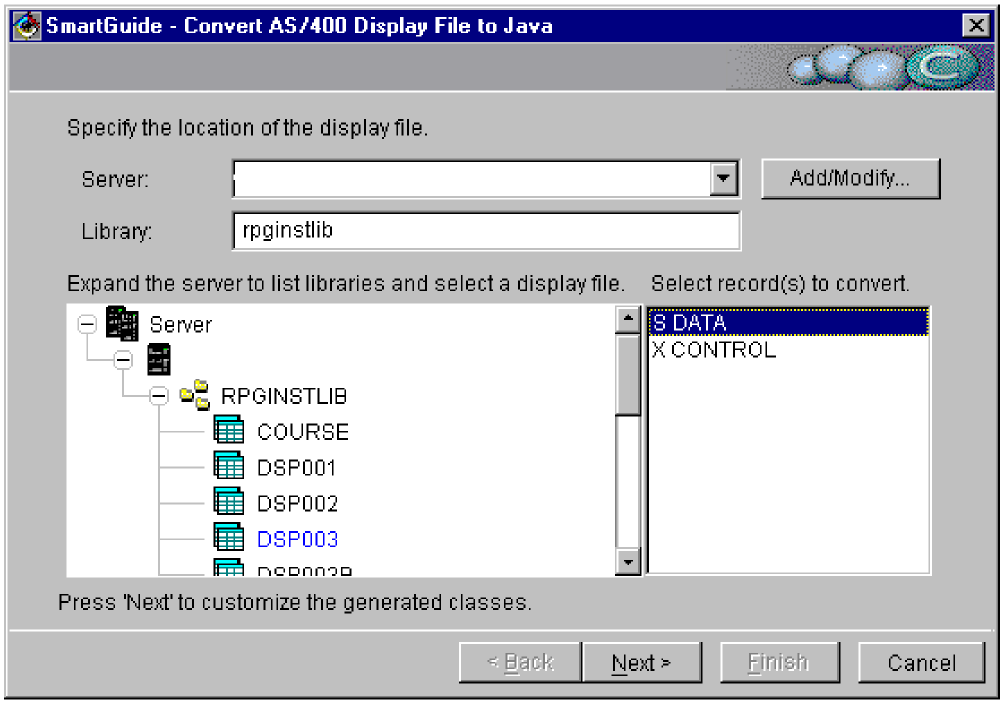
The input includes the AS/400 display file record format to convert. The output may include:

- JavaBean or class-per-record format
- Java AWT, which is used where possible
- Special AWT classes that are created for:
  - Subfiles
  - Datatype aware entry fields
  - Edit codes, edit words
  - Date and time constants

The output usage is:

- As is — Manually code to the Java generated code
- In the Visual Composition Editor as visual parts

The SmartGuide allows you to select which DDS file you want to use as input for the conversion. In Figure 267, we select DSP003.


SmartGuide - Convert AS/400 Display File to Java

Specify the location of the display file.

Server:

Library: rpginstlib

Expand the server to list libraries and select a display file.

```
Server
    RPGINSTLIB
        COURSE
        DSP001
        DSP002
        DSP003
        DSP003B
```

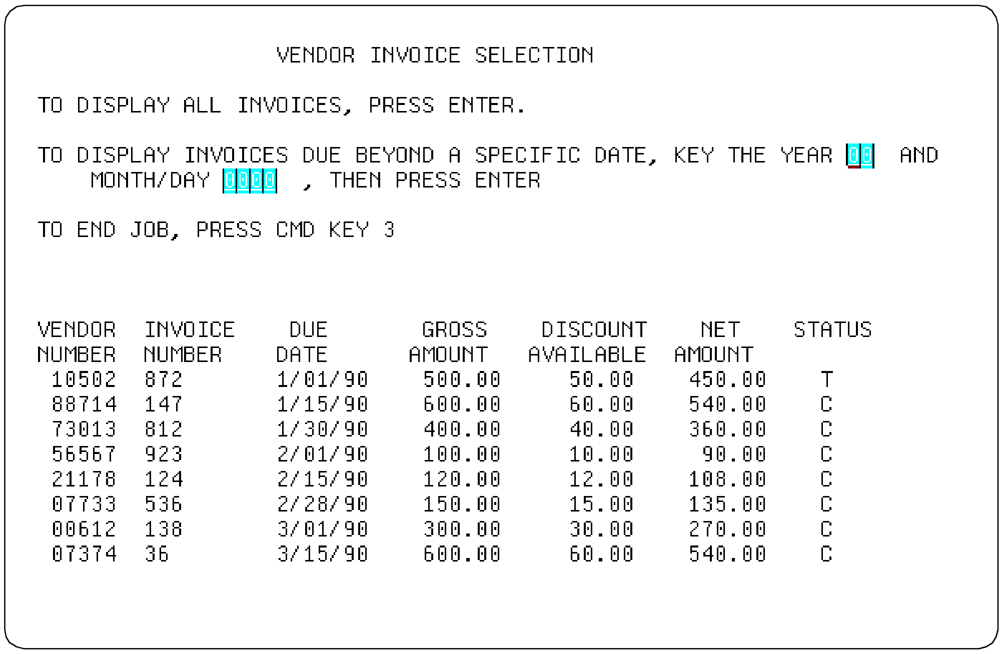
Select record(s) to convert.

```
S DATA
X CONTROL
```

Press 'Next' to customize the generated classes.

Figure 267. Convert DDS SmartGuide

Figure 268 on page 321 shows how the DSP003 file appears when used in a 5250 application running on an AS/400 system.



```
                  VENDOR INVOICE SELECTION

TO DISPLAY ALL INVOICES, PRESS ENTER.

TO DISPLAY INVOICES DUE BEYOND A SPECIFIC DATE, KEY THE YEAR |00|  AND
    MONTH/DAY |0000|  , THEN PRESS ENTER

TO END JOB, PRESS CMD KEY 3



VENDOR  INVOICE    DUE       GROSS    DISCOUNT    NET    STATUS
NUMBER  NUMBER     DATE      AMOUNT   AVAILABLE  AMOUNT
 10502  872        1/01/90    500.00     50.00    450.00    T
 88714  147        1/15/90    600.00     60.00    540.00    C
 73013  812        1/30/90    400.00     40.00    360.00    C
 56567  923        2/01/90    100.00     10.00     90.00    C
 21178  124        2/15/90    120.00     12.00    108.00    C
 07733  536        2/28/90    150.00     15.00    135.00    C
 00612  138        3/01/90    300.00     30.00    270.00    C
 07374  36         3/15/90    600.00     60.00    540.00    C
```

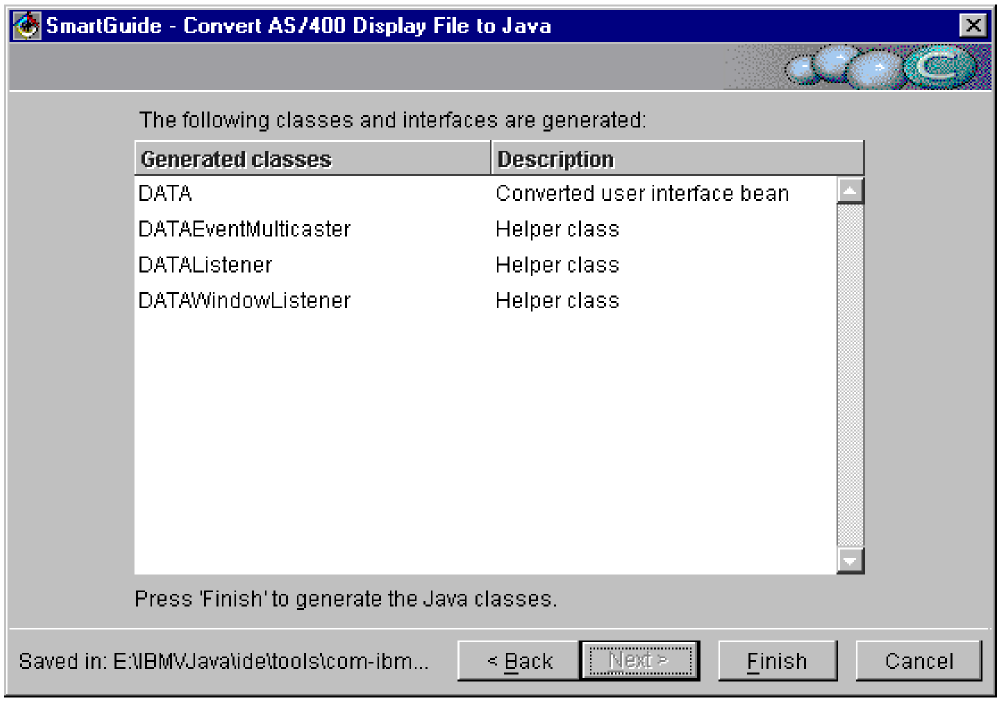
Figure 268. DSP003 File in a 5250 Application

Figure 269 on page 322 shows the source for the DSP03 display file. We use it to create a Java AWT class that is comparable. Notice that a subfile is used as part of this display file. Since the Java AWT does not support subfiles, special classes are built for it and any other AS/400 unique components.

```
A                                        REF(APFLDREF)
A          R DATA                        SFL
A            VNDNBR    R          15  4
A            INVNBR    R          15 11
A            DUDATE    R          15 20EDTCDE(Y)
A            MERCH     R          15 30EDTCDE(3)
A            DCTAVL    R          15 42EDTCDE(3)
A            NET       R          15 50EDTCDE(3)
A            STATUS    R          15 62
A          R CONTROL                     SFLCTL(DATA)
A  40                                    SFLEND
A  75                                    SFLCLR
A  85                                    SFLDSPCTL
A  95                                    SFLDSP
A                                        SFLSIZ(20)
A                                        SFLPAG(8)
A                                        CA03(03 'END OF JOB')
A                                     02 21'VENDOR INVOICE SELECTION'
A                                     04 03'TO DISPLAY ALL INVOICES,'
A                                     04 28'PRESS ENTER.'
A                                     06 03'TO DISPLAY INVOICES DUE BEYOND A
A                                     06 36'SPECIFIC DATE, KEY THE YEAR
A            YR        R     Y  B 06 64DSPATR(CS RI)
A                                        REFFLD(DUYR)
A                                     06 68'AND'
A                                     07 07'MONTH/DAY'
A            MODY      R     Y  B 07 17DSPATR(CS RI)
A                                        REFFLD(DUMODY)
A                                     07 23', THEN PRESS ENTER'
A                                     09  3'TO END JOB, PRESS CMD KEY 3
A  96                                 12 02'NO RECORDS FOUND'
A                                        DSPATR(RI)
A                                     13  3'VENDOR  INVOICE    DUE'
A                                     13 32'GROSS    DISCOUNT    NET'
A                                     13 60'STATUS'
A                                     14  3'NUMBER  NUMBER    DATE'
A                                     14 31'AMOUNT   AVAILABLE  AMOUNT
```

Figure 269. RPG03 Display File Source

As shown in Figure 270 on page 323, the SmartGuide generates a Java class for the DDS file and several helper classes.



SmartGuide - Convert AS/400 Display File to Java

The following classes and interfaces are generated:

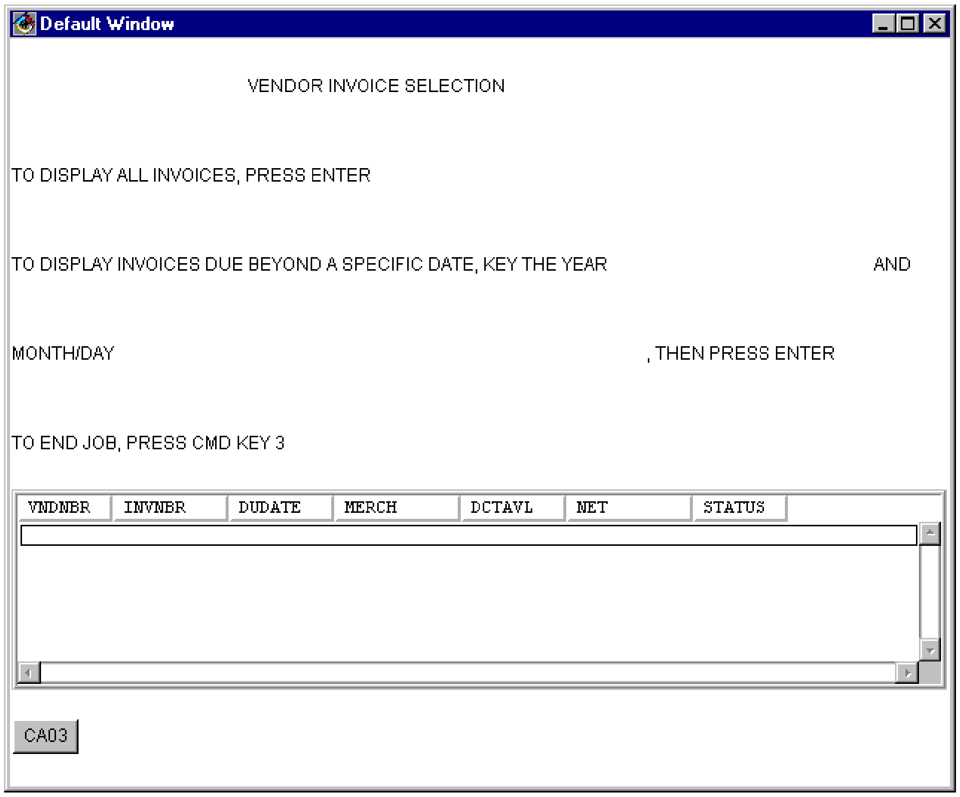
Generated classes	Description
DATA	Converted user interface bean
DATAEventMulticaster	Helper class
DATAListener	Helper class
DATAWindowListener	Helper class

Press 'Finish' to generate the Java classes.

Saved in: E:\IBMVJava\ide\tools\com-ibm...

Figure 270. DDS Conversion SmartGuide

Figure 271 shows how the converted output appears. It can be used as part of an application or as a visual component in a visual builder such as the VisualAge for Java IDE.



Default Window

VENDOR INVOICE SELECTION

TO DISPLAY ALL INVOICES, PRESS ENTER

TO DISPLAY INVOICES DUE BEYOND A SPECIFIC DATE, KEY THE YEAR AND

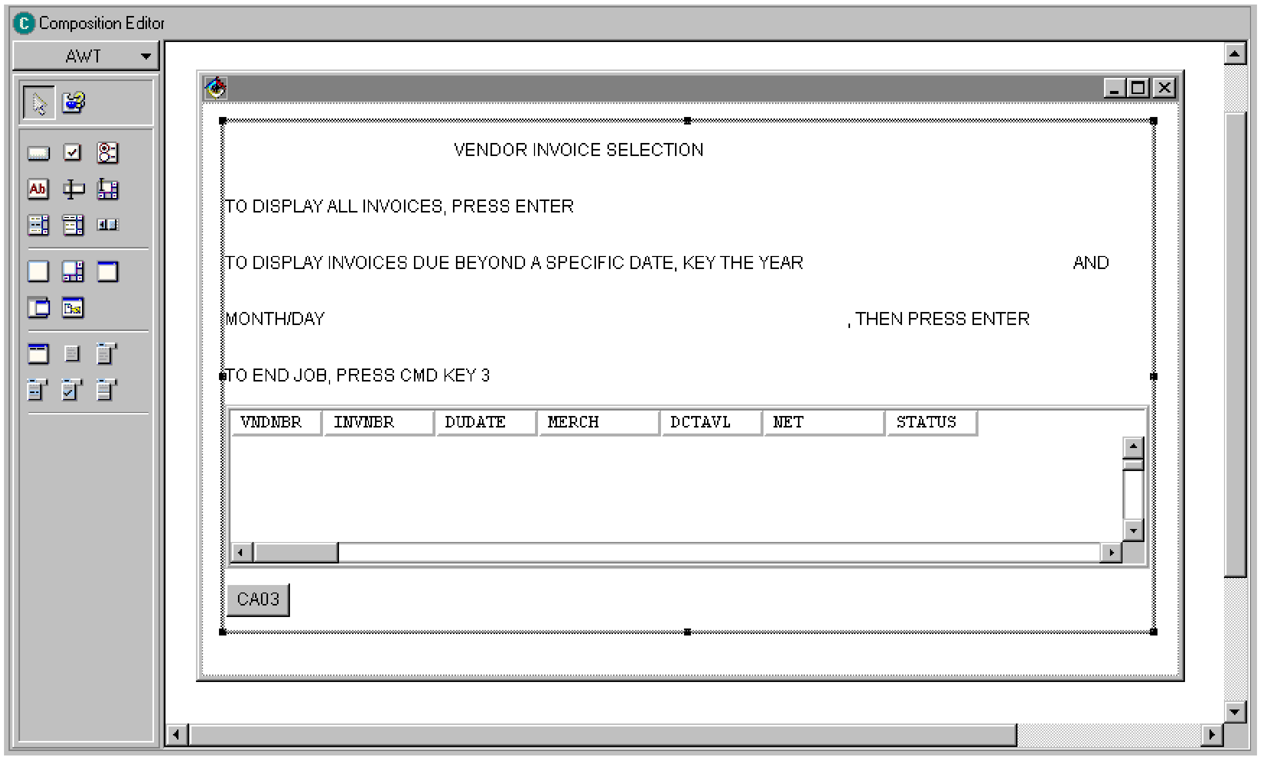
MONTH/DAY , THEN PRESS ENTER

TO END JOB, PRESS CMD KEY 3

VNDNBR	INVNBR	DUDATE	MERCH	DCTAVL	NET	STATUS

Figure 271. Converted DDS File

Figure 272 shows using the new bean in the Visual Composition Editor. You can choose to add a new Bean and then drop it on the frame with which you want to work.



VENDOR INVOICE SELECTION

TO DISPLAY ALL INVOICES, PRESS ENTER

TO DISPLAY INVOICES DUE BEYOND A SPECIFIC DATE, KEY THE YEAR AND

MONTH/DAY , THEN PRESS ENTER

TO END JOB, PRESS CMD KEY 3

VNDNBR	INVNBR	DUDATE	MERCH	DCTAVL	NET	STATUS

Figure 272. Java AWT Display in the VCE

The convert DDS SmartGuide allows you to quickly convert DDS source files to the Java AWT format. The resulting JavaBean can be used as part of a Java application. The SmartGuide uses AWT, where possible. If not, it uses the "com.ibm.ivj.et400.util" package to deal with the rest of the fields. For example, a subfile extends com.ibm.ivj.et400.util.AS400VisualSubfile.

If you need to change the bean, for example, improve the appearance of the display, then make the changes in the Java source. The Visual Composition editor does not work here.

8.5 Creating a Subfile SmartGuide

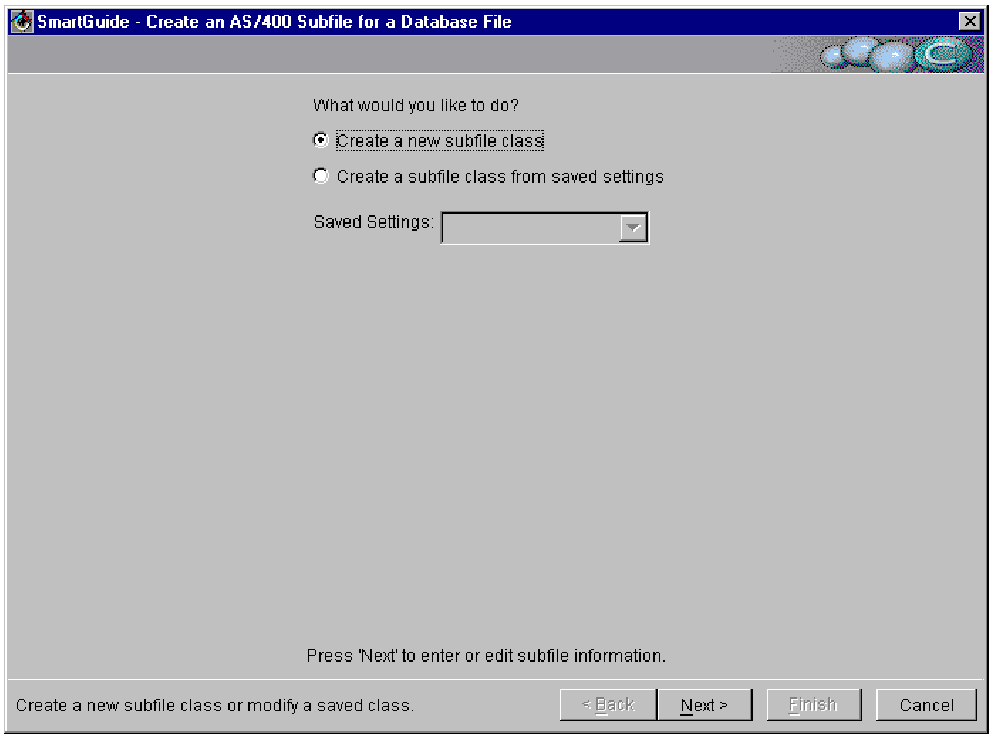
This SmartGuide creates a subfile from an AS/400 database file. The following limitations apply:

- Subfiles currently do not support:
  - Floating point fields
  - Duplicate keys
  - Partial keys
- AS/400 Record I/O maximum limit is 500 fields per record format. If the file has more than 500 fields, the SmartGuide fails to generate the class.
- Although the AS/400 Record I/O maximum size of a field can be 32766 bytes, the SmartGuide can only handle 20000 bytes.

- The Record I/O classes do not support logical join files and null key fields. Also the record level access from a Java Applet is allowed only in V4R2 and later. In V3R2, V3R7, and V4R1, only Java Application support is provided.
- The Subfile SmartGuide allows you to select only one record format. It supports *first member in the data file, even though the file can have multiple record formats or multiple members.
- The number of decimal places set in each numeric field of the subfile has to match with the decimal position defined in the physical file. Otherwise, the data is incorrect.
- To run the application generated by subfile SmartGuide, set the CLASSPATH using the **Selected—>Properties—>Class path** tab. Then, select **IBM Enterprise Toolkit for AS/400** and the JFC class libraries.
- Record numbering of a subfile is different from the record numbering of a database file.

8.5.1 Creating a Java Subfile Bean

This section demonstrates how to create a Java Subfile class (or bean) from an AS/400 database file. To start the Create Subfile SmartGuide, select **Create Subfile** from the ET/400 menu. The Create an AS/400 Subfile SmartGuide initial display appears as shown in Figure 273.


SmartGuide - Create an AS/400 Subfile for a Database File

What would you like to do?

(●) Create a new subfile class

( ) Create a subfile class from saved settings

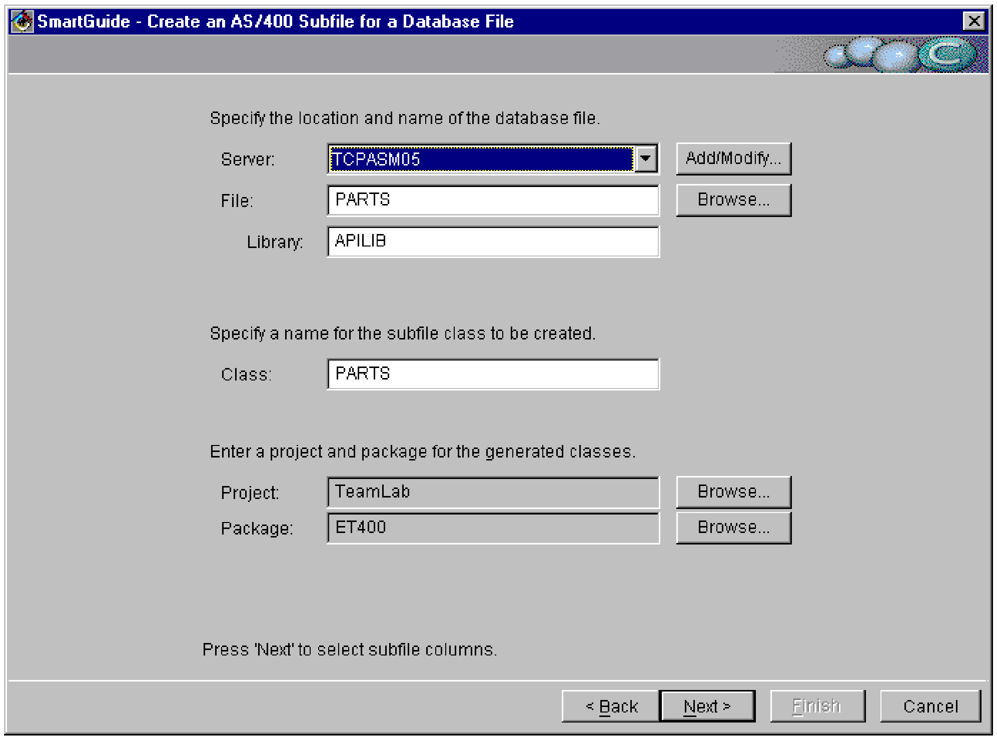
Saved Settings:

Press 'Next' to enter or edit subfile information.

Create a new subfile class or modify a saved class.

Figure 273. Create an AS/400 Subfile SmartGuide

The first Subfile SmartGuide display allows you to choose between create a new subfile or work with an exiting one. We choose the *Create a new subfile class*. This option shows a display, as shown in Figure 274 on page 326, that allows us to specify which AS/400 system to access and which database file to use as the base for creating a Java subfile class.


SmartGuide - Create an AS/400 Subfile for a Database File

Specify the location and name of the database file.

Server: TCPASM05

File: PARTS

Library: APILIB

Specify a name for the subfile class to be created.

Class: PARTS

Enter a project and package for the generated classes.

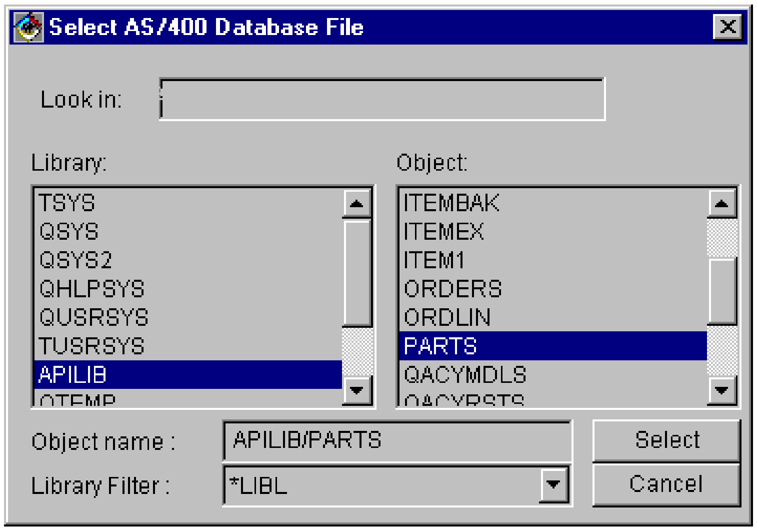
Project: TeamLab

Package: ET400

Press 'Next' to select subfile columns.

Figure 274. Create Subfile SmartGuide

If the Browse button is clicked, the SmartGuide retrieves a list of AS/400 libraries based on your library list. If you select a library from the list, the SmartGuide displays a list of objects from that library that are candidates for creating a subfile from. In Figure 275, we select the PARTS file from APILIB.


Select AS/400 Database File

Look in:

Library: TSYS, QSYS, QSYS2, QHLPSYS, QUSRSYS, TUSRSYS, APILIB, QTEMP

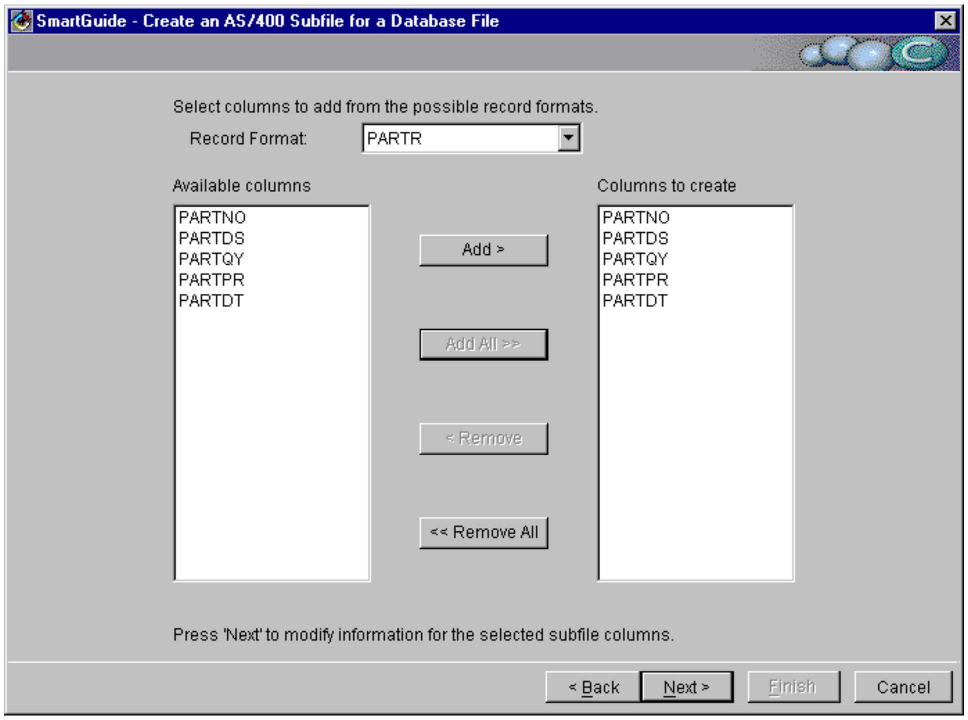
Object: ITEMBAK, ITEMEX, ITEM1, ORDERS, ORDLIN, PARTS, QACYMDLS, QACYRSTS

Object name : APILIB/PARTS

Library Filter : *LIBL

Figure 275. Select AS/400 Database File

In Figure 276 on page 327, we select the columns from the PARTS file that we want to display in the subfile JavaBean that we generate.



SmartGuide - Create an AS/400 Subfile for a Database File

Select columns to add from the possible record formats.

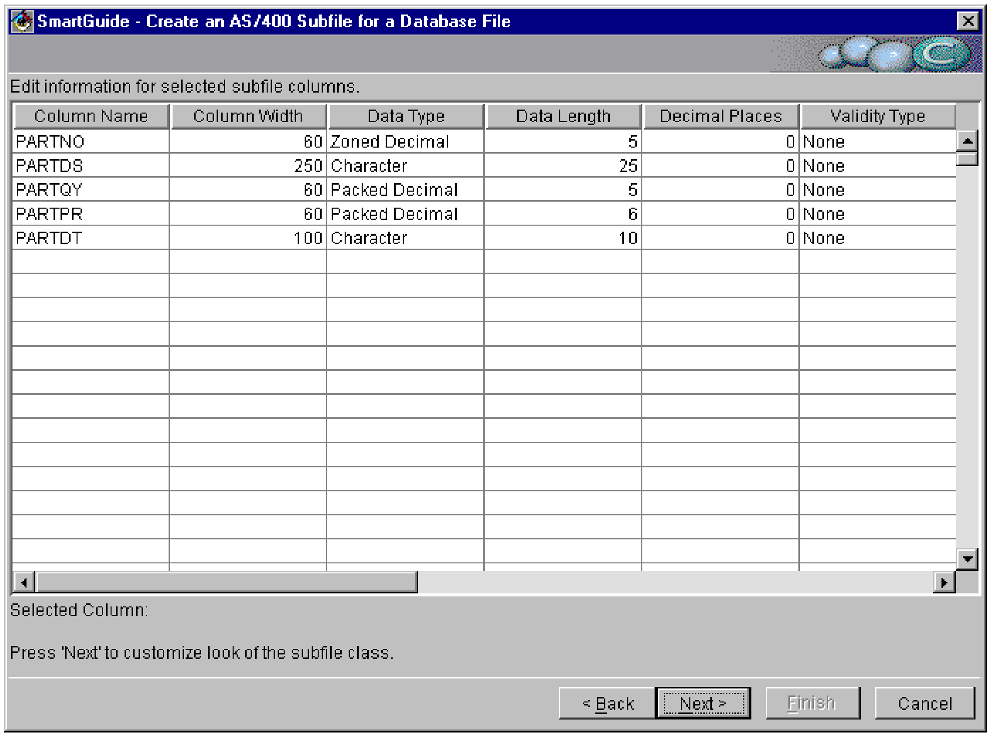
Record Format: PARTR

Available columns	Columns to create
PARTNO	PARTNO
PARTDS	PARTDS
PARTQY	PARTQY
PARTPR	PARTPR
PARTDT	PARTDT

Press 'Next' to modify information for the selected subfile columns.

Figure 276. Select Subfile Columns

The SmartGuide displays the selected columns and the column attributes as shown in Figure 277. At this point, we can change the format of the columns.



SmartGuide - Create an AS/400 Subfile for a Database File

Edit information for selected subfile columns.

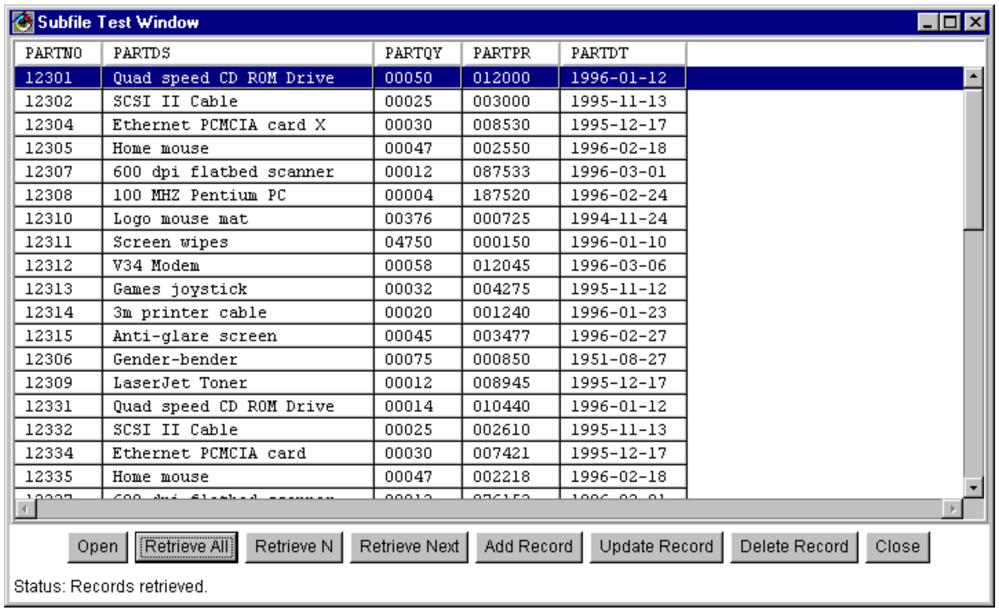
Column Name	Column Width	Data Type	Data Length	Decimal Places	Validity Type
PARTNO	60	Zoned Decimal	5	0	None
PARTDS	250	Character	25	0	None
PARTQY	60	Packed Decimal	5	0	None
PARTPR	60	Packed Decimal	6	0	None
PARTDT	100	Character	10	0	None

Selected Column:

Press 'Next' to customize look of the subfile class.

Figure 277. Edit Subfile Columns

When we are satisfied with the format of the subfile, we generate the subfile JavaBean. The SmartGuide generates a subfile JavaBean and a ready-to-use application as shown in Figure 278.



PARTNO	PARTDS	PARTQY	PARTPR	PARTDT
12301	Quad speed CD ROM Drive	00050	012000	1996-01-12
12302	SCSI II Cable	00025	003000	1995-11-13
12304	Ethernet PCMCIA card X	00030	008530	1995-12-17
12305	Home mouse	00047	002550	1996-02-18
12307	600 dpi flatbed scanner	00012	087533	1996-03-01
12308	100 MHZ Pentium PC	00004	187520	1996-02-24
12310	Logo mouse mat	00376	000725	1994-11-24
12311	Screen wipes	04750	000150	1996-01-10
12312	V34 Modem	00058	012045	1996-03-06
12313	Games joystick	00032	004275	1995-11-12
12314	3m printer cable	00020	001240	1996-01-23
12315	Anti-glare screen	00045	003477	1996-02-27
12306	Gender-bender	00075	000850	1951-08-27
12309	LaserJet Toner	00012	008945	1995-12-17
12331	Quad speed CD ROM Drive	00014	010440	1996-01-12
12332	SCSI II Cable	00025	002610	1995-11-13
12334	Ethernet PCMCIA card	00030	007421	1995-12-17
12335	Home mouse	00047	002218	1996-02-18

Status: Records retrieved.

Figure 278. Generated Subfile Application

The generated subfile JavaBean can be used as part of a custom application. Figure 279 shows how to use the subfile bean in the VisualAge for Java Visual Composition Editor.

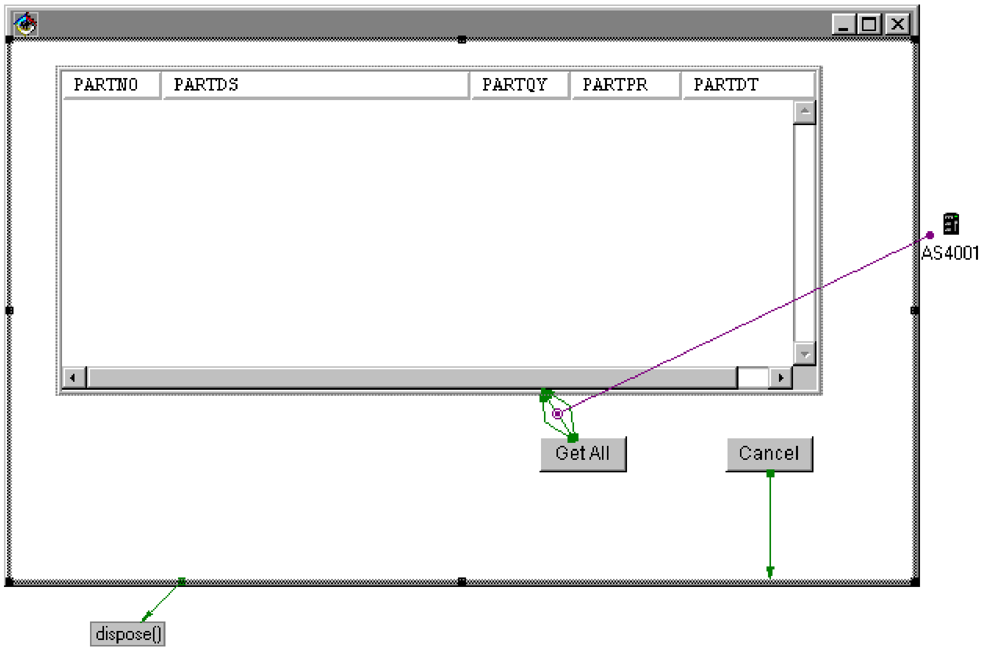
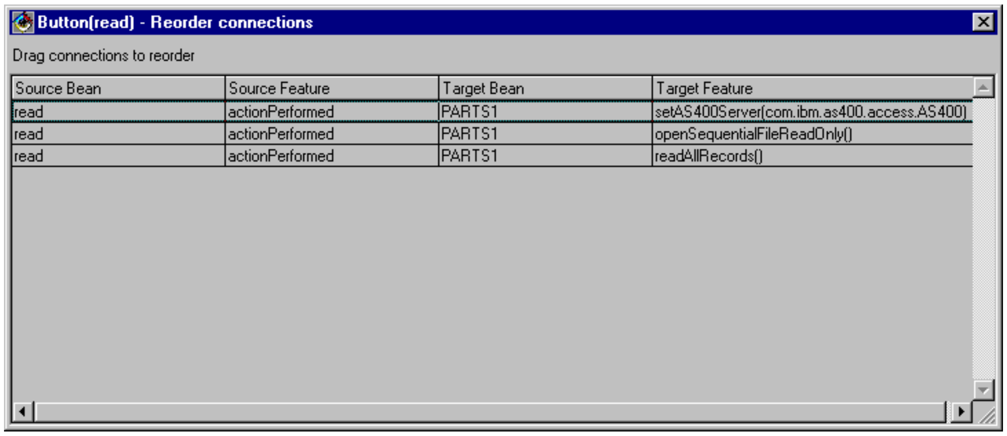


Figure 279. Subfile Application in the VCE

In this application, we use an AS400 object from the AS/400 Toolbox for Java to provide a connection to the AS/400 system. We use the subfile JavaBean as a visual component, which we drop on the frame. We use the actions of the Get All button to control the application processing. In Figure 280, we show the connections to the subfile JavaBean. We use the subfile setAS400Server method to set the AS/400 connection, using the AS400 object as input. We use the openSequentialFileReadOnly method to open the AS/400 file and the readAllRecords method to populate the subfile.



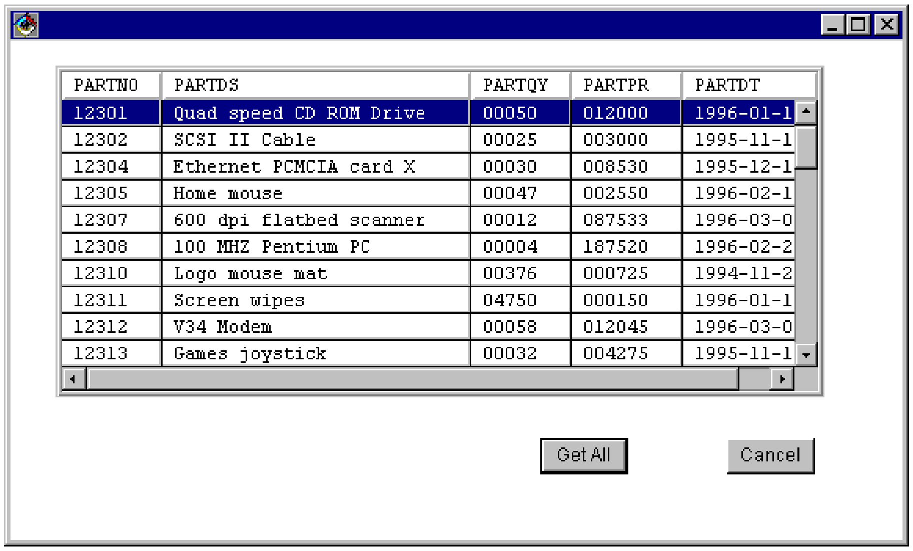
Button(read) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
read	actionPerformed	PARTS1	setAS400Server(com.ibm.as400.access.AS400)
read	actionPerformed	PARTS1	openSequentialFileReadOnly()
read	actionPerformed	PARTS1	readAllRecords()

Figure 280. Get All Button Connections

Figure 281 shows the subfile populated with information from the AS/400 PARTS file.



PARTNO	PARTDS	PARTQY	PARTPR	PARTDT
12301	Quad speed CD ROM Drive	00050	012000	1996-01-1
12302	SCSI II Cable	00025	003000	1995-11-1
12304	Ethernet PCMCIA card X	00030	008530	1995-12-1
12305	Home mouse	00047	002550	1996-02-1
12307	600 dpi flatbed scanner	00012	087533	1996-03-0
12308	100 MHZ Pentium PC	00004	187520	1996-02-2
12310	Logo mouse mat	00376	000725	1994-11-2
12311	Screen wipes	04750	000150	1996-01-1
12312	V34 Modem	00058	012045	1996-03-0
12313	Games joystick	00032	004275	1995-11-1

Figure 281. Completed Subfile Application

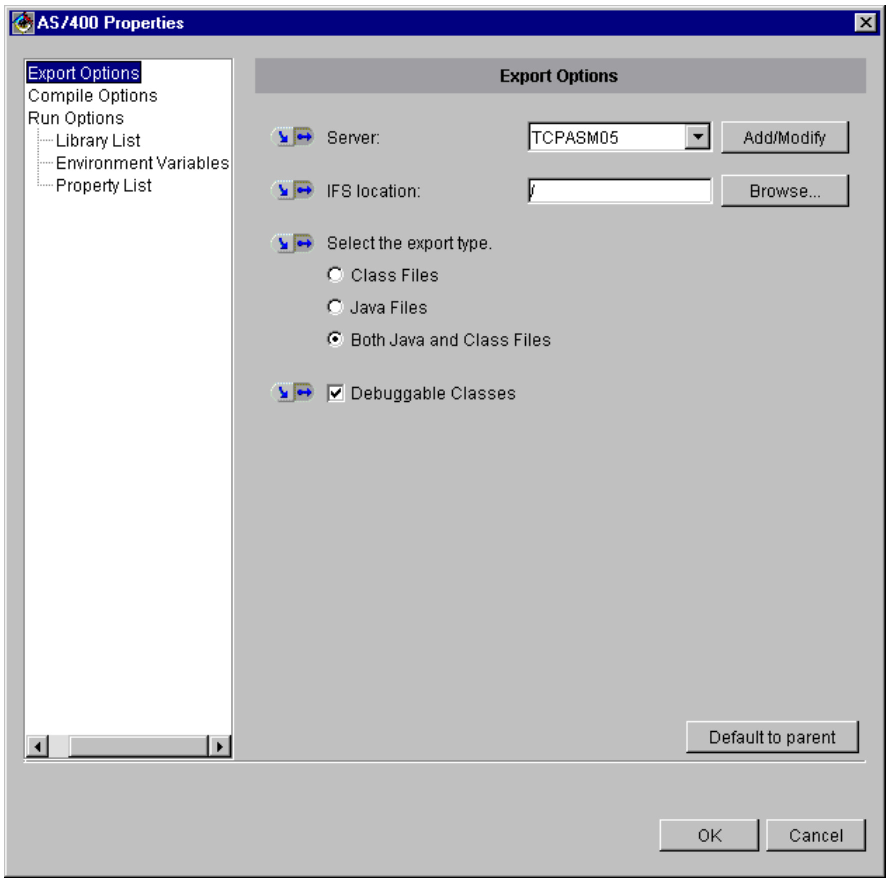
The Create subfile SmartGuide allows us to build a Java version of an AS/400 subfile. We can use an exiting AS/400 database as the basis for the generated subfile. The SmartGuide generates a JavaBean that we can use as a visual component in a visual builder such as the Visual Composition Editor. It also generates a finished application, which we can use as is.

8.6 Support for Export, Compile, Run, and Debug AS/400 Programs

This support allows you to create, run, and debug Java programs on an AS/400 system from the VisualAge for Java IDE. It allows you to customize and save options.

8.6.1 Setup

Before using ET/400 support to export, compile, or debug AS/400 Java programs, select the **Properties** option from the ET/400 menu and set up the options that you want to use. The AS/400 Properties menu is shown in Figure 282.



Server: TCPASM05

IFS location: /

Select the export type.

( ) Class Files

( ) Java Files

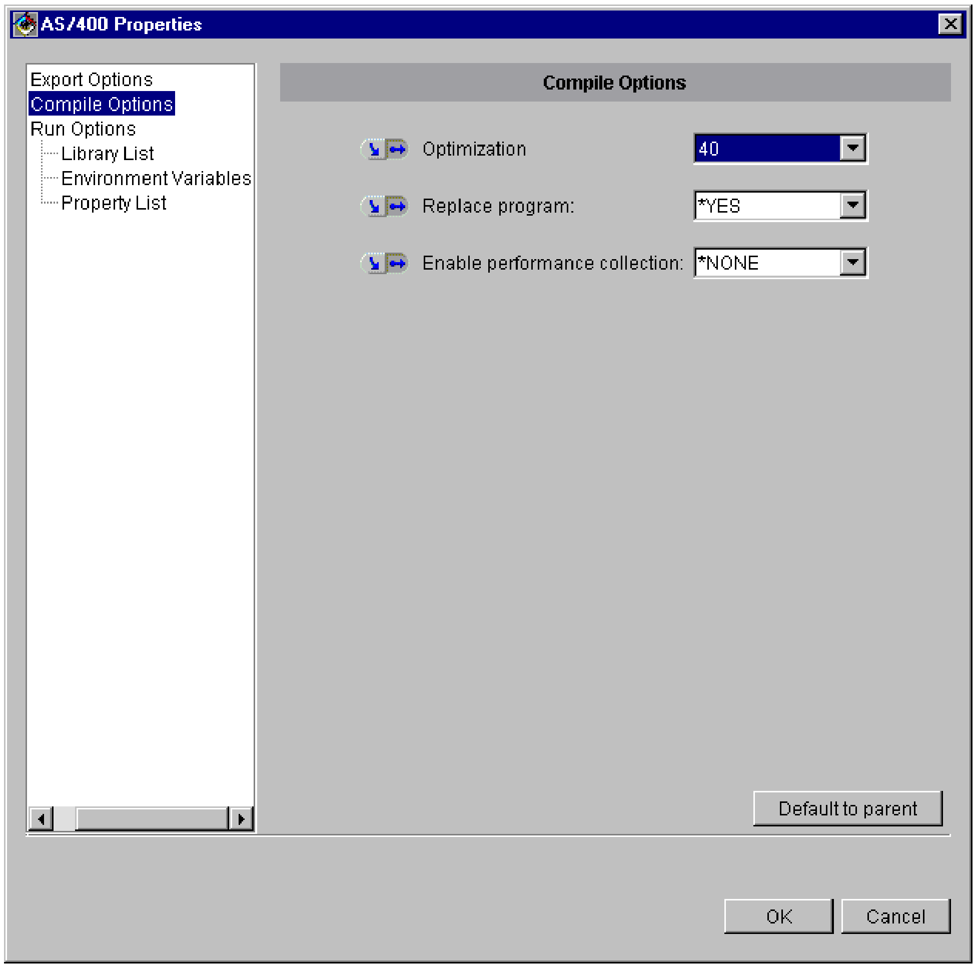
(●) Both Java and Class Files

☑ Debuggable Classes

Figure 282. AS/400 Properties

Export Options allows you to control the name of the AS/400 system that you are exporting to and the directory in the AS/400 integrated file system to which you want to export.

Compile options, as shown in Figure 283 on page 331, allows you to set the optimization level, whether to replace existing programs or to enable the collection of performance information.


AS/400 Properties

Compile Options

Optimization: 40

Replace program: *YES

Enable performance collection: *NONE

Figure 283. Compile Options

8.6.2 Export Support

Export support helps you export Java files from the VisualAge for Java Integrated Development Environment to the AS/400 integrated file system (IFS). If you have a network drive assigned to the AS/400 system, you can export directly from VisualAge for Java to the IFS. Using the ET/400 export support, you do not need a network drive. To use the ET/400 support, highlight the classes that you want to export, and select Export from the ET/400 menu. You can export multiple classes at once.

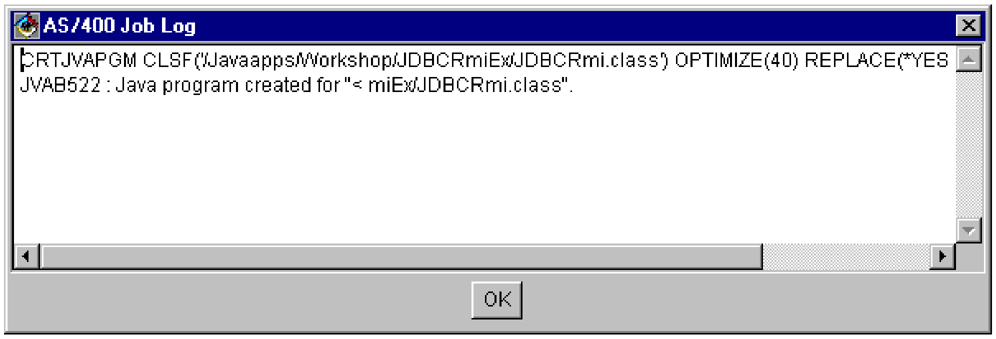
8.6.3 Compile Support

To compile a program on the AS/400 system, highlight the class file. You can have multiple classes highlighted. Select **Compile** from the ET/400 menu. The compile AS/400 Java class files support creates an AS/400 Java program from a Java class file. The resulting Java program object becomes part of the class file object, but cannot be viewed or modified directly. The Java class file name must be in one of the following AS/400 integrated file systems: QOpenSys, "root", or a user-defined file system. Behind the scenes, this support calls the Create Java Program (CRTJVAPGM) command on the AS/400 system and returns messages back to the user. The CRTJVAPGM process is done the first time you run a Java program on the AS/400 system. If you have large programs that take a long time to compile, you may want to use this support so the program is created before you run it. The only extension of the CRTJVAPGM command provided is the ability to save the compile definitions locally, so you do not have to specify them the next time you compile the same class. For example, you may want to save the optimization level with which the AS/400 Java program should be compiled.

The following optimization levels are valid:

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

The output from the compile is returned to you in the VisualAge IDE in a dialog box as shown in Figure 284.



```
CRTJVAPGM CLSF('/Javaapps/Workshop/JDBCRmiEx/JDBCRmi.class') OPTIMIZE(40) REPLACE(*YES
JVAB522 : Java program created for "< miEx/JDBCRmi.class".
```

Figure 284. ET/400 Compile Dialog Box

8.6.4 Debug Support

The cooperative debugger allows you to debug a Java program running on the AS/400 system from the VisualAge for Java IDE running on a workstation. Before you can use the debugger, the debug server must be started on the AS/400 system. To start the debug server, enter the Start Debug Server (STRDBGSVR) command on an AS/400 command line and press **Enter**.

Attention

The debug server needs to be started only once for the AS/400 system on which you plan to debug your application.

If the debug server has already been started previously, you see the message `Debug server router function already active`, when you issue the STRDBGSVR command.

To debug an AS/400 Java program, you need to compile it using the -g option. If you use ET/400 support to compile the program, check the Debuggable classes option as shown in Figure 282 on page 330.

If you compiled the program and used a higher optimization level higher than 10, recompile with level 10 for best results. Use:

```
CRTJVAPGM CLSF(xxxxxx) OPTIMIZE(10)
```

8.6.5 Debugging an AS/400 Java Program

This section shows how to debug a Java program running on the AS/400 system from within the VisualAge for Java IDE. The program that we debug is the RMI example that we created in Section 7.3, “Building a Simple AS/400 Application Using RMI” on page 291. To start the remote debugger, we highlight the program that we want to debug and select **Debug** from the ET/400 menu. The Debugger Logon prompt shown in Figure 285 appears.

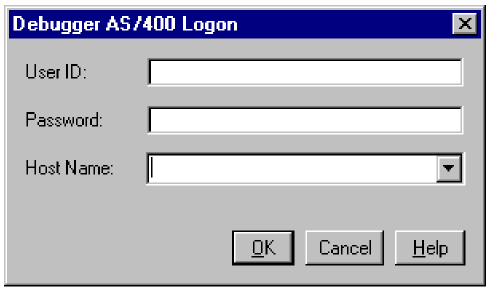
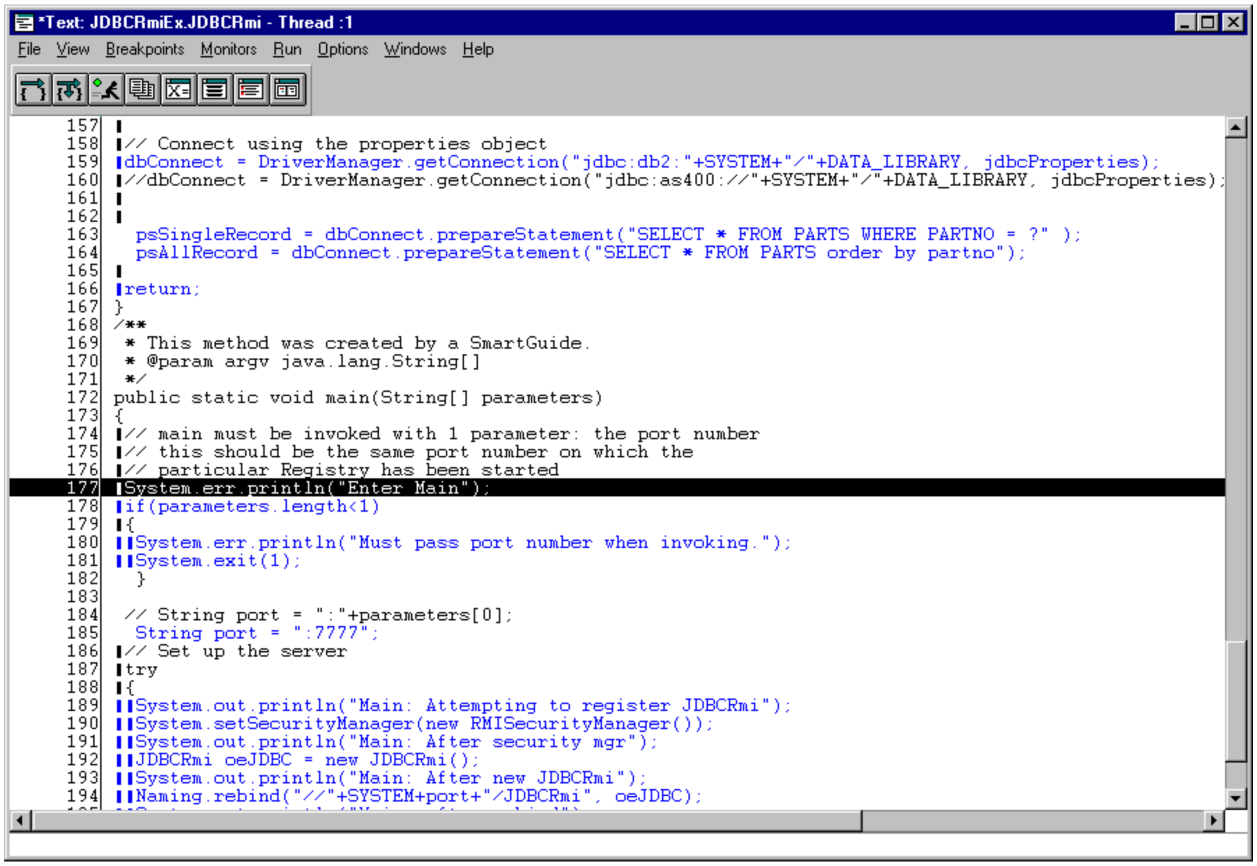


Figure 285. Debugger AS/400 Logon

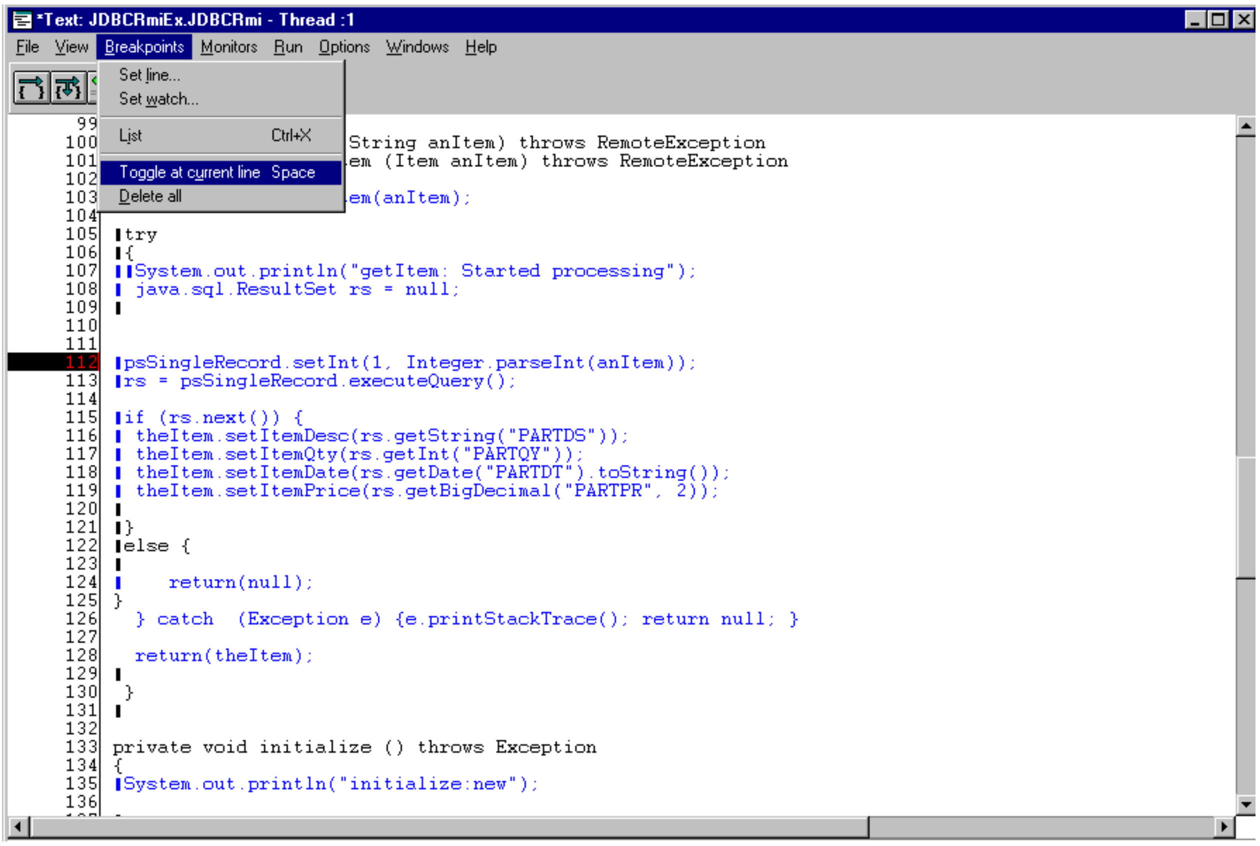
After signing on the AS/400 system, the debugger starts running the program on the AS/400 system. We see the source Java code for the program as shown in Figure 286 on page 334.


*Text: JDBCRmiEx.JDBCRmi - Thread :1
File View Breakpoints Monitors Run Options Windows Help

```

// Connect using the properties object
dbConnect = DriverManager.getConnection("jdbc:db2:"+SYSTEM+"/"+DATA_LIBRARY, jdbcProperties);
//dbConnect = DriverManager.getConnection("jdbc:as400://"+SYSTEM+"/"+DATA_LIBRARY, jdbcProperties);


  psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?" );
  psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS order by partno");

return;
}
/**
 * This method was created by a SmartGuide.
 * @param argv java.lang.String[]
 */
public static void main(String[] parameters)
{
// main must be invoked with 1 parameter: the port number
// this should be the same port number on which the
// particular Registry has been started
System.err.println("Enter Main");
if(parameters.length<1)
{
System.err.println("Must pass port number when invoking.");
System.exit(1);
  }

 // String port = ":"+parameters[0];
  String port = ":7777";
// Set up the server
try
{
System.out.println("Main: Attempting to register JDBCRmi");
System.setSecurityManager(new RMISecurityManager());
System.out.println("Main: After security mgr");
JDBCRmi oeJDBC = new JDBCRmi();
System.out.println("Main: After new JDBCRmi");
Naming.rebind("//"+SYSTEM+port+"/JDBCRmi", oeJDBC);
```

Figure 286. JDBCRmi Java Source

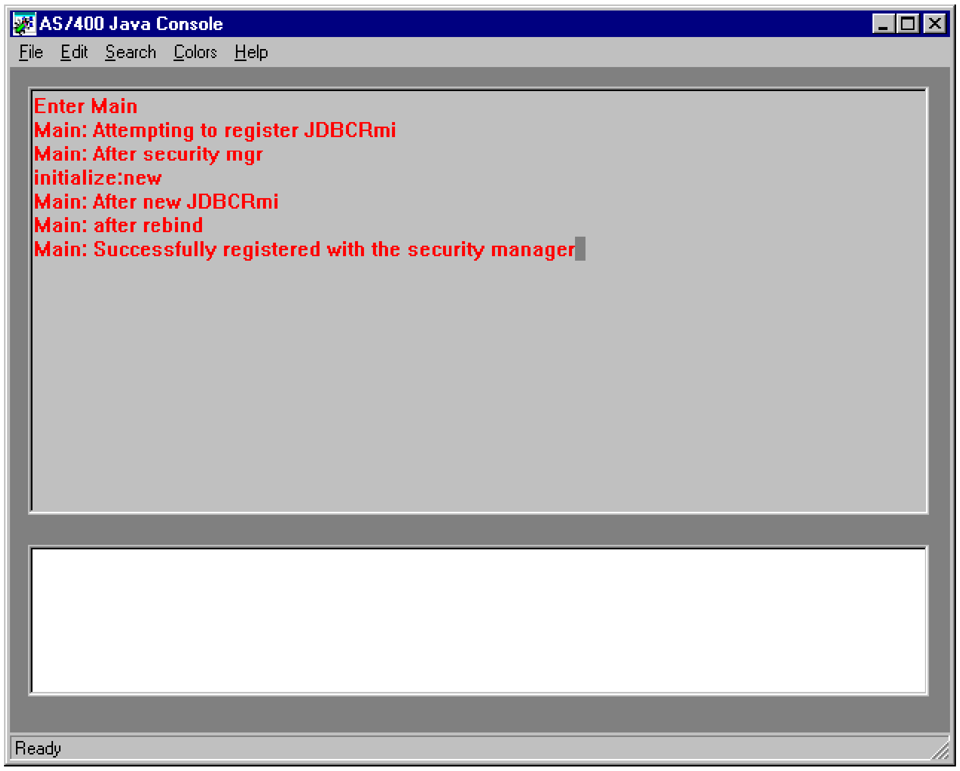
We can now set breakpoints or watches for variables. Figure 287 on page 335 shows how to set a breakpoint at line 112 of the JDBCRmi program.



```
                               String anItem) throws RemoteException
                               em (Item anItem) throws RemoteException

                               em(anItem);

try
{
 System.out.println("getItem: Started processing");
 java.sql.ResultSet rs = null;



psSingleRecord.setInt(1, Integer.parseInt(anItem));
rs = psSingleRecord.executeQuery();

if (rs.next()) {
 theItem.setItemDesc(rs.getString("PARTDS"));
 theItem.setItemQty(rs.getInt("PARTQY"));
 theItem.setItemDate(rs.getDate("PARTDT").toString());
 theItem.setItemPrice(rs.getBigDecimal("PARTPR", 2));

}
else {

    return(null);
}
  } catch  (Exception e) {e.printStackTrace(); return null; }

  return(theItem);

 }


private void initialize () throws Exception
{
System.out.println("initialize:new");
```

Figure 287. Setting a Breakpoint

Since this is a client/server application that gets its input from the client program named RMIExample, we start it on the client and connect to the AS/400 system. Figure 288 on page 336 shows the Java console for the program running on the AS/400 system.



```
Enter Main
Main: Attempting to register JDBCRmi
Main: After security mgr
initialize:new
Main: After new JDBCRmi
Main: after rebind
Main: Successfully registered with the security manager
```

Figure 288. Debugger Java Console

We can now use the debugger to display and work with the threads of the JDBCRmi program, as shown in Figure 289.

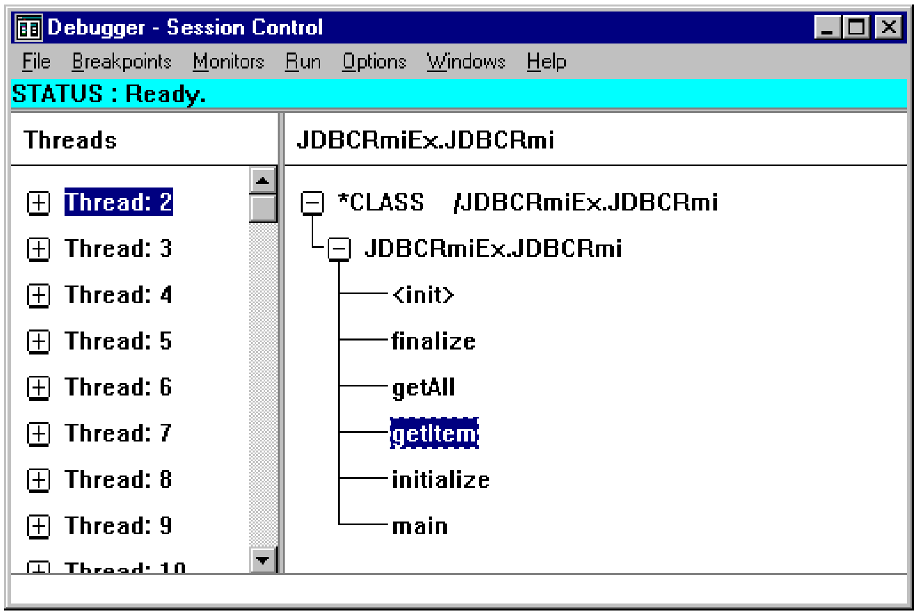
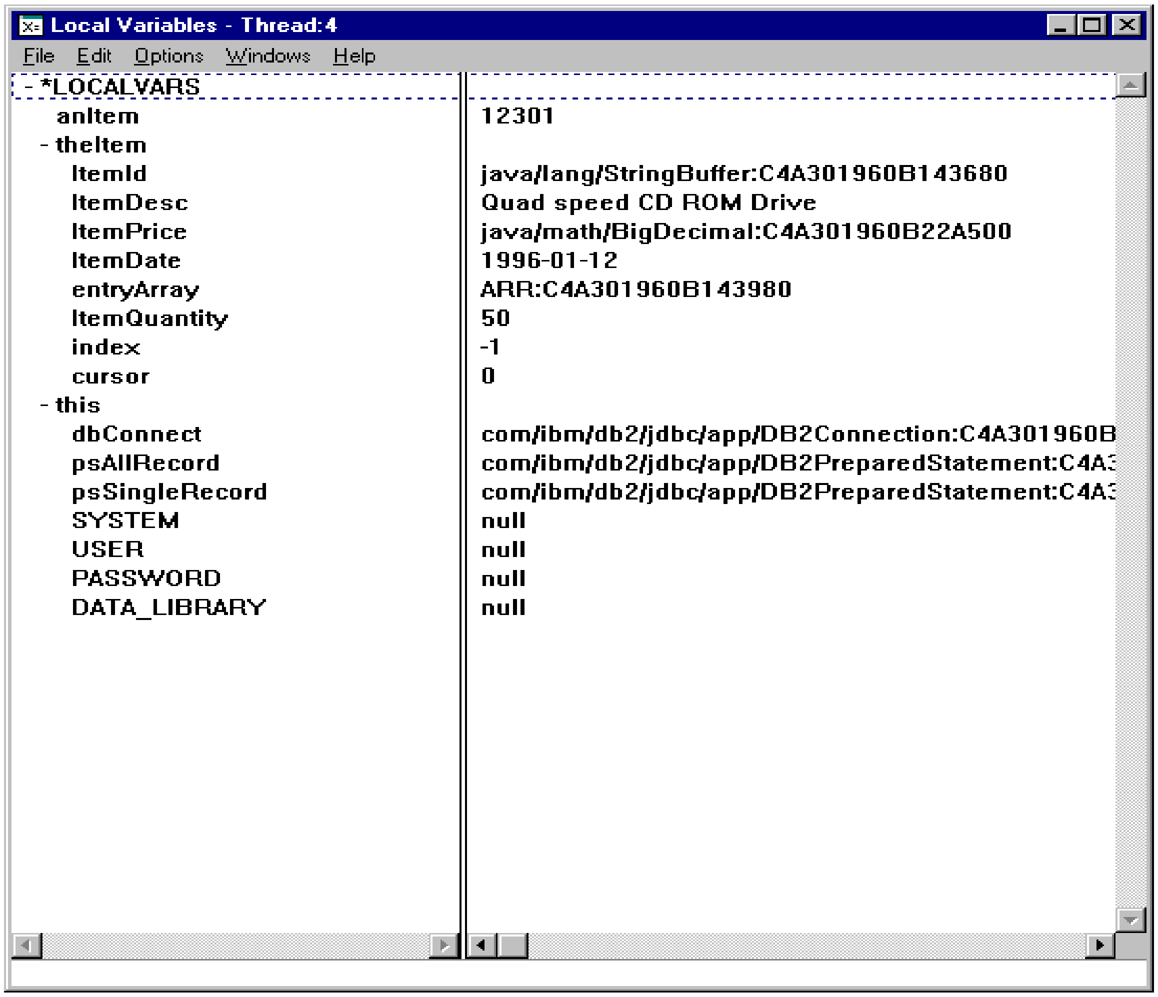


Figure 289. JDBCRmi Threads

If we request a part from the database, the host program stops at line 112 because we set a breakpoint there. We can now step through the code and look at the variables of JDBCRmi. In Figure 290, we can see the variables as they are displayed by the debugger.



```
- *LOCALVARS
    anItem              12301
  - theItem
      ItemId            java/lang/StringBuffer:C4A301960B143680
      ItemDesc          Quad speed CD ROM Drive
      ItemPrice         java/math/BigDecimal:C4A301960B22A500
      ItemDate          1996-01-12
      entryArray        ARR:C4A301960B143980
      ItemQuantity      50
      index             -1
      cursor            0
  - this
      dbConnect         com/ibm/db2/jdbc/app/DB2Connection:C4A301960B
      psAllRecord       com/ibm/db2/jdbc/app/DB2PreparedStatement:C4A3
      psSingleRecord    com/ibm/db2/jdbc/app/DB2PreparedStatement:C4A3
      SYSTEM            null
      USER              null
      PASSWORD          null
      DATA_LIBRARY      null
```

Figure 290. Displaying Program Variables

8.7 ET/400 System Requirements

The ET/400 system requires a client platform with:

- VisualAge for Java Enterprise edition
- TCP/IP setup on the workstation

The AS/400 system must be equipped with:

- Application Development Tool Set (ADTS), which is required for the Display File Conversion SmartGuide. ADTS Version 3 Release 2 or Version 3 Release 6 and above is required.
- OS/400 V4R2 (or later) QJAVA library to compile, run, and debug Java on the AS/400 system.

AS/400 connections must be established. By having a JRE (Java Runtime Environment) installed on the workstation system, you can avoid signing onto the same AS/400 more than once. A server is started that keeps a connection to the

AS/400 active, so that it may be used by other ET/400 components. This server remains active for six hours of inactivity, at which point it shuts down. You can also end this server by going to the Task Manager and ending the jre.exe process.

8.8 PTF Information

Load the latest cumulative PTF package for the release that you are using on your AS/400 system.

The AS/400 Cooperative Debugger requires that the following PTFs be applied:

• V4R3M0	5769SS100	SF49975
• V4R3M0	5769999	MF19487

If you are using the convert DDS SmartGuide, you need to apply the appropriate PTF as indicated in the following list for the Application Development ToolSet/400 product:

• V3R2M0	5763PW100	SF45556
• V3R6M0	5716PW100	SF45554
• V3R7M0	5716PW100	SF45552
• V4R2M0	5769PW100	SF49832
• V4R3M0	5769PW100	SF49832

The convert DDS SmartGuide support for edit codes has been improved. To take advantage of this, you need the following PTFs:

• V3R1M0	5763PW100	SF47783
• V3R2M0	5763PW100	SF47784
• V3R6M0	5716PW100	SF47785
• V3R7M0	5716PW100	SF47786
• V4R2M0	5769PW100	SF47774
• V4R3M0	5769PW100	SF47774

Chapter 9. JavaBeans

This chapter is designed to help you understand what a JavaBean (also referred to as a bean) is and how they are created. First, it discusses what a JavaBean is. You then learn what makes a good JavaBean and when to use them. Finally, this chapter explains how to use the Enterprise Toolkit/400 to create a JavaBean, which allows you to run an AS/400 program from a Java client. Java code examples are discussed throughout this chapter, followed by several complete working examples that you can compile and try on your own.

The source code for the examples discussed in this chapter is available on the Internet. For download instructions, please refer to Section A.1, “Downloading the Files from the Internet” on page 396.

Before you start, we should answer this question: What do you *need* and what do you *need to know*?

To create and use a JavaBean, you need a Java 1.1 compatible development tool such as Visual Age for Java or the Java Development Kit (JDK 1.1) and a text editor.

This chapter does not teach every detail about JavaBeans, but provides enough information for you to understand and create simple-to-intermediate JavaBeans. You learn what JavaBeans are capable of and what additional information you must gather for your specific needs.

9.1 What JavaBeans Offer

JavaBeans offer several benefits. Probably the largest benefit gained from using beans is the ability to use a bean over and over because it is a component.

How many graphical applications have you used that have buttons? Probably every single one. The same is true for text boxes, scrollbars, and menus. These components are so common and are used in so many applications. The time saved by programmers who can use the standard Java button rather than create their own is unimaginable. This does not only apply to graphical components. A bean can be something as complex as a grammar and spelling checker, and can also be reused because of the large number of word processing applications.

9.1.1 Visual Manipulation and Building

Using a good Java builder tool, it is possible to import and connect several beans together to make a complete application without writing a single line of Java code. Of course, someone has to create the beans to start. Many beans can be bought and reused, and many builder tools actually allow the creation of beans without writing any Java code. Even if your bean is not a graphical one, it can be much faster to draw connection lines and have the builder tool write the Java code and send the correct parameters. Otherwise, you may end up writing everything yourself, only to get syntax or logic errors.

9.1.2 Everything Java Offers and More

Many cautious or skeptical developers want to make sure they are making a good investment before completely jumping into the world of beans. The main point to remember is that JavaBeans are 100% Java and simply an extension of Java. JavaBeans offer everything Java offers and more. The data processing industry has seen incredible growth and support for Java in the past few years, and there seems to be no stopping point yet. As with Java, JavaBeans is not expected to die anytime soon. Because beans are Java, they are relatively simple to program. Beans give you all the benefits of an object-oriented programming model. Beans are also Internet and intranet ready and are perfect for distributed applications. They also inherit all of the built-in capabilities that Java offers.

9.1.3 Easy Packaging and Distribution

Java JAR files make it easy to package the several class files that make up a bean or several beans into one easy-to-ship JAR file. Most builder tools have a wizard that allows the user to import a JAR file and select which beans inside the JAR file to use. Because of introspection and the bean standards, the builder tool can also inform the user of the beans properties, methods, and events without the user having to read any documentation or look through program code to determine a method's parameters.

9.2 The Basics of JavaBeans

The definition of a *JavaBean* is: A reusable software component that can be visually manipulated in a builder tool.

That definition is pretty general because the beans specification itself is quite general and leaves a lot of room for variety and customizing. Therefore, beans can come in a wide variety of shapes and sizes, and perform a number of different tasks and still conform to the JavaBeans specification.

To begin understanding JavaBeans, this section discusses some basic JavaBeans concepts and terminology used. In Section 9.3, “Creating a Simple JavaBean” on page 342, you learn what is done to create a simple JavaBean without any complex BeanInfo knowledge.

9.2.1 What It Actually Means to Be a Bean

Any object can be a bean. Almost any Java object is already a bean or can be quickly changed to follow the beans rules. To be a bean really means that the class follows a few simple rules and naming conventions. There is no class that a bean must extend or interfaces that must be implemented. However, some are offered to help with complex beans.

It is also important to remember that just because the definition says that a bean can be visually manipulated, all beans are not graphical. Many components, such as buttons and textfields, can be beans. Another example of a bean is a text convertor or a credit card number verifier. All of the components we just listed have one task — to be general enough to be used in many applications. Most builder tools allow the user to have an icon that represents the bean while designing the application, which is invisible at runtime.

9.2.1.1 JavaBean Terminology

This section defines basic JavaBean terms to help you gain a better understanding of this topic. These terms include:

Property A property is a piece of information about a particular bean that is used to give, get, or pass information to and from a bean. Suppose we have a bean that represents a person. One property is age. This is an example of a readable property because you can ask for someone's age, but there is not a way to change it. Another property is hair color. This is a readable and writable property because we can change their hair color.

The JavaBean specification suggests a naming convention to be used when creating properties. A property should be private or protected. Public methods (getters and setters) should be created to give other classes access to the property if necessary. The method names should be getXXX and setXXX, where XXX is the property name. Figure 291 contains a Java code sample for an age property.

```
private int age;
public int getAge() {
 return age;
}
public void setAge(int newAge) {
 age = newAge;
}
```

Figure 291. Age Property Java Code Sample

Method A method for a bean is nothing different from any regular Java method. It is simply an interface in which beans can communicate by passing parameters and getting values or objects back.

There are several points to keep in mind when creating methods. Name the methods descriptive enough so someone else using your bean has an idea of what a method does simply by seeing the name. Also, unless you specify exactly which methods to show and which ones to name in a BeanInfo class (see Advanced JavaBeans concepts), all public methods are displayed using Introspection. Only make public the methods you want other objects to invoke.

Events Events are a way for beans to communicate by allowing a bean to let other beans know when something has occurred. For example, a button must be able to let other beans know if it has been pushed. There are several events already built into Java such as events for mouse movements, windows being opened or minimized, and so on. New events can also be created and used by a bean to allow almost anything to be communicated. For example, a database access bean can fire (inform other beans of) an event if the database connection has closed so the rest of the application can take appropriate action.

Introspection

Introspection is one of the concepts that make JavaBeans easy to use and be used by others. Introspection means that a builder tool or person analyzes the bean first to determine what properties, methods, and events the bean has.

Customization

Customization is exactly as it sounds—the ability to change a bean to better suit your application's needs. Customization makes beans powerful and reusable by giving a developer an easy way to change the look or functionality of a bean. New application development does not have to take place. The JavaBeans specification gives us two ways to do this. First, property editors can be created to make changing a bean property easier and more robust by checking to make sure a property's value is set within a valid range, and so on. Second, a Customizer class can be created, which can be a wizard to take a developer step-by-step through using the bean.

Persistence

Persistence means that your data exists even after you close the program or shut off the computer. This is important when using customizable beans. If you customize a bean with a builder tool, the state of the bean can be saved to disk and brought back later.

9.3 Creating a Simple JavaBean

This section shows you how to create a simple bean named FancyLabel. It demonstrates how we can externalize methods and properties. The function of this bean allows us to add a label to our application that can sense when the mouse moves over it and changes color accordingly. The code for the bean is shown in the following example. It was written using VisualAge for Java.

When we run the application, the label named Fancy is shown in red when the mouse passes over it and green when the mouse is not over it. The window in Figure 292 appears.

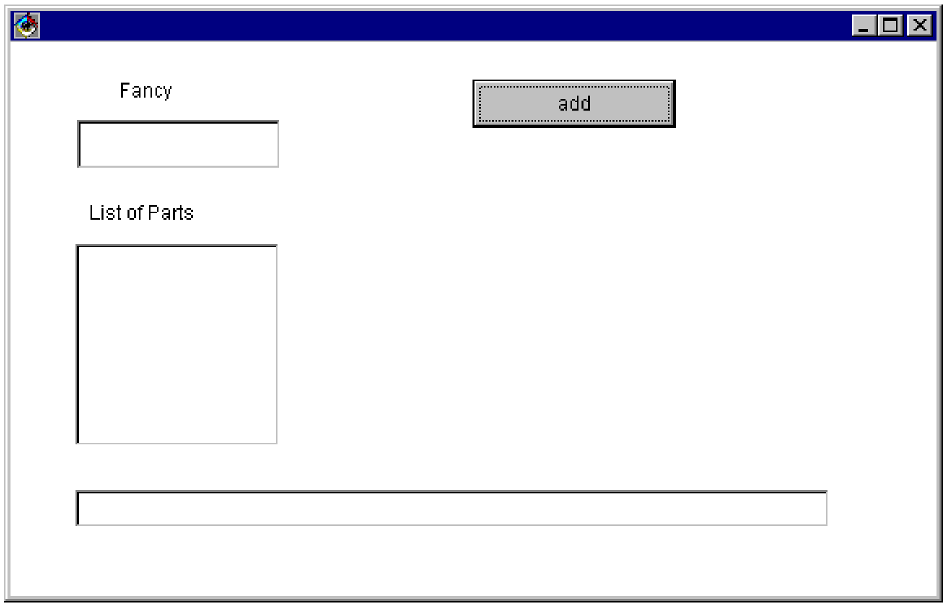


Figure 292. FancyLabel Example

To make FancyLabel a simple bean, we publicize its methods, properties, and events. Actually, we do not need to do anything with all the public methods. They are automatically seen.

We have three properties that we make available for others to use:

- mouseInsideColor
- mouseOutsideColor
- mouseInside

```
public class FancyLabel extends java.awt.Label implements
java.awt.event.MouseListener {

 // Properties
 public java.awt.Color mouseInsideColor = java.awt.Color.red;
 public java.awt.Color mouseOutsideColor = java.awt.Color.green;
 public boolean mouseInside = false;

 }
```

Figure 293. FancyLabel Class Definition

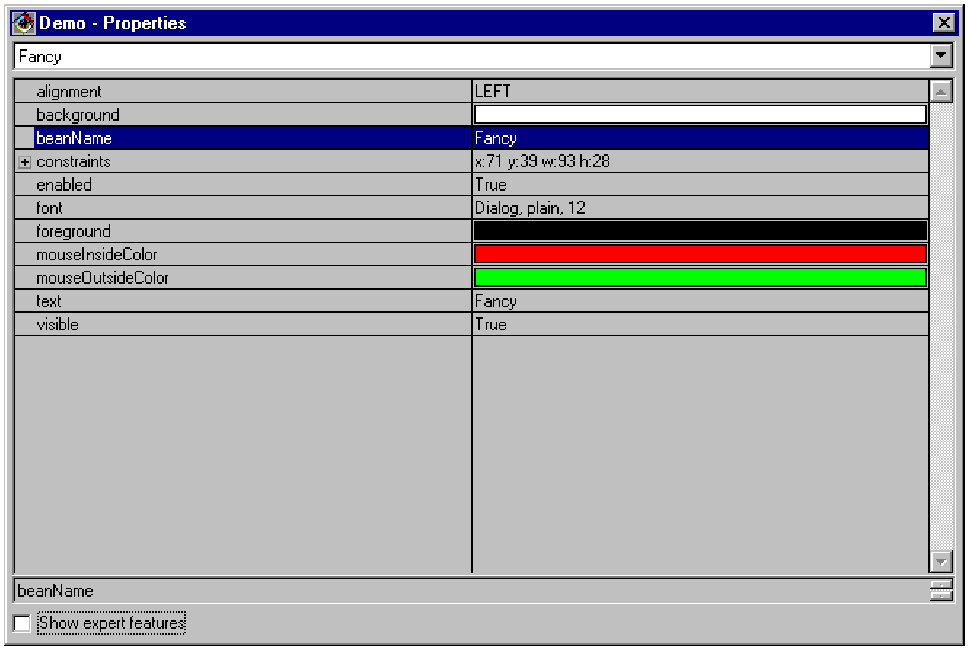
We use JDK1.1 events to control the application.

```
Mouse Listener Methods
public void mouseClicked(java.awt.event.MouseEvent e) {
  repaint();}
public void mouseEntered(java.awt.event.MouseEvent e) {
   mouseInside = true;
   paint(java.awt.Graphics );}
public void  mouseExited(java.awt.event.MouseEvent e) {
   mouseInside = false;
   paint(java.awt.Graphics );}
public void  mousePressed(java.awt.event.MouseEvent e) {}
 public void mouseReleased(java.awt.event.MouseEvent e) {}
 // Sets the color and calls the java.awt.Label's paint method
public void paint(java.awt.Graphics g) {
 if (isMouseInside())
  setForeground(mouseInsideColor);
 else
  setForeground(mouseOutsideColor);
 super.paint(g);}
```

Figure 294. Listener Methods

We implement the java.awt.event.MouseListener interface. We only use two of the MouseListener interface methods, mouseEntered and mouseExited. We override the paint method to set the actual label color.

Because FancyLabel is a bean, we can use a tool such as VisualAge for Java to display its methods, properties, and events. Figure 295 shows the properties of FancyLabel.

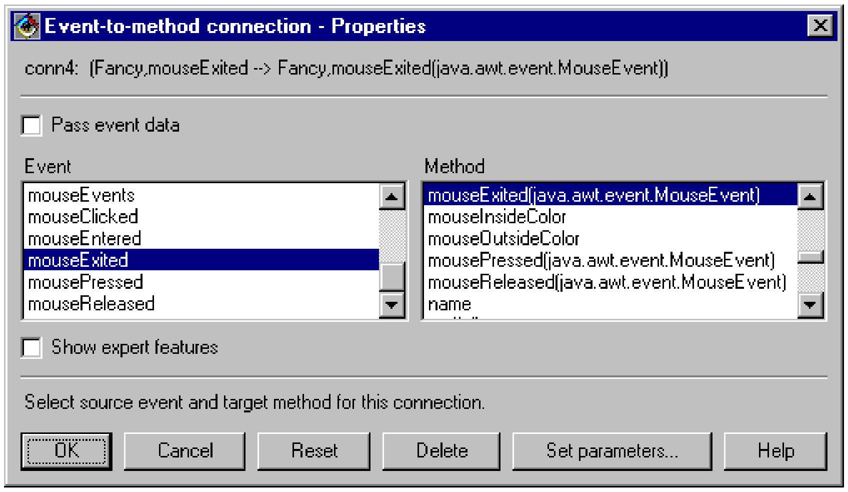


alignment	LEFT
background	⬜
beanName	Fancy
constraints	x:71 y:39 w:93 h:28
enabled	True
font	Dialog, plain, 12
foreground	⬛
mouseInsideColor	🟥
mouseOutsideColor	🟩
text	Fancy
visible	True

beanName

Figure 295. FancyLabel Properties

Figure 296 shows the methods available with FancyLabel.



conn4: (Fancy,mouseExited --> Fancy,mouseExited(java.awt.event.MouseEvent))

Pass event data [ ]

Event	Method
mouseEvents	mouseExited(java.awt.event.MouseEvent)
mouseClicked	mouseInsideColor
mouseEntered	mouseOutsideColor
mouseExited	mousePressed(java.awt.event.MouseEvent)
mousePressed	mouseReleased(java.awt.event.MouseEvent)
mouseReleased	name

Show expert features [ ]

Select source event and target method for this connection.

Figure 296. FancyLabel Methods

To make the bean work with our application, we make two connections as shown in Figure 297 on page 345:

- Event **mouseEntered** to the mouseEntered method
- Event **mouseExited** to the mouseExited method

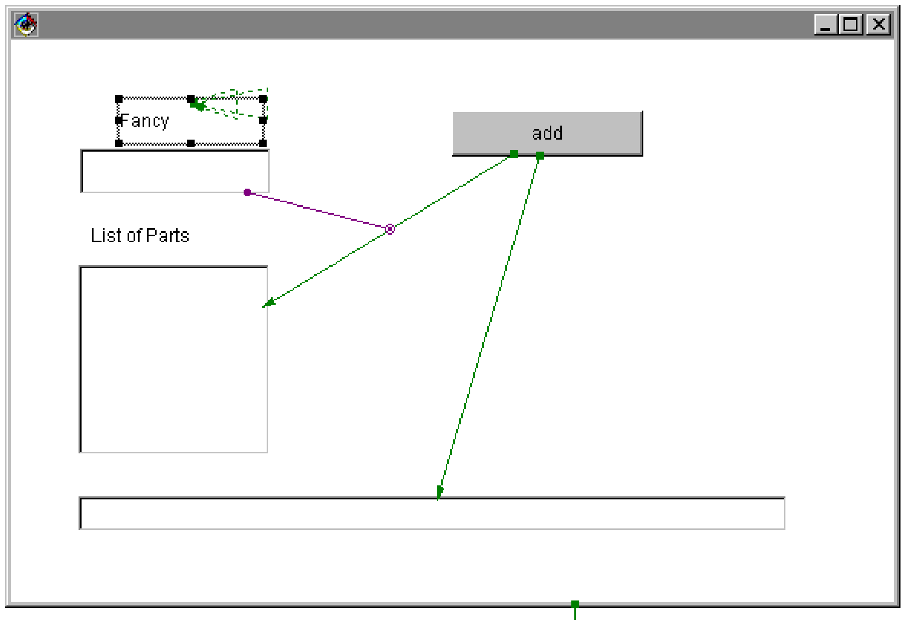


Figure 297. VisualAge for Java VCE Connections

This example shows some of the basic capabilities of JavaBeans. The key capabilities that JavaBeans bring us are the ability to externalize properties and methods and to use a visual tool to work with these properties and methods.

9.4 Creating a Program Call JavaBean

This section presents a more practical use of JavaBeans. We create a JavaBean that allows us to run an AS/400 RPG program by using the SmartGuide provided by Enterprise Toolkit/400. We also create an application that uses the program call JavaBean that we make.

9.4.1 Distributed Program Call Feature

Instead of directly writing to the Distributed Program Call (DPC) support provided by the AS/400 Toolbox for Java, we create a Program Call JavaBean and create an application which uses it. First, let us review the AS/400 Toolbox for Java Program Call feature.

9.4.1.1 The Program Call Feature

The Program Call feature of the AS/400 Toolbox allows a Java program to directly run any non-interactive program object (*PGM) on the AS/400 system. It passes input data as parameters and returns results through parameters.

The Java developer must use the data conversion classes from the Toolbox to convert input parameters from a Java format to an AS/400 data type and convert output parameters from an AS/400 format to a Java format.

The advantage of using the Distributed Program Call class is that native AS/400 non-interactive programs can be run from a Java application unchanged. Native program calls can also result in better performance of a Java application when compared with JDBC. Additionally, this interface can call programs on the AS/400 system that do more than just database access. For example, a Java application can call a program that starts nightly job processing, saves libraries to tape, or sends or receives data through communication lines.

Calling a native AS/400 program involves the following steps:

1. Connect to the AS/400 system by creating an AS400 object.
2. Create a ProgramCall object.
3. Define and initialize a ProgramParameter array for passing parameters to or from the called program.
4. Use the Data Conversion classes to convert input parameter values from a Java format to an AS/400 format.
5. Use the setProgram method to specify the qualified name of the program to call and parameters to use, if not declared on the ProgramCall constructor.
6. Execute the program using the run method.
7. If the run method fails, obtain detailed error information through AS400Message objects.
8. Retrieve output parameters using the getOutputData method of the ProgramParameter object.
9. Convert output parameter values using the data conversion classes.

9.4.2 Application Description

In this example, we use the Distributed Program Call (DPC) interface (Figure 298) to allow a client program to call an AS/400 program.

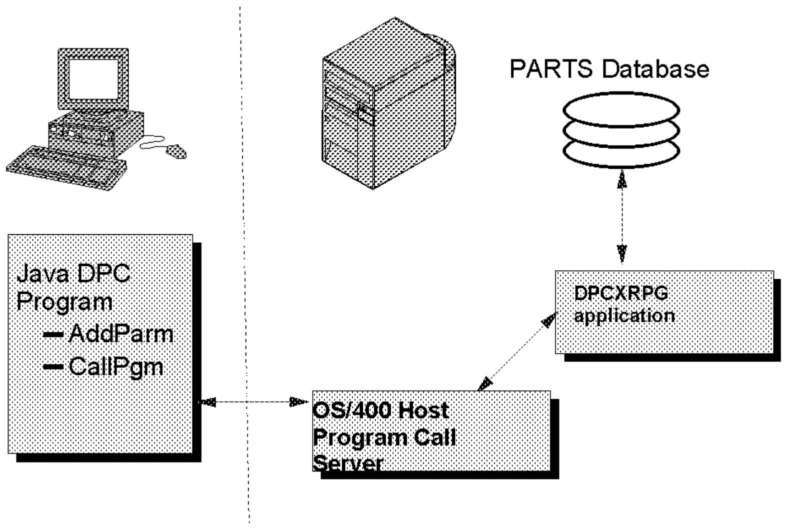
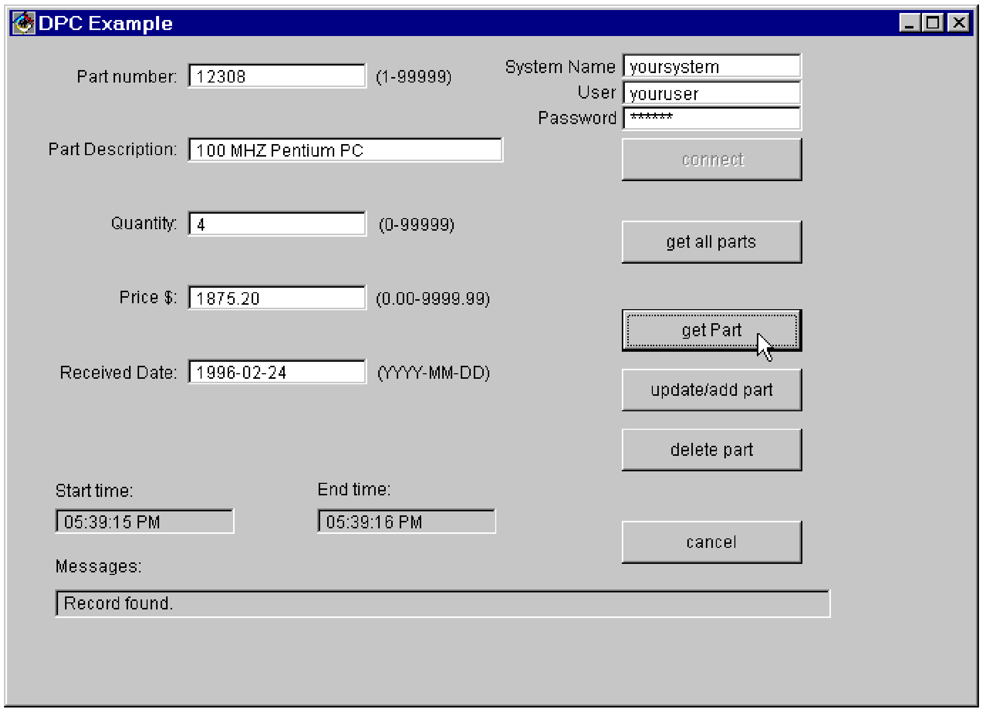


Figure 298. Distributed Program Call Example

The client program requests data from the AS/400 database by calling an AS/400 program. Information is passed between the programs using parameters. It is up to the application implementer to handle data conversions.



Part number:	12308	(1-99999)
Part Description:	100 MHZ Pentium PC	
Quantity:	4	(0-99999)
Price $:	1875.20	(0.00-9999.99)
Received Date:	1996-02-24	(YYYY-MM-DD)
System Name	yoursystem	
User	youruser	
Password	******	
Start time:	05:39:15 PM	
End time:	05:39:16 PM	
Messages:	Record found.	

Figure 299. Distributed Program Call Example

The client program requests data from the server program by calling it and passing it parameters. The input parameters are a flag, a part number, and all attributes of a part. For example, S12301 is a request for a single record (Flag = S) of part number 12301. If requesting all parts (Flag=A), the part number is not necessary. The server program, DPCXRPG, searches the database for the requested information. The result is passed back in the output parameters.

9.4.2.1 RPG Program Background

Library: APILIB

Program Name: DPCXRPG

Parameters (all are used as Input/Output):

Table 48. Parameter List

Sequence / Field	Description	Length/Type
1 / OPTION	In: Operation Code / Out: Return Code	1 character
2 / PARTNO	Part Number	5.0 packed
3 / PARTDS	Part Description	25 character
4 / PARTQY	Part Quantity	5.0 packed
5 / PARTPR	Part Price	6.2 packed
6 / PARTDT	Part Date Received	10 date

Values of the Operation Code (Input OPTION):

Table 49. Flag Operation Codes

Operation Code	Database Operation to Execute
S	Retrieve a single record for the supplied key.
A	Retrieve all records.
F	Fetch next record based on the current position.
E	End the program.
D	Delete a single record for the supplied key.
U	Update a single record for the supplied key with the attribute data. Write a single record for the supplied key with the attribute data if it does not yet exist.

Values of the Return Code (Output OPTION):

Table 50. Flag Operation Codes

Return Code	Result Description
Y	Normal: Operation has succeeded / When operation code was U: Record updated
X	Normal: Operation has failed / When operation code was U: Record added
U	Unknown operation code has been supplied

9.4.3 Creating a Program Call JavaBean

As shown in Figure 300 on page 349, to start the ET/400 Program Call SmartGuide, we highlight a package and use the right mouse button to select Create Program Call. The output is a new class or JavaBean in the selected package.

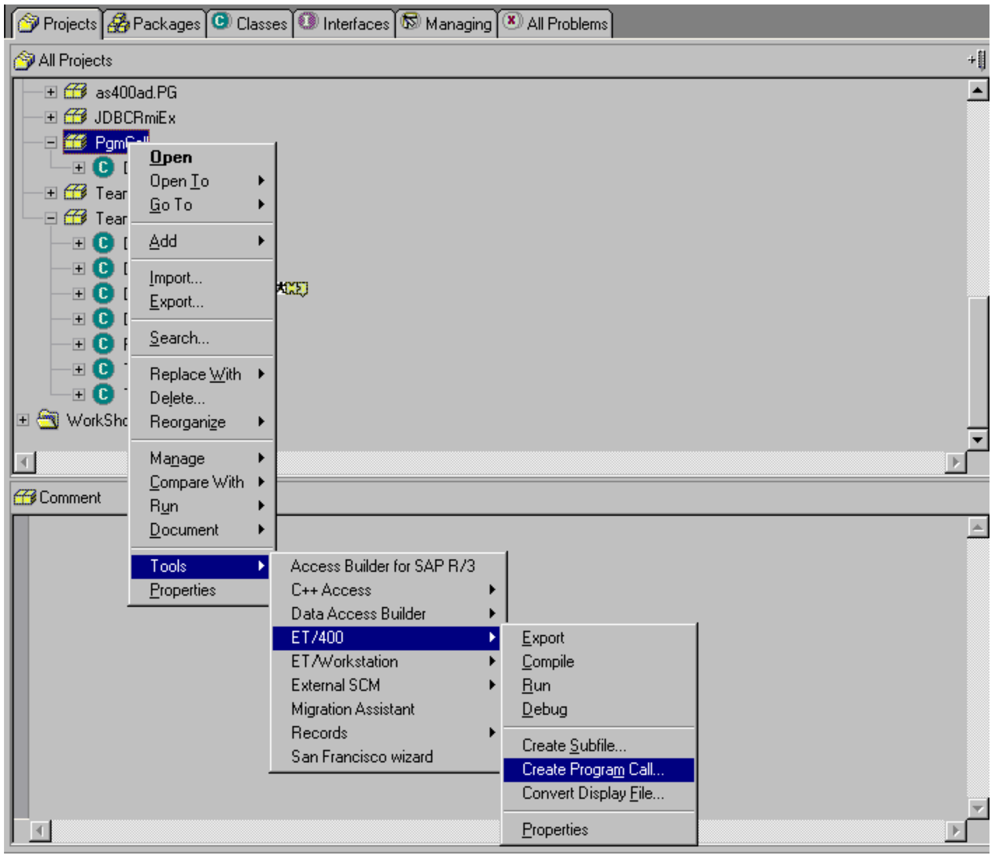
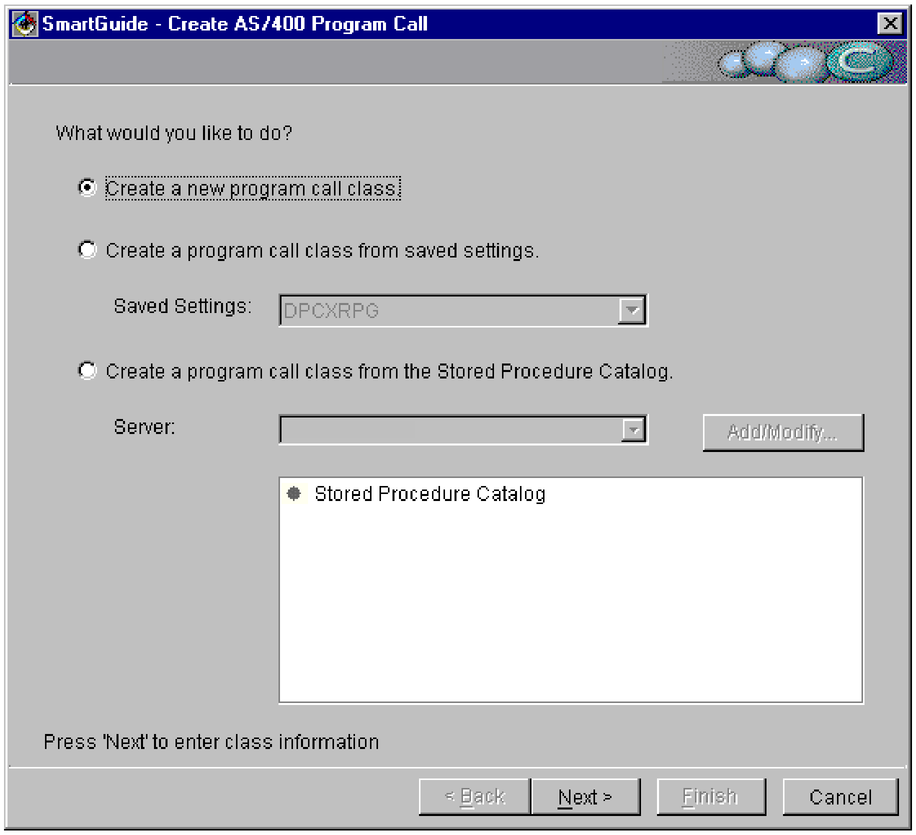


Figure 300. ET/400 Create Program Call

As shown in Figure 301 on page 350, the first AS/400 Program Call SmartGuide window allows us to choose whether we want to create a new class or modify an existing class. We choose Create a new program call class.



SmartGuide - Create AS/400 Program Call

What would you like to do?

(●) Create a new program call class.

( ) Create a program call class from saved settings.

Saved Settings: DPCXRPG

( ) Create a program call class from the Stored Procedure Catalog.

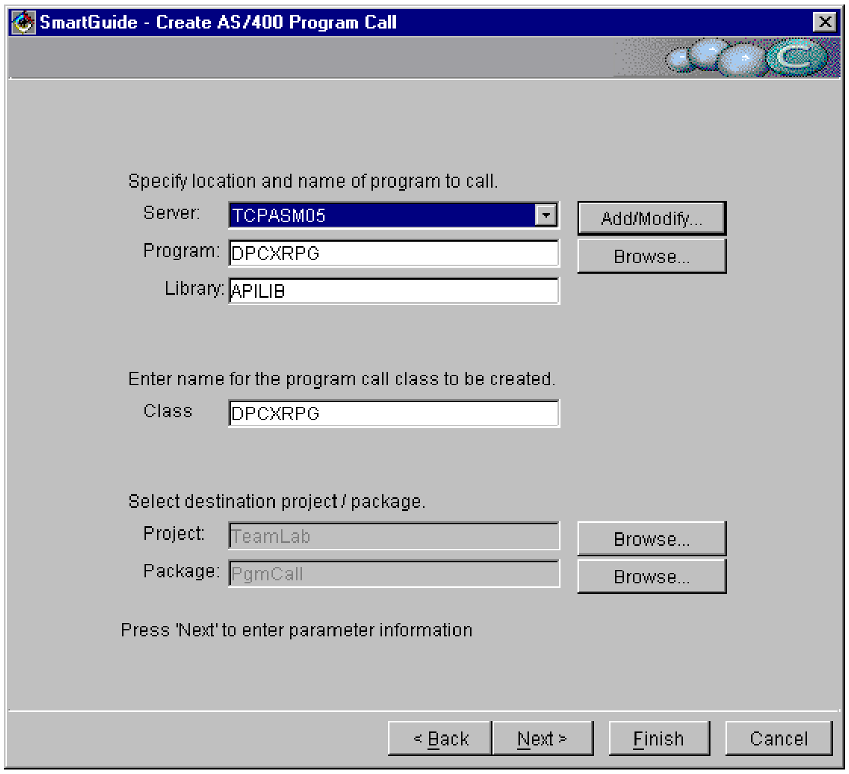
Server:

Stored Procedure Catalog

Press 'Next' to enter class information

Figure 301. SmartGuide — Create AS/400 Program Call 1

Next, we specify the name of the AS/400 system, the name of the program, and the library where the program is found. We can choose the Browse button, and the SmartGuide interactively retrieves the programs from the specified library. We then select the program that we want to call. Once we enter the name program to call, we enter the name of the JavaBean or class that we want to generate and the project and package in which to store it.



SmartGuide - Create AS/400 Program Call

Specify location and name of program to call.

Server: TCPASM05

Program: DPCXRPG

Library: APILIB

Enter name for the program call class to be created.

Class DPCXRPG

Select destination project / package.

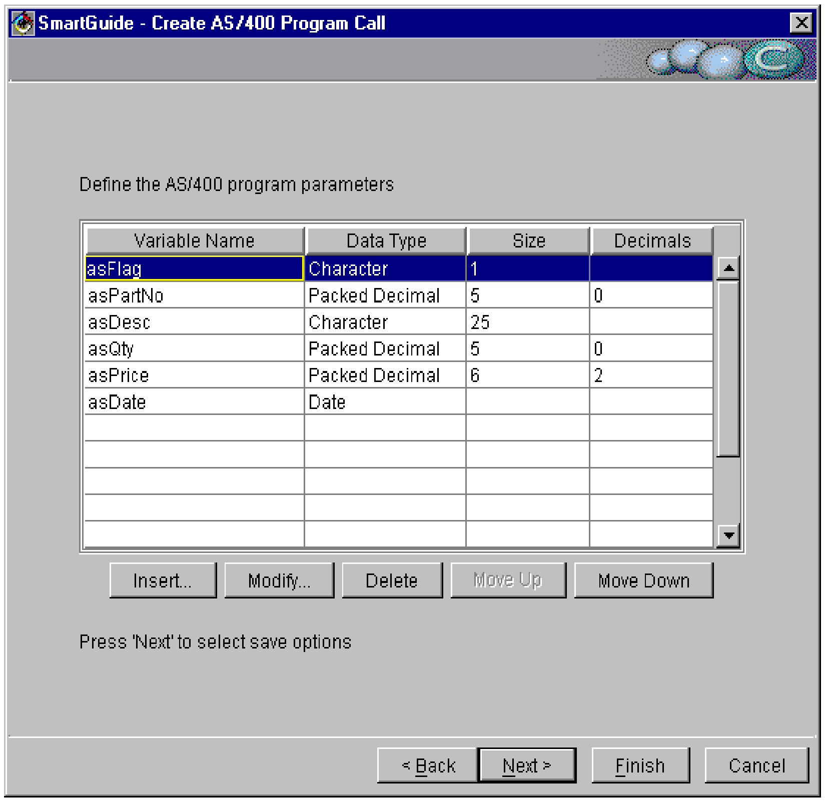
Project: TeamLab

Package: PgmCall

Press 'Next' to enter parameter information

Figure 302. SmartGuide — Create AS/400 Program Call 2

Finally, we need to define the parameters that the AS/400 program uses. The SmartGuide provides choice boxes to insure that we configure the parameters properly. Since this is an AS400 RPG program, the SmartGuide does not have any way of knowing what the required parameters are. We need to check the AS/400 program to determine this. In Figure 302, we show the parameters required for the AS/400 RPG Program named DPCXRPG.



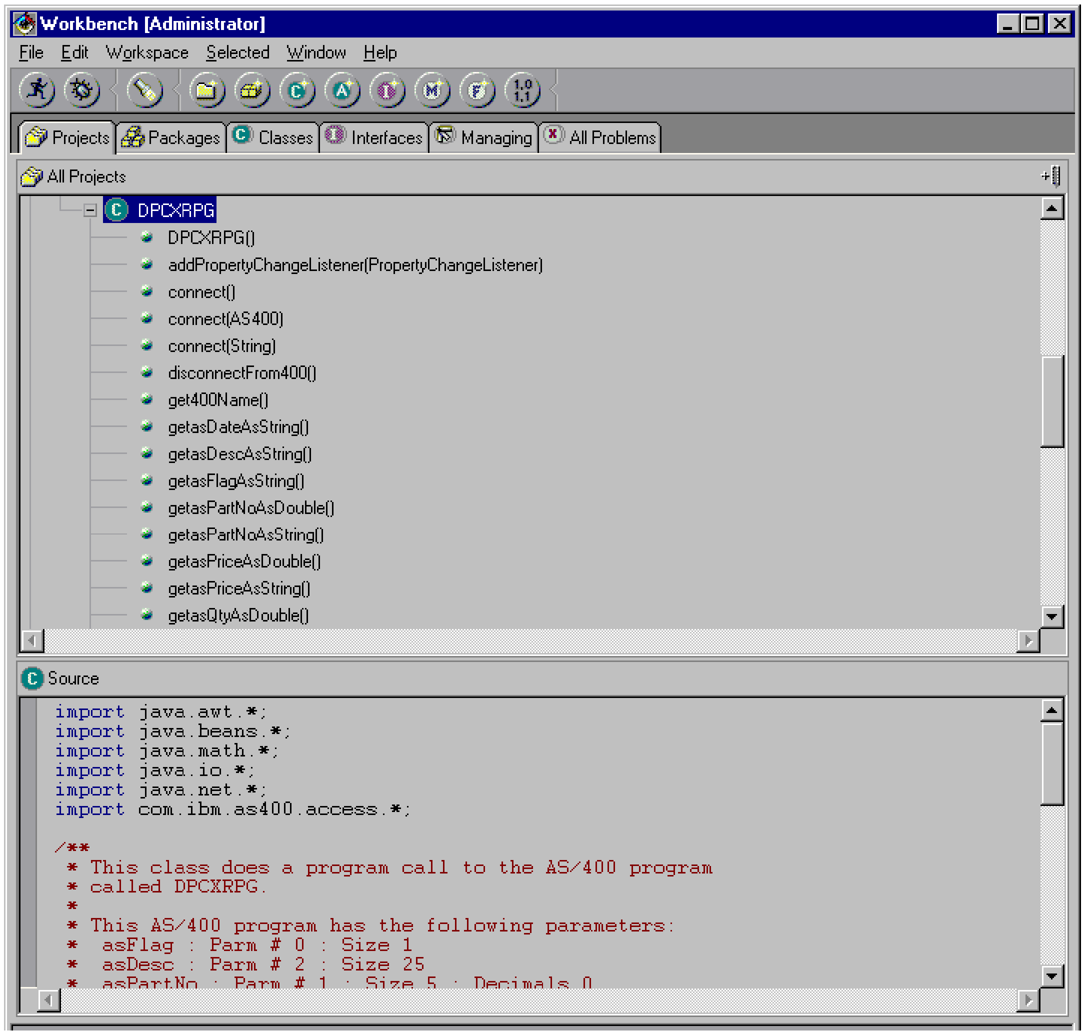
Define the AS/400 program parameters

Variable Name	Data Type	Size	Decimals
asFlag	Character	1	
asPartNo	Packed Decimal	5	0
asDesc	Character	25	
asQty	Packed Decimal	5	0
asPrice	Packed Decimal	6	2
asDate	Date		

Press 'Next' to select save options

Figure 303. SmartGuide — Create AS/400 Program Call 3

Click on the Finish button to generate the new bean. In this case, a bean named DPCXRPG is generated. A number of methods are generated that allow us to effectively use the generated bean. The generated methods are shown in Figure 304 on page 353.


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
Source

DPCXRPG
- DPCXRPG()
- addPropertyChangeListener(PropertyChangeListener)
- connect()
- connect(AS400)
- connect(String)
- disconnectFrom400()
- get400Name()
- getasDateAsString()
- getasDescAsString()
- getasFlagAsString()
- getasPartNoAsDouble()
- getasPartNoAsString()
- getasPriceAsDouble()
- getasPriceAsString()
- getasQtyAsDouble()

```
import java.awt.*;
import java.beans.*;
import java.math.*;
import java.io.*;
import java.net.*;
import com.ibm.as400.access.*;

/**
 * This class does a program call to the AS/400 program
 * called DPCXRPG.
 *
 * This AS/400 program has the following parameters:
 *  asFlag : Parm # 0 : Size 1
 *  asDesc : Parm # 2 : Size 25
 *  asPartNo : Parm # 1 : Size 5 : Decimals 0
```

Figure 304. DPCXRPG Generated Methods

The constructor method generated, DPCXRPG(), does not initialize the date properly. This causes a problem for the host AS/400 RPG program. We fix this problem by changing the line of code in the DPCXRPG() constructor method from:

```
setasDateAsString (" ");
```

to:

```
SetasDateAsString("1999-01-01");
```

The SmartGuide generates several connection methods for connecting to the AS/400 system. It does not generate a connect method that takes three parameters (system name, user ID and password). We add such a method to the DPCXRPG bean to make our application development easier. The new connection method is shown in Figure 305 on page 354.

```
public void connect(String name, String user, String password)
{
as400 = new AS400(name, user, password);
as400Name = name;
connect();
}
```

Figure 305. New connect Method

This user-supplied connect method accepts system name, user ID, and password as parameters. We can use this method in the Visual Composition editor and visually pass in these parameters from screen text fields.

9.4.4 Building an Application Using the DPCXRPG Bean

This section defines how to create a class, named DpcxRpg2, which uses the DPCXRPG JavaBean. We use the ToolboxGUI class to create a new class that is a subclass of java.awt.Frame and implements the PartsContainer interface. For details about the ToolboxGUI class and the PartsContainer interface, please refer to Section 3.5.6, “Reusable GUI Part” on page 126.

In the class description, we declare a DPCXRPG object, which we name aDpcxRpg. We use this object to run the AS/400 program. We also declare an AS400 object, named as400, which provides connectivity to the AS/400 system.

```
import com.ibm.as400.access.*;
import PgmCall.*;
public class DpcxRpg2 extends java.awt.Frame implements
java.awt.event.WindowListener, WorkShop.PartsContainer {
private DPCXRPG aDpcxRpg;
private AS400 as400;
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
private ProgramCall pgm;
}
```

Figure 306. Class Description for DPCXRPG Object

We write a connectToDB method which is shown in Figure 307. This code example uses the user-written connect method.

```
public void connectToDB(String systemName, String userid, String password)
throws Exception
{
aDpcxRpg = new DPCXRPG();
aDpcxRpg.connect(systemName, userid, password);
return;
}
```

Figure 307. The connectToDB Method

We pass in system name, user ID, and password. A new DPCXRPG object is named aDpcxRpg is instantiated and its connect method is called.

The getRecord method shown in Figure 308 takes advantage of the DPCXRPG bean’s generated methods.

```
public String getRecord(String partNo,
          java.awt.TextField partDesc, java.awt.TextField
          partQty, java.awt.TextField partPrice,
          java.awt.TextField partDate) throws Exception {
// Setup the parameters
aDpcxRpg.setasFlagAsString("S");
aDpcxRpg.setasPartNoAsString(partNo);

   // Run the program
if (aDpcxRpg.runProgram() != true) {
              handle error
```

Figure 308. The getRecord Method (Part 1 of 2)

We use the setter methods to set the value for the flag and the part number. Methods are provided to convert these parameters to Strings. This is much easier than doing all the conversion work on your own. The runProgram method is provided to actually make the program call.

The code example in Figure 309 shows the advantages of using the generated methods.

```
else {
    if (aDpcxRpg.getasFlagAsString().equals("Y")) {
partDesc.setText(aDpcxRpg.getasDescAsString());
    partQty.setText(aDpcxRpg.getasQtyAsString());
    partPrice.setText("$" + aDpcxRpg.getasPriceAsString());
    partDate.setText(aDpcxRpg.getasDateAsString());
  }
 else {
    partDesc.setText("");
    partQty.setText("");
    partPrice.setText("");
    partDate.setText("");
    return "Record not found.";
}
```

Figure 309. The getRecord Method (Part 2 of 2)

We use the generated getter methods to retrieve the information returned as parameters. Notice that getter methods are provided that return the values as Strings, so we can display them to the window.

In the populateListBox method shown in Figure 310 on page 356, we use the methods of the DPCXRPG bean to populate the listbox with records from the PARTS file.

```
public void populateListBox(MultiColumnListbox
                      aListBox) throws Exception {
   String[] array = new String[5];
 // Setup the parameters
   aDpcxRpg.setasFlagAsString("A");
    if (aDpcxRpg.runProgram() != true) {
// Note that there was an error
System.out.println( "program failed:"  );
   // Show the messages
      return;
   }
    else {
```

Figure 310. The populateListBox Method (Part 1 of 2)

We use the generated setter methods to set the value for the flag. Methods are provided to convert these parameters to Strings. In this case, we send a Flag set to "A" to tell the AS/400 program to position to the first record. The runProgram method is provided to actually make the program call.

If we successfully position to the first record, we call the AS/400 program to retrieve records from the file.

```
 if (aDpcxRpg.getasFlagAsString().equals("Y")) {
      aDpcxRpg.setasFlagAsString("F");
    do {
        if (aDpcxRpg.runProgram() != true) {
       // Handle error}
        else {
           if (aDpcxRpg.getasFlagAsString().equals("Y")) {
       array[0] =aDpcxRpg.getasPartNoAsString();
      array[1] =aDpcxRpg.getasDescAsString();
      array[2] =aDpcxRpg.getasQtyAsString();;
      array[3] = "$" + aDpcxRpg.getasPriceAsString();;
      array[4] =aDpcxRpg.getasDateAsString();
       aListBox.addRow(array);
      }
              }
       } while (aDpcxRpg.getasFlagAsString().equals("Y"));
}
```

Figure 311. The populateListBox Method (Part 2 of 2)

To sequentially retrieve records, we send the AS/400 program a flag set to "F". If a record is returned (we check for a flag set to "Y" in the return from the AS/400 program), we format and place it in the multi-column listbox. We use the bean's getters and setters to help do this. We do this in a While loop to retrieve all the records. Note that if we wanted to improve the performance of this method, we can return multiple records at a time.

Calling an AS/400 program is made much easier by using the ET/400 Program Call SmartGuide to generate a JavaBean and using the generated JavaBean as part of the application. The generated JavaBean provides getter and setter methods for accessing and setting program parameters. These methods handle conversions between the AS/400 and Java formats. Methods are also provided to run the program and check error conditions.

9.4.5 Using the DPCXRPG Bean in the VCE

You can also use the DPCXRPG bean as a non-visual part in a visual editor. This section describes how to use the generated bean as a non-visual component in the VisualAge for Java Visual Composition Editor. We create a class named DPCExample2. We open the Visual Composition Editor and build the graphical user interface. We add a DPCXRPG bean as a non-visual part, called aDpcxRpg. This has the effect of instantiating a DPCXRPG object, which we can use in the visual builder. This is shown in Figure 312.

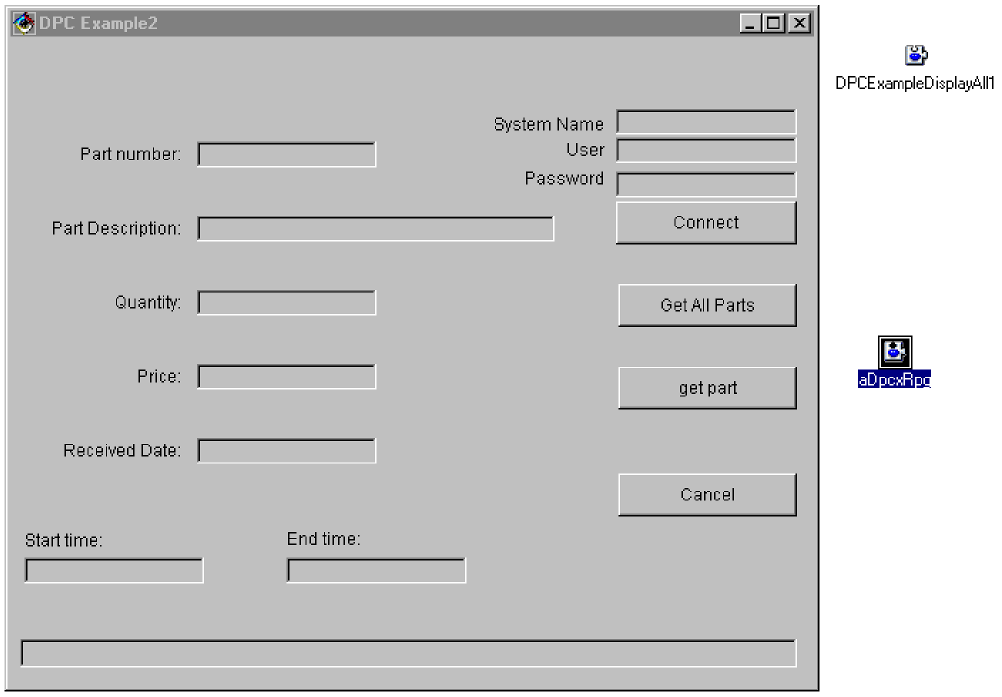
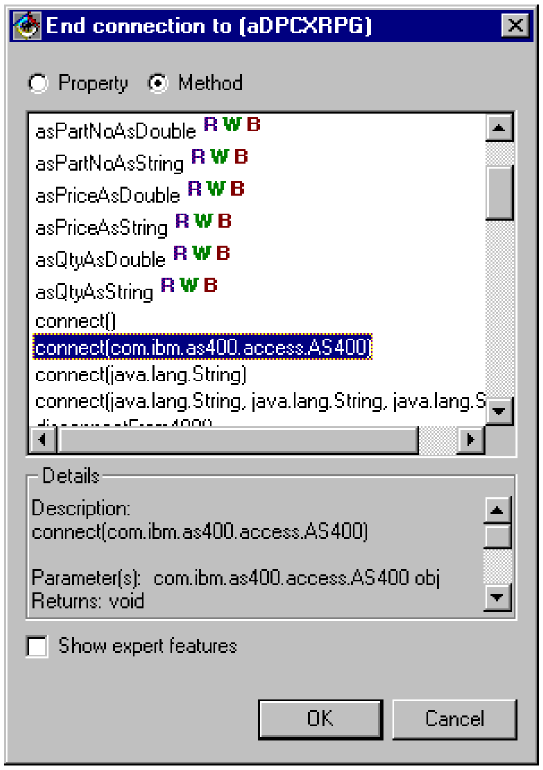


Figure 312. Building the Graphical User Interface

We can use the VCE to display the methods available with DPCXRPG. To handle the event generated when a user clicks on the Connect button, we connect the actionPerformed event of the button to the DPCXRPG bean's connect method which takes three parameters. The DPCXRPG bean's methods are shown in Figure 313 on page 358.



End connection to (aDPCXRPG)

( ) Property [✓ Method]

asPartNoAsDouble R W B
asPartNoAsString R W B
asPriceAsDouble R W B
asPriceAsString R W B
asQtyAsDouble R W B
asQtyAsString R W B
connect()
connect(com.ibm.as400.access.AS400)
connect(java.lang.String)
connect(java.lang.String, java.lang.String, java.lang.S

Details

Description:
connect(com.ibm.as400.access.AS400)

Parameter(s): com.ibm.as400.access.AS400 obj
Returns: void

[ ] Show expert features

OK Cancel

Figure 313. DPCXRPG Methods

As shown in Figure 314, we pass in the three parameters. We connect the System Name, User, and Password TextFields fields text property to the connecting line between the connect button and the DPC bean.

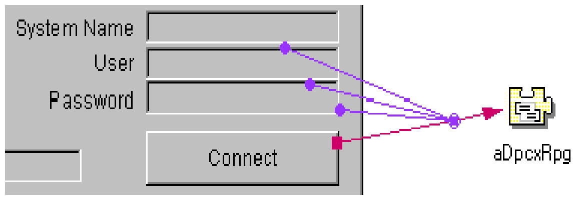


Figure 314. Passing in the Parameter to the connect Method

Figure 315 shows the Cancel button connections. We connect the ActionPerformed event of the Cancel button to the DPC bean's disconnectFromAS400 method. We also connect the ActionPerformed event of the Cancel button to the dispose method of the form.

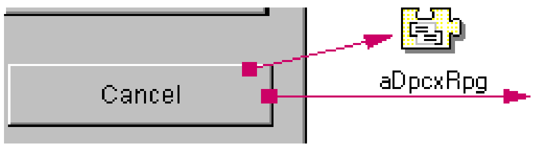


Figure 315. Cancel Button Processing

To handle the action performed when the user clicks on the Get Part button, we connect the ActionPerformed event of the Get Part button to the DPCXRPG bean's asFlagAsString method and supply the connection with the parameter value "S." To set the parameter, we double click on the connection line and click on Set parameters. Next, we connect the ActionPerformed event of the button to the DPCXRPG bean's asPartNoAsString method and supply the connection with the parameter value of the text field Part number. Finally, we connect the ActionPerformed event of the button to the bean's runProgram method. These connections are shown in Figure 316.

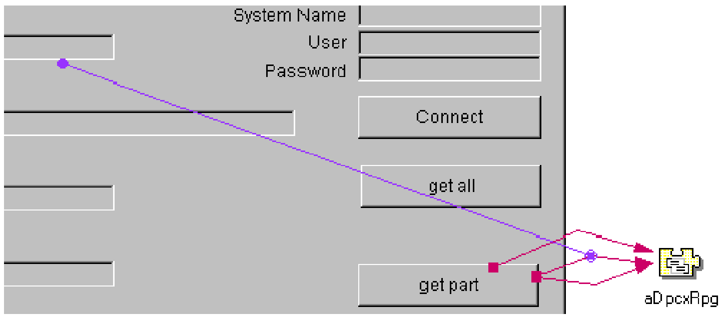


Figure 316. Get Part Button Processing

To set the other parameters, perform the following steps:

1. Move the mouse pointer on top of the last connection line (for runProgram).
2. Click the right mouse button. Select **Connect** and **NormalResult** from the pop-up menu that appears.
3. Drag the mouse and click the left mouse button on the Part description text field. Select **Text** from the pop-up menu.
4. Move the mouse pointer on top of the new connection line.
5. Click the right mouse button. Select **Connect** and **Value** from the pop-up menu that appears.
6. Drag and click the left mouse button on the DPCXRPG bean. Select **All Features** from the pop-up menu and the **getasDescAsString ( )** method.
7. Click **OK**.

Repeat steps one through seven for the rest of the text fields in the part using the appropriate getter methods from the DPCXRPG bean. The completed Get Part button's connections are shown in Figure 317 on page 360.

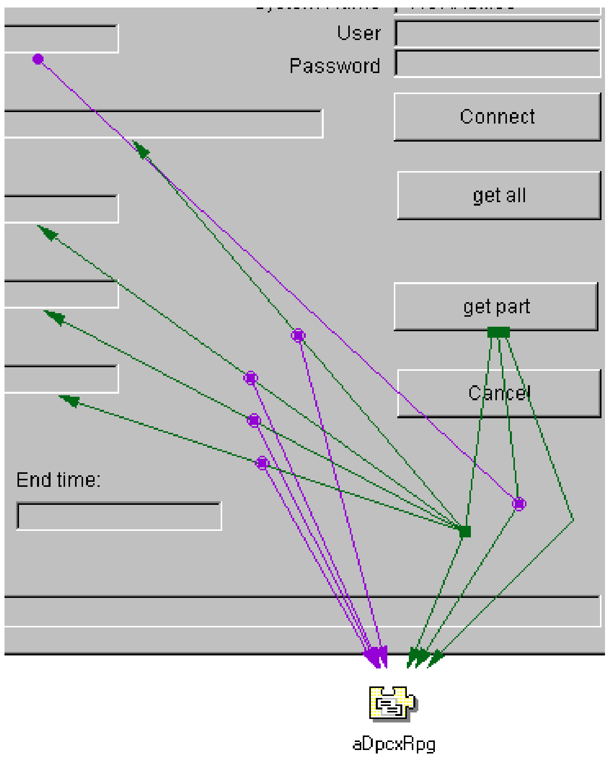
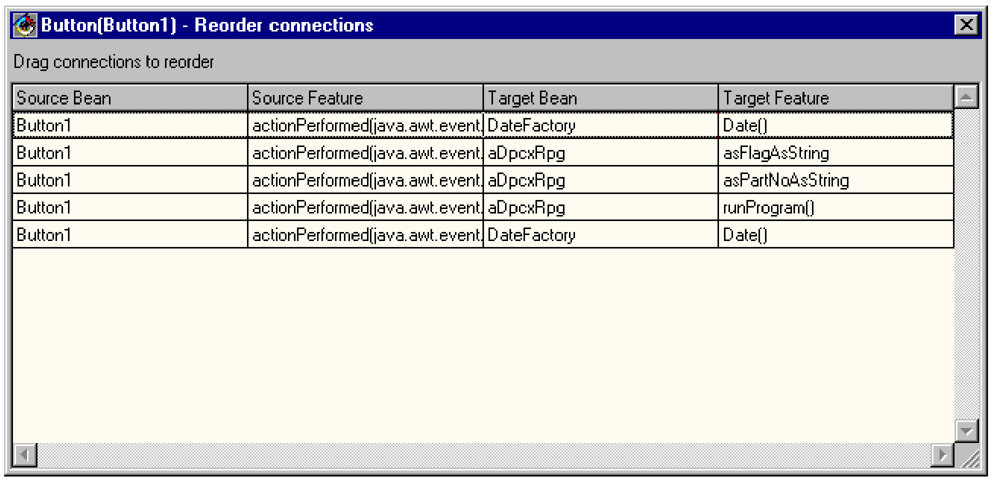


Figure 317. Completing the Connections for the Get Part Button

The Get Part button connections should appears as shown in Figure 318.


Button(Button1) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
Button1	actionPerformed(java.awt.event	DateFactory	Date()
Button1	actionPerformed(java.awt.event	aDpcxRpg	asFlagAsString
Button1	actionPerformed(java.awt.event	aDpcxRpg	asPartNoAsString
Button1	actionPerformed(java.awt.event	aDpcxRpg	runProgram()
Button1	actionPerformed(java.awt.event	DateFactory	Date()

Figure 318. Get Part Connections

In the VCE, the final application is shown in Figure 319 on page 361.

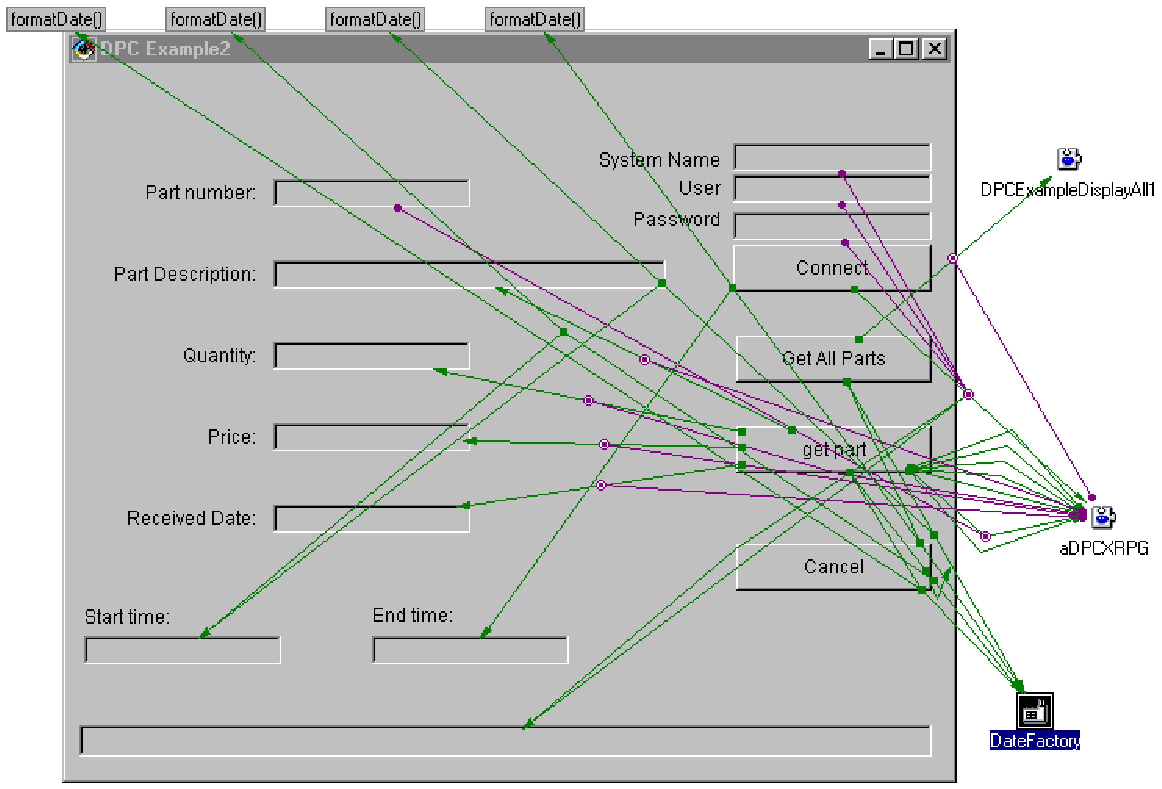


Figure 319. Completed Application

That completes the processing to connect to the AS/400 system and to call an AS/400 RPG program. To complete the application, we supply a DPCExampleDisplayAll bean to handle retrieving all parts from the database.

We built this entire application in the VisualAge for Java Visual Composition Editor. We did not write a single line of code. We used the Distributed Program Call bean that we generated using the ET/400 SmartGuide to do all the work. An additional benefit of the DPCXRPG bean is that it can easily be made available for others to use in their applications.

9.5 Advanced JavaBeans Concepts

This section explains the advanced concepts of JavaBeans.

BeanInfo Class

A bean creator may not want to leave property, method, and event finding up to the Introspector or want to add more advanced features such as custom property editors or bean customizers. A BeanInfo file can be created to let a user or builder tool know what to make public.

A BeanInfo file must have the same name as the bean with BeanInfo appended to the end. For example, our Fancy Label bean can have a class called FancyLabelBeanInfo. This BeanInfo class must either implement the java.beans.BeanInfo interface or extend the Java.beans.SimpleBeanInfo class. The Java.beans.SimpleBeanInfo class implements the java.beans.BeanInfo interface and makes it easier for a developer to quickly add only a few BeanInfo methods by overriding the methods already present in the SimpleBeanInfo class.

Note: Some builder tools do not use Introspection if a BeanInfo file is present. You must list all properties, methods, and events you want visible in the BeanInfo class.

Advanced Properties

For an application or applet to be built well graphically, the beans need to have effective communication between them. In addition to methods and regular events, beans allow two special property types: *bound* and *constrained*.

Bound properties

A bound property is the same as any other property we discussed earlier with an additional feature. Bound properties make an announcement to any interested listener that its value has changed. To let a builder tool know a property is bound, a BeanInfo class must be created and methods need to be added to the main class to support bound property listeners.

Constrained properties

A constrained or vetoable property is similar to a bound property. Not only are listeners notified when the property has changed, they have the opportunity to disallow a change to occur. For example, a person may ask a loan bean to change the interest rate to 10%, but a bank bean that contains the loan bean does not allow interest rates below 15%. The bank is informed of the possible change in the interest rate and vetoes the change.

Indexed properties

Another special property is the indexed property. An indexed property works the same as an array. It lets you work with all of the contents of the property at once, and allows you to read and write one item in the array at a time. The example in Figure 320 is an indexed property and also shows the BeanInfo data to accompany it.

```
private String[] names = {};

public void setNames(String[] allNames) {
 names = allNames;
}
public String[] getNames() {
 return names;
}
public void setIndexNames(int index, String name) {
 names[index] = name;
}
public String getIndexNames(int index) {
 return names[index];
}
```

Figure 320. BeanInfo Indexed Property

```
public class BeanNameBeanInfo extends SimpleBeanInfo{
public PropertyDescriptor[] getPropertyDescriptors() {
try {
IndexedPropertyDescriptor pd = new IndexedPropertyDescroptor("names",
BeanName,"getNames","setNames","getIndexNames","setIndexNames");
PropertyDescriptor allDescriptors[] = {pd};
return allDescriptors;
} catch(Exception e) {
return null;
}
}
}
```

Figure 321. Using an Indexed Property

Methods

You can also use the BeanInfo file to let the builder tool know which methods to make available to the bean user. You can also provide more information about a method this way, such as a better description of what the method does and better descriptions of the method's parameters.

9.5.1 What Makes a Good JavaBean

Before going out and converting or updating all of your Java classes to beans, decide which Java classes are best suited to become beans. The basic question to ask yourself is: Is this class discrete or general enough to be reused?

If you have a class that pulls data from a particular database, it is probably not worth making it a bean. On the other hand, with just a little work, a customizer can be added to that class to let a user select which database and fields to retrieve. Then you have a bean that can be reused in several applications. Another thing to be aware of is that introspection, itself, may be enough to make a good bean. For example, a standard Java button is a bean that does not have a BeanInfo file or customizer. All of its properties (text and color) simply have get and set methods. If you follow the naming conventions when making any object, less work must be done to make your object a bean.

9.5.2 References and More Information

For more information about JavaBeans, refer to the following resources:

- *JavaBeans for Dummies,* SR23-7895, by Emily Vander Veer
- Sun Web site: http://java.sun.com/beans

Chapter 10. Deployment Considerations and Tools

This chapter introduces some deployment considerations and the tools available to enable you to deploy a finished Java applications more effectively.

The two items covered are:

- How to reduce the size of Java archive files (.jar and .zip files)
- How to encrypt data transmissions between a client and server

Reducing Java archive size is important for two reasons:

- A smaller archive will be downloaded more quickly to a client.
- A smaller archive will be faster to search (this will usually be a minor performance improvement).

Secure application communication with an AS/400 system is important if your application is deployed on the Internet and the data is sensitive (such as banking information). Secure Sockets Layer (SSL) encrypts data at a socket layer before it is transmitted between a client and server. The concepts and environment configuration can be quite complex. However, the modifications required to an AS/400 Toolbox for Java program are very simple.

New in OS/400 V4R4 is the ability for host servers to communicate using SSL support. To provide this function in Java, the AS/400 Toolbox for Java now has a SecureAS400 Object. SSL conversations can only take place between an AS/400 Toolbox for Java class and a V4R4 system that has SSL-enabled host servers.

10.1 Java Archive Files

Reducing the size of a Java archive can be important when the archive is downloaded across the Internet or through a slow WAN connection. The current jt400.jar file is 2.3 MB. With a slow modem or congested Internet route, downloading this file from a Web server can be time consuming.

To help overcome this problem, two tools are available with the AS/400 Toolbox for Java Modification 2. The following classes are found in the /QIBM/ProdData/Http/Public/jt400/utilities integrated file system directory and perform similar functions:

- **JarMaker —** A general purpose archive tool, primarily used to extract classes from jar files and re-package them in new jar files.
- **AS400ToolboxJarMaker —** An extension to JarMaker, specifically for AS/400 Toolbox for Java jar files. It can be used to extract specified AS/400 components from the jt400.jar file to produce new jar file containing only the selected components.

10.1.1 JarMaker

The JarMaker class is used to generate a smaller (and, therefore, faster loading) jar or zip file from a larger one. In addition, you can also use JarMaker to:

- Extract desired files from a jar or zip file.
- Split a jar or zip file into smaller jar or zip files.

You can use JarMaker in a program, or run it from a command line:

```
java utilities.JarMaker [ options ]
```

You must specify one of the following options:

```
-requiredFile
-additionalFile
-package
-extract
-split
```

If the following options are specified multiple times in a single command string, only the final specification applies:

```
-source
-destination
-additionalFilesDirectory
-extract
-split
```

Other options have a cumulative effect when you specify them multiple times in a single command string.

10.1.1.1 Options

These options control what JarMaker will do:

- **-source *sourceJarFile***

  Specifies the source jar or zip file to extract the required classes from. If you specify a relative path, the path is assumed to be relative to the current directory. If you specify this as the first positional argument, the tag (-source) is optional. You may abbreviate the option tag to -s.

- **-destination *destinationJarFile***

  Specifies the destination jar or zip file, which will contain the desired subset of the files in the source jar or zip file. If you do not specify the path name, the file is created in the current directory. You may abbreviate the option tag to -d. The default name is generated by appending "Small" to the source file name. For example, if the source file is myfile.jar, then the default destination file would be myfileSmall.jar.

- **-requiredFile *jarEntry1[,jarEntry2[...]]***

  The files in the source jar or zip file that are to be copied to the destination. Entries are separated by commas (no spaces). The specified files, along with all of their dependencies, will be considered required. Files are specified in jar entry name syntax, such as com/ibm/as400/access/DataQueue.class. You may abbreviate the option tag to -rf.

- **-additionalFile *file1[,file2[...]]***

  Specifies additional files (not included in the source jar or zip file) which are to be copied to the destination. Entries are separated by commas (no spaces). Files are specified by either their absolute path, or their path relative to the current directory. The specified files will be included, regardless of the settings of other options. You may abbreviate the option tag to -af.

- **-additionalFilesDirectory *baseDirectory***

  Specifies the base directory for additional files. This should be the parent directory of the directory where the package path starts. For example, if file foo.class in package com.ibm.mypackage is located in directory C:\dir1\subdir2\com\ibm\mypackage\, then specify base directory C:\dir1\subdir2. You may abbreviate the option tag to -afd. The default is the current directory.

- **-package *package1[,package2[...]]***

  The packages that are required. Entries are separated by commas (no spaces). You may abbreviate the option tag to -p. Package names are specified in standard syntax, such as com.ibm.component.

  Note: The specified packages are simply included in the output. No additional dependency analysis is done on the files in a package, unless they are explicitly specified as required files.

- **-extract *[baseDirectory]***

  Extracts the desired entries of the source jar or zip file into the specified base directory, without generating a new jar or zip file. This option enables the user to build up a customized jar or zip file empirically, based on the requirements of their particular application. When this option is specified, -additionalFile, -additionalFilesDirectory, and -destination are ignored. You may abbreviate the option tag to -x. By default, no extraction is done. The default base directory is the current directory.

- **-split *[splitSize]***

  Splits the source jar or zip file into smaller jar or zip files. No zip entries are added or excluded. The entries in the source jar or zip file are simply distributed among the destination jar or zip files. The split size is in units of kilobytes (1024 bytes), and specifies the maximum size for the destination files. The destination files are created in the current directory, and are named by appending integers to the source file name. Any existing files by the same name are overwritten. For example, if the source jar file is myfile.jar, the destination jar files would be myfile0.jar, myfile1.jar, and so on. When this option is specified, all other options except -source and -verbose are ignored. You may abbreviate the option tag to -sp. The default split size is 2 MB (2048 KB).

- **-verbose**

  Causes progress messages to be printed to System.out. You may abbreviate the option tag to -v. The default is non-verbose.

10.1.2 JarMaker Example

In this example, the source jar file is named myJar.jar, and is in the current directory. To create a jar file that contains only the classes mypackage.MyClass1 and mypackage.MyClass2, along with their dependencies, enter:

```
java utilities.JarMaker -source myJar.jar -requiredFile
       mypackage/MyClass1.class,mypackage/MyClass2.class
```

Alternatively, the same function can be done with a Java program as shown here:

```
import utilities.JarMaker;
// Set up the list of required files.
Vector classList = new Vector ();
classList.addElement ("mypackage/MyClass1.class");
classList.addElement ("mypackage/MyClass2.class");
JarMaker jm = new JarMaker ();
jm.setRequiredFiles (classList);
// Make a new jar file, that contains only MyClass1, MyClass2...
File sourceJar = new File ("myJar.jar");
File newJar = jm.makeJar (sourceJar);  // smaller jar file
```

10.1.3 AS400ToolboxJarMaker

The AS400ToolboxJarMaker class is used to generate a smaller jar or zip file from the shipped AS/400 Toolbox for Java jar or zip file. In addition, you can use the AS400ToolboxJarMaker to:

- Extract desired files from a jar or zip file
- Split a jar or zip file into smaller jar or zip files

You can use AS400ToolboxJarMaker in a program, or you can run AS400ToolboxJarMaker as a command line program, as shown here:

```
java utilities.AS400ToolboxJarMaker [ options ]
```

AS400ToolboxJarMaker extends the functionality of JarMaker by allowing the user to specify one or more AS/400 Toolbox for Java component, language, or CCSID. You specify whether to include or exclude any JavaBean files that are associated with the specified components.

You must specify at least one of the following options:

```
-requiredFile
-additionalFile
-package
-extract
-split
-component
-language
-ccsid
-ccsidExcluded
```

If these options are specified multiple times in a single command string, only the final specification applies:

```
-source
-destination
-additionalFilesDirectory
-extract
-split
-languageDirectory
```

Other options have a cumulative effect when you specify them multiple times in a single command string.

10.1.3.1 Options

These options control what AS400ToolboxJarMaker will do:

- **-source *sourceJarFile***

  Specifies the source jar or zip file from which to derive the destination jar or zip file. If you specify a relative path, the path is assumed to be relative to the current directory. If this option is specified as the first positional argument, the tag (-source) is optional. You may abbreviate the option tag to -s. The default is jt400.jar, in the current directory.

- **-destination *destinationJarFile***

  Specifies the destination jar or zip file, which will contain the desired subset of the files in the source jar or zip file. If a path name is not specified, the file is created in the current directory. You may abbreviate the option tag to -d. The default name is generated by appending "Small" to the source file name. For example, if the source file is myfile.jar, then the default destination file would be myfileSmall.jar.

- **-requiredFile *jarEntry1[,jarEntry2[...]]***

  The files in the source jar or zip file that are to be copied to the destination. Entries are separated by commas (no spaces). The specified files, along with all of their dependencies, will be considered required. Files are specified in jar entry name syntax, such as com/ibm/as400/access/DataQueue.class. You may abbreviate the option tag to -rf.

- **-additionalFile *file1[,file2[...]]***

  Specifies additional files (not included in the source jar) that are to be copied to the destination. Entries are separated by commas (no spaces). Files are specified by either their absolute path, or their path relative to the current directory. The specified files will be included, regardless of the settings of other options. You may abbreviate the option tag to -af.

- **-additionalFilesDirectory *baseDirectory***

  Specifies the base directory for additional files. This should be the parent directory of the directory where the package path starts. For example, if file foo.class in package com.ibm.mypackage is located in directory C:\dir1\subdir2\com\ibm\mypackage\, then specify base directory C:\dir1\subdir2. You may abbreviate the option tag to -afd. The default is the current directory.

- **-package *package1[,package2[...]]***

  The packages that are required. Entries are separated by commas (no spaces). You may abbreviate the option tag to -p. Package names are specified in standard syntax, such as com.ibm.component.

  Note: The specified packages are simply included in the output. No additional dependency analysis is done on the files in a package, unless they are explicitly specified as required files.

- **-extract *[baseDirectory]***

  Extracts the desired entries of the source jar into the specified base directory, without generating a new jar or zip file. This option enables the user to build up a customized jar or zip file empirically, based on the requirements of their particular application. When this option is specified, -additionalFile, -additionalFilesDirectory, and -destination are ignored. You may abbreviate the option tag to -x. By default, no extraction is done. The default base directory is the current directory.

- **-split *[splitSize]***

  Splits the source jar or zip file into smaller jar or zip files. No zip entries are added or excluded. The entries in the source jar or zip file are simply distributed among the destination jar or zip files. The split size is in units of kilobytes (1024 bytes), and specifies the maximum size for the destination files. The destination files are created in the current directory, and are named by appending integers to the source file name. Any existing files by the same name are overwritten. For example, if the source jar file is myfile.jar, then the destination jar files would be myfile0.jar, myfile1.jar, and so on. When this option is specified, all other options except -source and -verbose are ignored. You may abbreviate the option tag to -sp. The default split size is 2 MB (2048 KB).

- **-verbose**

  Causes progress messages to be printed to System.out. You may abbreviate the option tag to -v. The default is non-verbose.

- **-component componentID1[,componentID2[...]]**

  The AS/400 Toolbox for Java components that are required. Entries are separated by commas (no spaces), and are case insensitive. You may abbreviate the option tag to -c. See the list of components that are supported by AS/400 Toolbox for Java.

- **-beans**

  Causes inclusion of all Java Beans files (classes, gifs) that are directly associated with the specified components. This option is valid only if -component is also specified. You may abbreviate the option tag to -b. The default is no Beans.

- **-language *language1[,language2[...]]***

  Specifies the desired languages for the messages produced by the Toolbox classes. Entries are separated by commas (no spaces). The languages are identified by their Java locale name, such as fr_CA (for Canadian French).

  Note: The shipped jt400.jar file contains only English messages. You may abbreviate the option tag to -l. By default, only English messages are included.

- **-languageDirectory *baseDirectory***

  Specifies the base directory for additional Toolbox language files. The path below this directory should correspond to the package name the language files. For example, if the desired MRI files are located in directories /usr/myDir/com/ibm/as400/access/ and /usr/myDir/com/ibm/as400/vaccess/, then the base directory should be set to /usr/myDir. You may abbreviate the option tag to -ld. By default, language files are searched for relative to the current directory.

- **-ccsid *ccsid1[,ccsid2[...]]***

  Specifies the CCSIDs whose conversion tables should be included. Conversion tables for other CCSIDs will not be included. Entries are separated by commas (no spaces). You may abbreviate the option tag to -cc.

By default, all CCSIDs are included. See the list of CCSIDs and encoding that are supported by AS/400 Toolbox for Java.

- **-ccsidExcluded *ccsid1[,ccsid2[...]]***

  Specifies the CCSIDs whose conversion tables should not be included. Entries are separated by commas (no spaces). If a CCSID is specified on both the -ccsid and -ccsidExcluded, it is included, and a warning message is sent to **System.err**. You may abbreviate the option tag to -cx. By default, all shipped CCSIDs are included. See the list of CCSIDs and encoding that are supported by AS/400 Toolbox for Java.

10.1.4 Example Usage

To create a jar file that contains only those files needed by the AS/400 Toolbox for Java CommandCall and ProgramCall classes, enter the following command (assuming you are in the same directory as the jt400.jar file):

```
java utilities.AS400ToolboxJarMaker -component
          CommandCall,ProgramCall
```

The resulting jt400Small.jar file is 630K. It includes all the supporting classes required. This drastically improves the amount of time required to download an applet that only uses these components.

10.2 Securing Applications with SSL

Prior to discussing SSL, it is useful to understand some elements of Internet security and how they relate to the AS/400 system. The remainder of this chapter contains the following information:

- An overview of the elements of transaction security available on the Internet
- A high-level description of the Secure Sockets Layer (SSL) protocol
- How to use Digital Certificate Manager (DCM) on AS/400 to create an intranet certificate authority (CA) and server certificates
- How to apply the server certificate to the Host Servers
- How to receive a CA certificate into a Java class
- How to load the SSL classes into VisualAge for Java
- How to modify an existing application to use SSL
- How to verify that SSL is being used

10.2.1 Internet Security Elements

There is no one single answer to Internet security. Some people believe that by installing a firewall that their company network is safe.

Will a firewall alone shield your company from any inappropriate Internet access? No, security is not a matter of a single device or procedure. Security is a concept, a set of different security measures that are selected based on the needs of a specific installation.

Therefore, it is essential to discuss first the type of Internet security you need to achieve. Chances are that it is not just a firewall.

Figure 322 on page 372 shows some of the elements that you can address when designing a total security solution for a company. However, this redbook is not intended as an aid for deciding upon your company security policy. If you need help in forming a security policy for you company, it is better that you seek professional advice or services, such as the IBM SecureWay services.

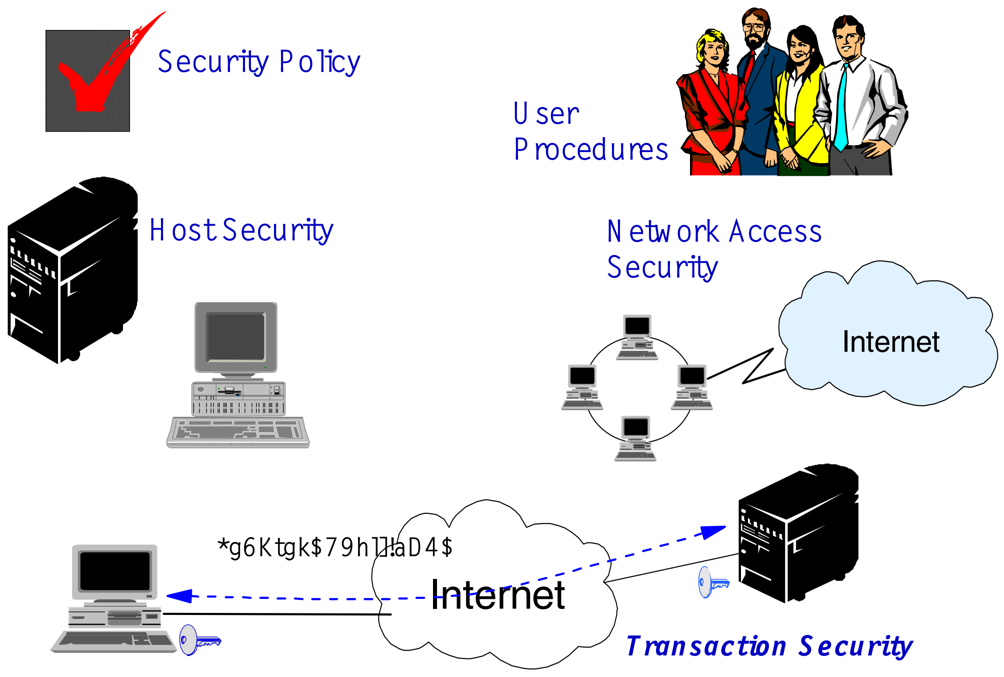


Figure 322. Internet Security Elements

First of all, a policy established by high-level management indicates how your company wants to deal with the Internet. It should define what level of security you want to achieve and how valuable or sensitive the different types of information you posses are. Various Internet security features, such as cryptography or host system security functions, can help you to implement what you design.

In addition, users need to be educated to follow and maintain the implemented security procedures, as well as to observe specific rules when acting as Internet clients.

10.2.2 Transaction Security and Secure Sockets Layer

Transaction security includes several basic elements, such as:

- Confidentiality/privacy
- Integrity
- Authentication
- Accountability

SSL is the Secure Sockets Layer protocol defined by Netscape Communications Corporation. It provides a private channel between a client and server that ensures privacy of data, authentication of session partners and message integrity.

Digital certificates are used for session partner authentication. Server authentication is common. Client authentication is not yet common, but it is growing in popularity.

Keys, are the base for end-to-end information encryption. Figure 323 provides a high-level view of SSL and transaction security.

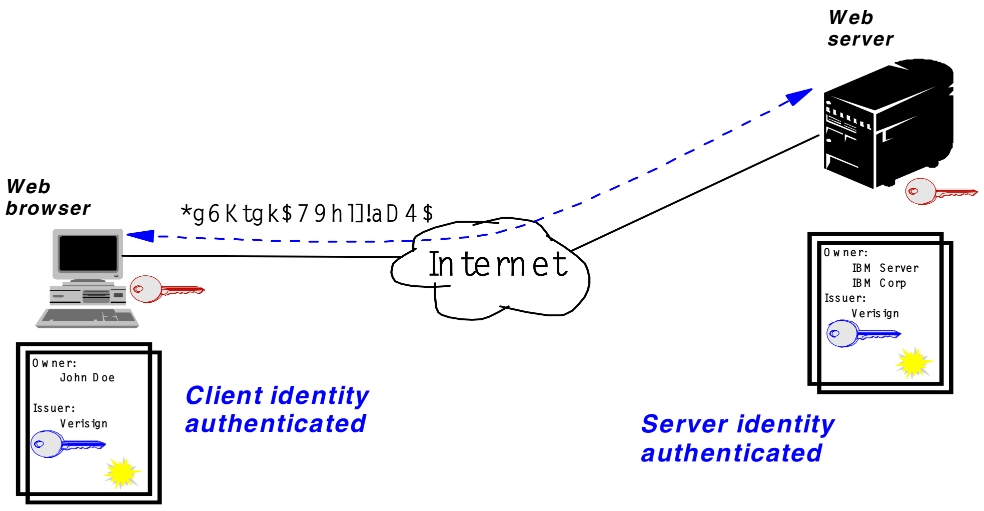


Figure 323. Transaction Security

You must re-write TCP/IP applications to use SSL. Primarily, SSL is used by HTTP (HTTPS) for Web browsing. In OS/400 V4R3 Directory Services Server (LDAP) is SSL enabled. With OS/400 V4R4, the Host Server applications have been enabled by the licensed products as shown in Table 51.

Table 51. AS/400 SSL Licensed Program Products

Installed Cryptographic Access Provider Product	Required Client Encryption Product	Key length
5769AC1	5769CE1	40 bit
5769AC2	5769CE2	56 bit
5769AC3	5769CE3	128 bit

When you order OS/400, you are shipped the appropriate version of the 5769ACx and 5769CEx products in accordance with US export laws.

10.2.2.1 Confidentiality

When a packet travels across a standard network, it is possible to use a *packet sniffer* to passively read the message. This means packets travelling a network can be read without either the sender or receiver ever knowing. To overcome this, messages should be encrypted to assure confidentiality.

Confidentiality means that the contents of the messages remain private as they pass through the Internet. Without confidentiality, your computer broadcasts the message to the network, similarly to shouting the information across a crowded room. *Encryption* ensures confidentiality.

10.2.2.2 Integrity

For example, you may want to know if the data received is the same as the data that was sent. You can determine this through two possible solutions: *digital signature (or hashing)* and *encryption*.

The sending system calculates a hash value based on the message being sent. The hash value is appended to the transmission. The receiving system uses the same calculation to generate a value. The receiving system then compares the calculated value with the received value. If the values are different, then it assumes that the data changed. To provide more bullet proof security, the message should first be encrypted using an appropriate encryption algorithm.

Integrity means that the messages are not altered while being transmitted. If a router or other network device inserts, deletes, or garbles the message as it passes by, the receiver would detect the modification. Without integrity, you have no guarantee that the message you sent matches the message that was received. Encryption and *digital signature* ensure integrity.

10.2.2.3 Authenticity

Consider the scenario where you want to know who is at the other end of a Web site to test its authenticity. One way to find out is through the use of digital certificates and digital signatures (see Figure 324).

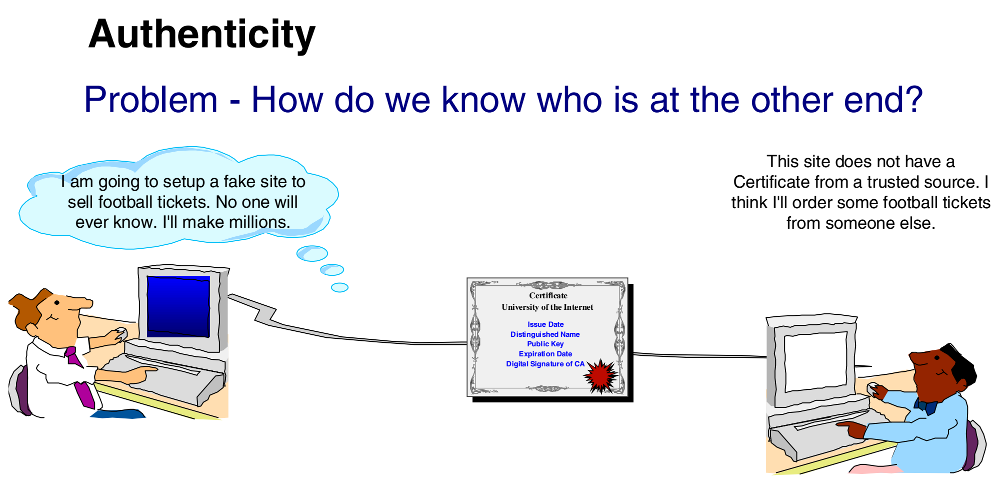

Authenticity
Problem - How do we know who is at the other end?
I am going to setup a fake site to sell football tickets. No one will ever know. I'll make millions.
This site does not have a Certificate from a trusted source. I think I'll order some football tickets from someone else.
Certificate
University of the Internet
Issue Date
Distinguished Name
Public Key
Expiration Date
Digital Signature of CA

Figure 324. Verifying Identity — Digital Certificates and Digital Signatures

Authenticity means that you know who you are talking to and that you trust that person. Without authenticity, you have no way to be sure that anyone is who they say they are. *Authentication* through digital certificates and digital signatures ensure authenticity.

There are two ways in which the server uses authentication:

- Digital signature
- Digital certificates

Digital signature ensures accountability. But how do you know if the person sending you a message is who they say they are?

To ensure accountability look at the sender's *digital certificate*. A public key certificate is issued by a trusted third party known as the *certifying authority* (CA). The browser and server exchange information, including their public key certificate. SSL uses the information to identify and authenticate the sender of the certificate.

You can think of a digital certificate as being like a credit card with your picture on it and a picture of the bank president with his arm around you. A merchant will trust you more because not only do you look like the picture on the credit card, but the bank president trusts you, too.

You base your trust for the authenticity of the sender on whether you trust the third party (a person or agency) that certified the sender. The third party or certification authority (CA) issues digital certificates.

How can you ensure that the person sending the message is really trustworthy? Let us use an example to illustrate this point.

If you wake up one day feeling ill, you may decide to visit a doctor. You can select a doctor from your phone book and go to his or her office for a visit. Once you get to the office, how can you be sure that the person about to examine you is really a doctor? After all, you have never met this person before. They may look like a doctor and act like a doctor, but who's to say that this person has successfully completed all of the training necessary to become a doctor?

You need certification by a trusted third party to reassure you that this person really is a doctor. The doctor probably has a diploma on the wall stating that they have successfully completed their training. If the diploma is from a well known school, you would probably be reassured that you are about to be examined by a real doctor. But what if the diploma is from the medical school of a correspondence school whose name you do not recognize? You may not be so reassured.

Authentication works the same way. Trusted third parties verify that the server really is who it claims to be. This verification is provided with a digital certificate—the digital equivalent of your doctor's diploma hanging on the wall. You base your trust for the authenticity of the server on whether you trust the third party that certified the server—the school that issued the diploma. That third party is called a *certifying authority* (CA).

The term *trusted root* is given to a trusted certifying authority (CA) on your server. A *trusted root key* is the key belonging to the CA.

You can use authentication server to client (Server authentication) or client to server (Client authentication). Server authentication is described above. The clients authenticate the servers. With client authentication, the client is authenticated by the server. For example, you may use client authentication if a server contained hospital patient information, to verify that the client attempting to access the data is really who he said he is before allowing him access to patient records.

10.2.2.4 Accountability

If you need to prove that specific transactions took place, you need to implement some form of accountability.

To prove that a transaction occurred, combine all the previous techniques. First, calculate the hash code of the data to assure data integrity. The data is then encrypted using the keys derived from the public key exchange. This assures the identity of the session partners. This is used in combination with a time stamp in the data to provide a log of the transactions.

Accountability means that both sender and receiver agree that the exchange took place. Without accountability, the target application user can easily deny that the data arrived. You can use digital signatures to ensure accountability. However, *accountability is not part of the SSL protocol* since it requires an application to perform the tasks described.

10.3 Digital Certificates and Certificate Authority

A digital certificate identifies a user or a system, and is required before you can use SSL. Once a server has a digital certificate, SSL-enabled browsers, such as the Netscape Navigator, can communicate securely with the server using SSL. A digital certificate consists of:

- Owner’s distinguished name
- Owner’s public key
- Digital signature of certificate authority (CA)
- Name of the CA
- Issue date of certificate
- Certificate expiration date
- Serial number

Plus, digital certificates have the following characteristics:

- Digital certificates are digital documents that validate the identity of the certificate's owner.
- There are three types of digital certificates: CA, server, and client certificates.
- Digital certificates contain a public key, which binds it to an identity.
- Digital certificates are created by trusted third parties called certificate authorities (CA).
- Digital certificates can be distributed freely.
- A digital signature in the digital certificate prevents tampering.

A certificate authority (CA) issues a digital certificate. CAs are entities that are trusted to properly issue certificates and have controls in place to prevent fraudulent use. They are the equivalent to the Department of Motor Vehicles for a driver's license. An individual may have many certificates from different CAs just as we may have many forms of identification (passport, credit card, gym membership card, and so on). If you trust a CA, you can be reasonably assured that any certificate they issue properly represents the individual that is holding it. The CA charges a fee for issuing a certificate.

- CAs broadcast their public key and distinguished name.
- People add them as trusted root keys to Web servers and browsers.
- Your server trusts anyone who has a certificate from that CA.
- There are several common CAs in the marketplace.
- Servers and browsers are shipped with several default trusted root keys and more can be added as needed.

Some examples of universally recognized Internet CAs include:

- Thawte Consulting
- VeriSign, Inc.
- IBM World Registry

For testing purposes or for applications that will be used exclusively in an intranet environment, you may issue digital certificates using an intranet certificate authority. The AS/400 system with Digital Certificate Manager (DCM) can act as an intranet certificate authority. However, be aware that the real cost of being your own CA can escalate. It is often cheaper to purchase a valid certificate from a well known certificate authority.

For secure communications, the receiver must trust the CA that issued the certificate, whether the receiver is a browser or a server. Anytime a sender signs a message, the receiver must have the corresponding CA certificate and public key designated as a trusted root key.

10.4 AS/400 Implementation of Digital Certificate Management

You can configure your AS/400 system as an intranet certificate authority. Digital Certificate Manager (DCM) is a Web browser-based administration facility that allows you to create, manage, and use certificates within an enterprise and with partners of an enterprise. You can use DCM to request digital certificates from such Internet CA as VeriSign and Thawte.

DCM allows you to create your own intranet CA. You can then use the CA to dynamically issue digital certificates to servers and users (client certificates) on your intranet. When you create a server certificate, DCM automatically generates the private key and public key for the certificate. You can also use DCM to register and use digital certificates from Verisign or other commercial organizations on your intranet or the Internet.

DCM is option 34 of OS/400 (5769-SS1). You must install this option to use DCM. DCM is a link in the AS/400 Tasks page, which runs in the *ADMIN HTTP server instance. Therefore, you must install and use IBM HTTP Server for AS/400 (5769-DG1) to access DCM. In addition, you must install IBM Cryptographic Access Provider licensed program (5769-AC1, AC2, or AC3) to create certificate keys. These cryptographic products determine the maximum key length permitted for cryptographic algorithms on your AS/400 system. Government export or import regulations determine which version is available in your country.

Note: To use all the options available in DCM, you must have *SECOFR and *SECADM authority.

To access DCM, click on the hyperlink for **Digital Certificate Manager** from the AS/400 Tasks Page. When using Digital Certificate Manager, you can click the *Help* button on any page at any time to access on-line help.

10.4.1 Configuring a Digital Certificate Environment

You can use your AS/400 system to configure a digital certificate environment. Once the environment is configured, you can use certificates to enable SSL communication.

Perform the following series of steps to configure an intranet digital certificate environment using the AS/400 system as a CA:

1. Use DCM to create an intranet CA in one or more AS/400 systems.
2. Using DCM, use the intranet CA to issue server certificates, which can be used in the local server (same AS/400 system where the CA is configured) or exported to a remote server.

3. For the clients to recognize and trust the server certificates issued by the intranet CA, the client must download and designate the CA certificate as a trusted root.

10.5 Using a Self-Signed Certificate for SSL

This section describes how to create a self-signed certificate using your AS/400 system as an intranet CA.

Because self-signed certificates are not recognized by client applications as coming from a trusted third party, you should not use them in customer transaction situations over the Internet. Use them only on your test and development systems, and for demonstration purposes. You can also use a self-signed certificate for intranet applications.

To obtain a self-signed certificate, complete the following steps:

1. Create an intranet CA.
2. Create a server certificate with your intranet CA.
3. Configure your servers to use the server certificate.

10.5.1 Creating an Intranet Certificate Authority

DCM allows you to create your own intranet CA in your AS/400 system and use it to issue server and client certificates for testing purposes or applications within your organization.

This section outlines the steps that you must perform to create a CA on your AS/400 system. You only need to perform these steps if the system administrator did not previously create an intranet CA and if you wish to use your AS/400 system to issue intranet server certificates.

To create an intranet CA in your AS/400 system, follow these steps:

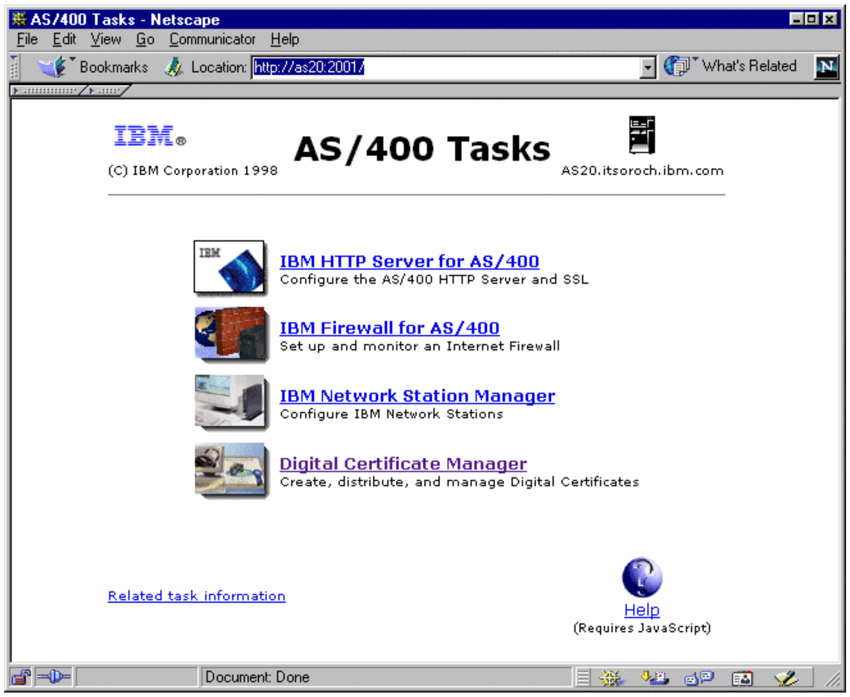
1. Start the HTTP *ADMIN server on your AS/400 system. From the command line, enter this command:

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
```

2. Access the AS/400 Tasks page from your browser by entering this URL:

```
http://System_name:2001
```

3. You are prompted to enter your user name and password. Sign on with a user that has *SECOFR and *SECADM authority. The AS/400 Tasks Page is displayed as shown in Figure 325 on page 379.


AS/400 Tasks - Netscape

File Edit View Go Communicator Help

Bookmarks Location: http://as20:2001/ What's Related

AS/400 Tasks

(C) IBM Corporation 1998

AS20.itsoroch.ibm.com

IBM HTTP Server for AS/400
Configure the AS/400 HTTP Server and SSL

IBM Firewall for AS/400
Set up and monitor an Internet Firewall

IBM Network Station Manager
Configure IBM Network Stations

Digital Certificate Manager
Create, distribute, and manage Digital Certificates

Related task information

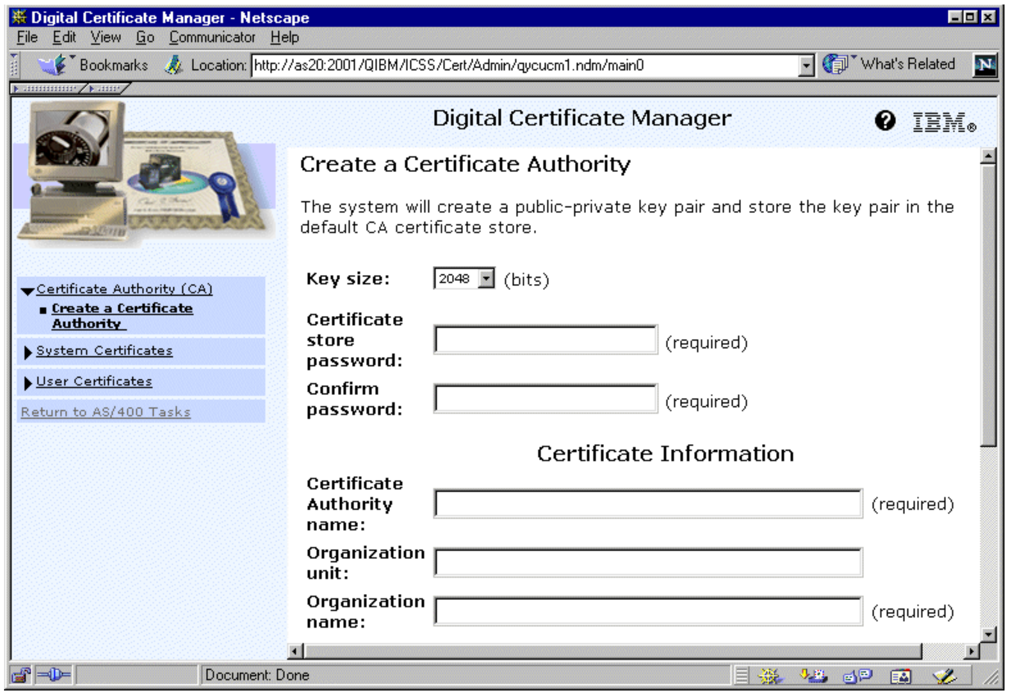
Help
(Requires JavaScript)

Document: Done

Figure 325. AS/400 Tasks Page

4. Click **Digital Certificate Manager.**
5. Click **Certificate Authority (CA).**
6. Click **Create a Certificate Authority.**

   Note: If a Certificate Authority (CA) was previously created on your system, the Create a Certificate Authority link is not displayed.
7. Complete the Create a Certificate Authority form as shown in Figure 326 on page 380. Replace the field values as appropriate with your organization information.


Digital Certificate Manager - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://as20:2001/QIBM/ICSS/Cert/Admin/qycucm1.ndm/main0
What's Related
Certificate Authority (CA)
Create a Certificate Authority
System Certificates
User Certificates
Return to AS/400 Tasks
Document: Done

Digital Certificate Manager

Create a Certificate Authority

The system will create a public-private key pair and store the key pair in the default CA certificate store.

Key size: 2048 (bits)

Certificate store password: (required)

Confirm password: (required)

Certificate Information

Certificate Authority name: (required)

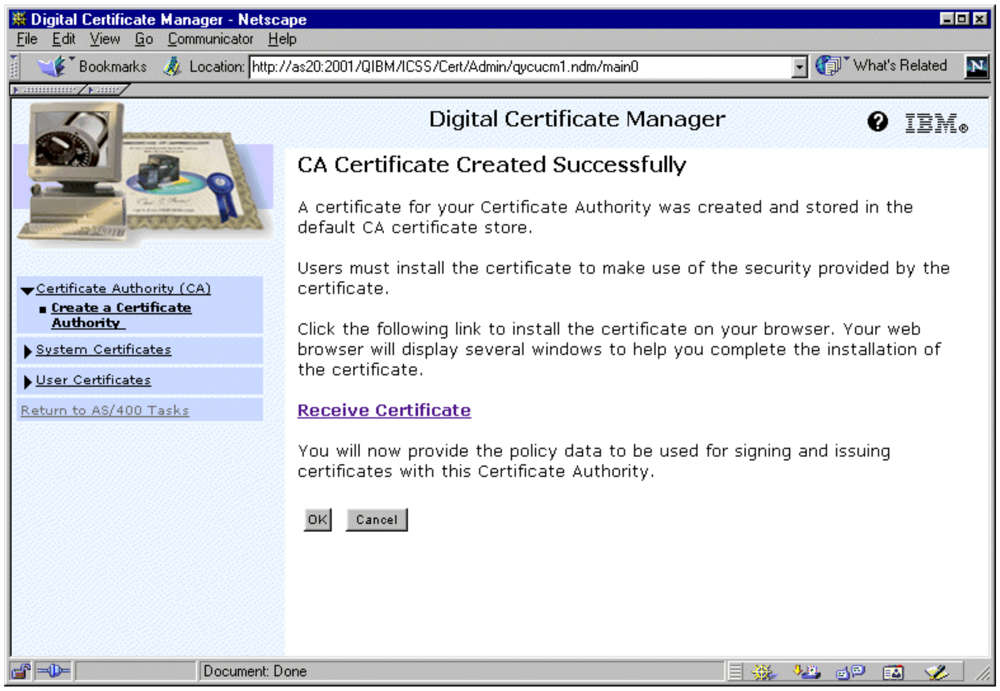
Organization unit:

Organization name: (required)

Figure 326. Create an Intranet Certificate Authority

8. Click **OK**.
9. After DCM processes the form, it stores a copy of the CA certificate in the CA default key ring file.

   DCM displays the page shown in Figure 327.


Digital Certificate Manager - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://as20:2001/QIBM/ICSS/Cert/Admin/qycucm1.ndm/main0
What's Related
Certificate Authority (CA)
Create a Certificate Authority
System Certificates
User Certificates
Return to AS/400 Tasks
OK
Cancel
Document: Done

Digital Certificate Manager

CA Certificate Created Successfully

A certificate for your Certificate Authority was created and stored in the default CA certificate store.

Users must install the certificate to make use of the security provided by the certificate.

Click the following link to install the certificate on your browser. Your web browser will display several windows to help you complete the installation of the certificate.

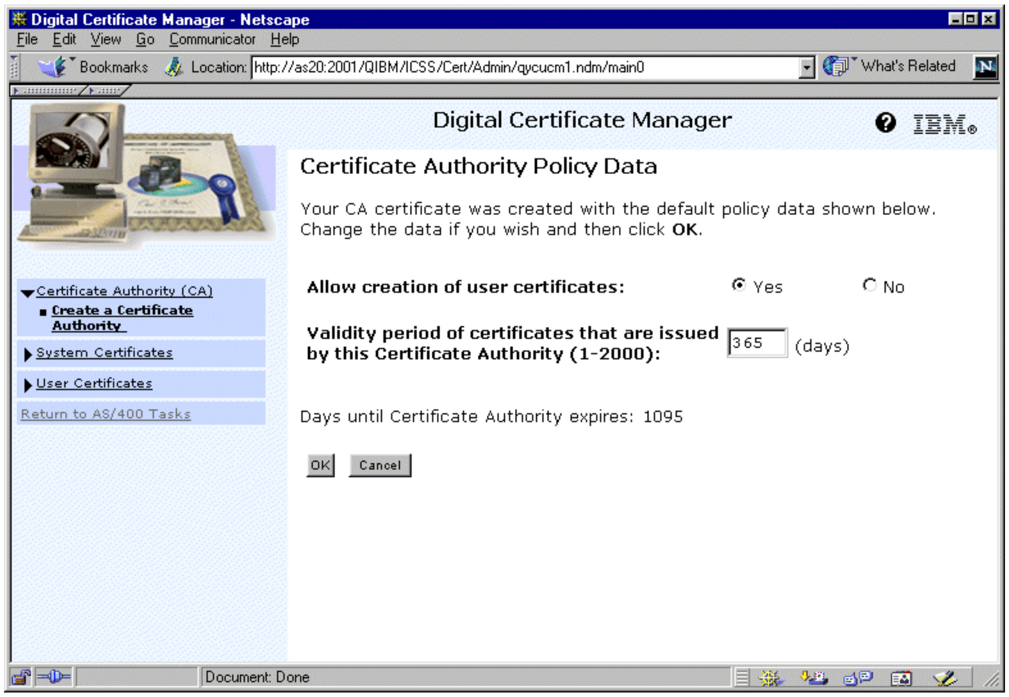
Receive Certificate

You will now provide the policy data to be used for signing and issuing certificates with this Certificate Authority.

Figure 327. CA Certificate Created Successfully

10. Click **OK**.

Complete the CA Policy Data form to set the client certificate policy for your CA (see Figure 328). This is where you define whether your CA can issue and sign client certificates. If the CA can issue client certificates, indicate the length of time for which the certificates will be valid.


Digital Certificate Manager - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://as20:2001/QIBM/ICSS/Cert/Admin/qycucm1.ndm/main0
What's Related
Digital Certificate Manager
IBM
Certificate Authority (CA)
Create a Certificate Authority
System Certificates
User Certificates
Return to AS/400 Tasks
Document: Done

Certificate Authority Policy Data

Your CA certificate was created with the default policy data shown below. Change the data if you wish and then click **OK**.

Allow creation of user certificates: [✓ Yes] No

Validity period of certificates that are issued by this Certificate Authority (1-2000): 365 (days)

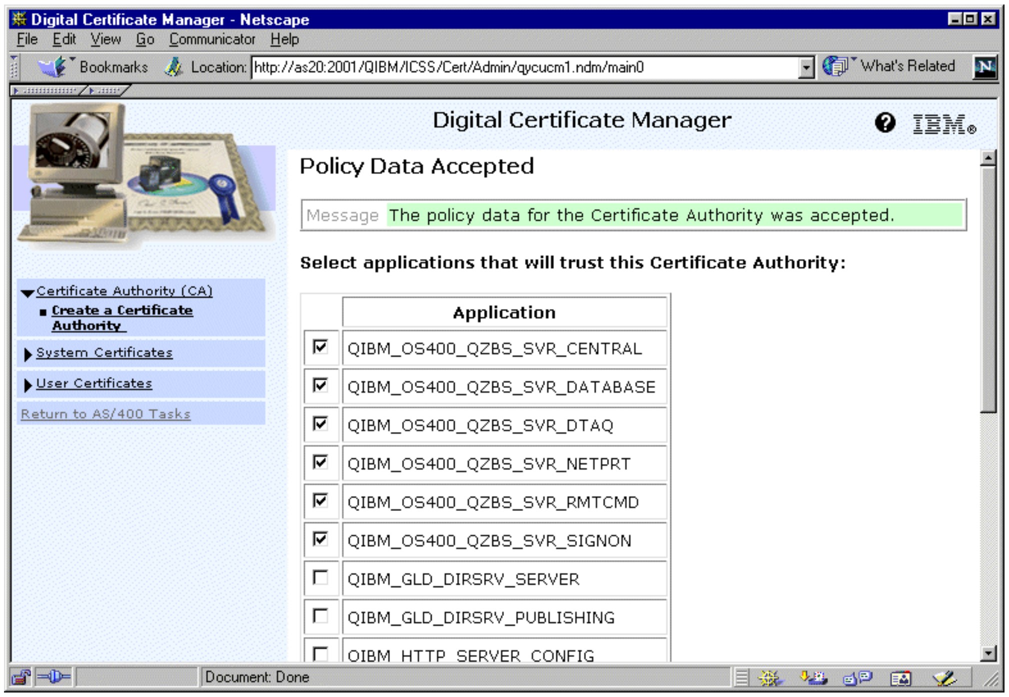
Days until Certificate Authority expires: 1095

OK Cancel

Figure 328. Certificate Authority Policy

11. Click **OK**.

12. The next display that appears invites you to allow certain applications to trust the newly created CA, as shown in Figure 329 on page 382. Select all of the applications that start **QIBM_OS400_** to enable SSL to all the host servers.


Digital Certificate Manager - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://as20:2001/QIBM/ICSS/Cert/Admin/qycucm1.ndm/main0
What's Related
Digital Certificate Manager
Certificate Authority (CA)
Create a Certificate Authority
System Certificates
User Certificates
Return to AS/400 Tasks
Document: Done

Policy Data Accepted

Message The policy data for the Certificate Authority was accepted.

Select applications that will trust this Certificate Authority:

	Application
☑	QIBM_OS400_QZBS_SVR_CENTRAL
☑	QIBM_OS400_QZBS_SVR_DATABASE
☑	QIBM_OS400_QZBS_SVR_DTAQ
☑	QIBM_OS400_QZBS_SVR_NETPRT
☑	QIBM_OS400_QZBS_SVR_RMTCMD
☑	QIBM_OS400_QZBS_SVR_SIGNON
☐	QIBM_GLD_DIRSRV_SERVER
☐	QIBM_GLD_DIRSRV_PUBLISHING
☐	QIBM_HTTP_SERVER_CONFIG

Figure 329. Setting the Created CA to Be Trusted by Host Servers

13. The message appears: `The policy data for the Certificate Authority was successfully changed`. At this point, you can continue to create a server certificate signed by your certificate authority. This allows server authentication by clients that use this system as a server.

10.5.2 Creating a Server Certificate with Your Intranet CA

Immediately after creating the intranet CA, DCM leads you to create a server (system) certificate. To use Secure Sockets Layer (SSL), your server must have a digital certificate. When you create a server certificate in DCM, the server certificate and keys are stored in the following default directory and file. Make sure it gets backed up:

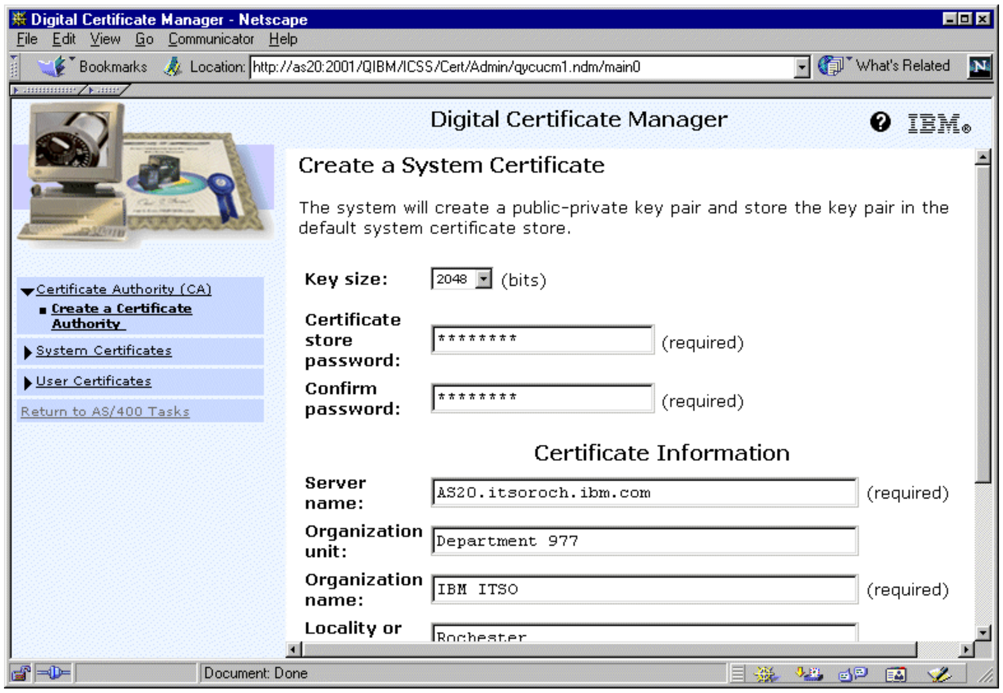
```
/QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB
```

Note: When you create a server certificate, DCM stores a copy of the CA certificate in the server key ring and designates it as a trusted root.

Complete the following steps:

1. Complete the Create a System Certificate form, as shown in Figure 330 on page 383, by replacing the field values with your organization information.

   The options for the key size are determined by the IBM Cryptographic Access Provider (5769-ACx) licensed program product installed in your system. This is the key size that is used to generate your public and private keys. The higher the value is, the more secure the conversation is.


Digital Certificate Manager - Netscape

File Edit View Go Communicator Help

Bookmarks Location: http://as20:2001/QIBM/ICSS/Cert/Admin/qycucm1.ndm/main0 What's Related

Digital Certificate Manager

- Certificate Authority (CA)
  - Create a Certificate Authority
- System Certificates
- User Certificates
- Return to AS/400 Tasks

Create a System Certificate

The system will create a public-private key pair and store the key pair in the default system certificate store.

Key size: 2048 (bits)

Certificate store password: ******** (required)

Confirm password: ******** (required)

Certificate Information

Server name: AS20.itsoroch.ibm.com (required)

Organization unit: Department 977

Organization name: IBM ITSO (required)

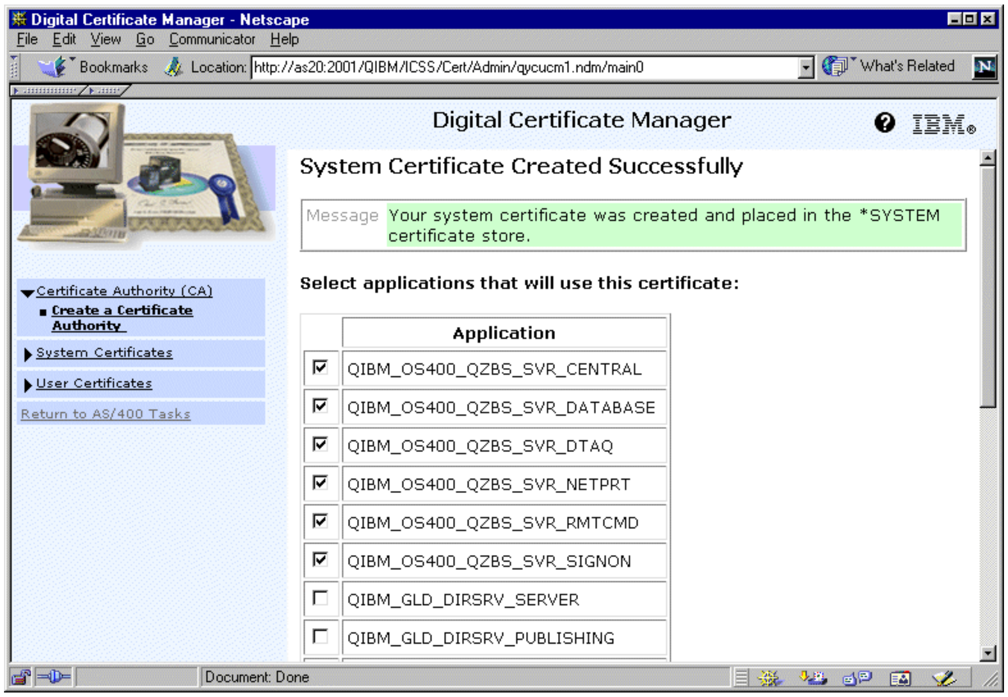
Locality or: Rochester

Document: Done

Figure 330. Create a System Certificate Page

By default, the system inserts the fully-qualified name of the AS/400 system into the system name field. Do not change this name. This is the name used to describe your server. You can give the server any name, although the fully qualified TCP/IP host name is usually used for the server name.

2. Click **OK**.
3. The Server Certificate Created Successfully page displays (see Figure 331 on page 384).


Digital Certificate Manager - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://as20:2001/QIBM/ICSS/Cert/Admin/qycucm1.ndm/main0
What's Related
Digital Certificate Manager
Certificate Authority (CA)
Create a Certificate Authority
System Certificates
User Certificates
Return to AS/400 Tasks
Document: Done

System Certificate Created Successfully

Message Your system certificate was created and placed in the *SYSTEM certificate store.

Select applications that will use this certificate:

	Application
✓	QIBM_OS400_QZBS_SVR_CENTRAL
✓	QIBM_OS400_QZBS_SVR_DATABASE
✓	QIBM_OS400_QZBS_SVR_DTAQ
✓	QIBM_OS400_QZBS_SVR_NETPRT
✓	QIBM_OS400_QZBS_SVR_RMTCMD
✓	QIBM_OS400_QZBS_SVR_SIGNON
☐	QIBM_GLD_DIRSRV_SERVER
☐	QIBM_GLD_DIRSRV_PUBLISHING

Figure 331. Server Certificate Created Successfully Page

4. From this page, apply the server certificate to the same servers to which you applied the CA certificate to (QIBM_OS400_xxx).
5. Click **Done**.
6. End and restart host servers by issuing the `ENDHOSTSVR *ALL` and `STRHOSTSVR *ALL` commands from OS/400.

10.6 Using a Server Certificate from an Internet CA

To conduct commercial business on the Internet, you should request your server certificate from an Internet certificate authority, such as VeriSign or Thawte, who are widely known by clients browsers and servers.

For your private Web network within your own company, university, or group, or for testing purposes, you can act as your own CA by using Digital Certificate Manager (DCM). Section 10.5, “Using a Self-Signed Certificate for SSL” on page 378, explains this procedure.

This section describes how to obtain a server certificate from an Internet certificate authority.

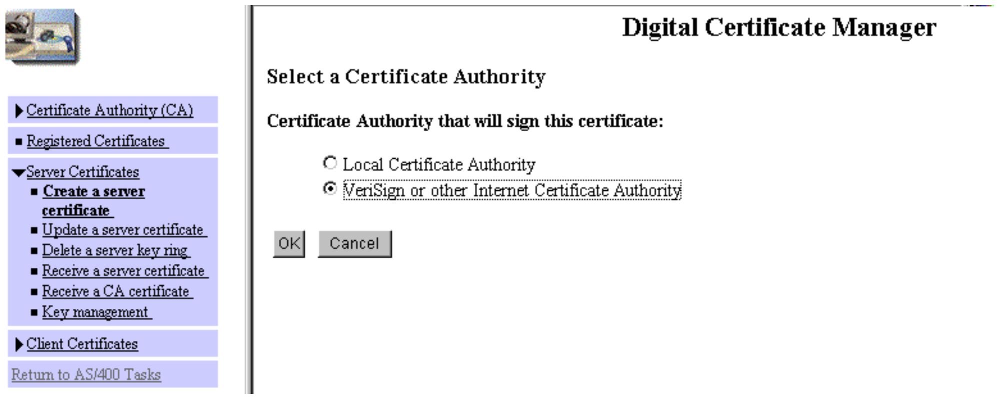
To use a server certificate issued by an Internet CA, you must complete these steps:

1. Request the server certificate from an Internet CA.
2. Receive a server certificate for this server.
3. Configure host servers to use this certificate.

To use SSL for secure Web serving, your server must have a digital certificate. You can use an intranet CA to issue a server certificate (see Section 10.5, "Using a Self-Signed Certificate for SSL" on page 378). Or, you can use an Internet CA.

When you choose to use an Internet CA to issue a server certificate, you must first request the certificate. To request a certificate, use the following steps:

1. From the Digital Certificate Manager (DCM) page, click **Server Certificates** in the left-hand frame to display an extended list of server tasks.
2. Click **Create a server certificate** from the list to display the Select a Certificate Authority page.
3. Select **VeriSign or other Internet Certificate Authority** as shown in Figure 332.



Digital Certificate Manager

Select a Certificate Authority

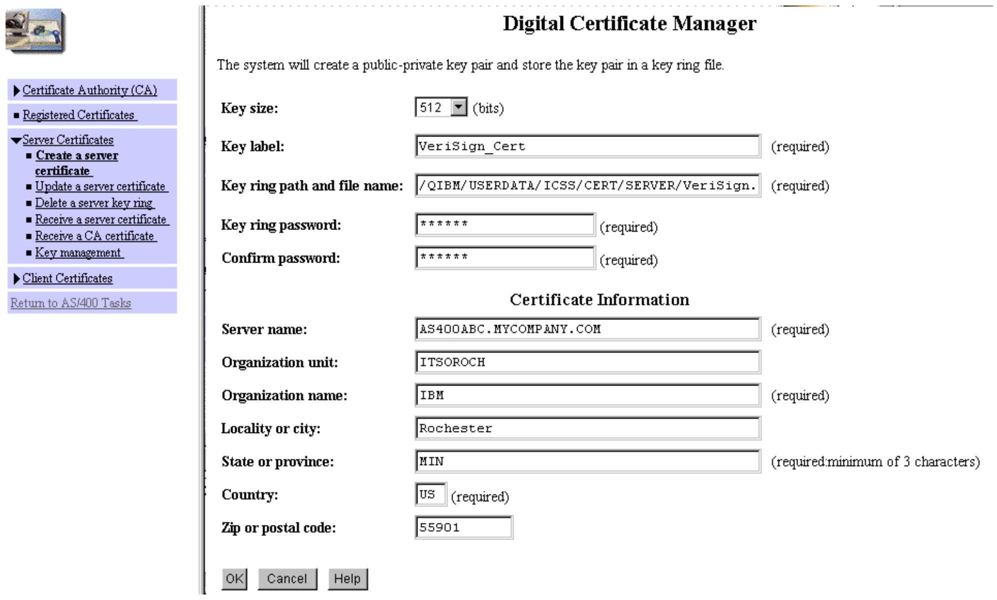
Certificate Authority that will sign this certificate:

Local Certificate Authority [✓ VeriSign or other Internet Certificate Authority]

Figure 332. Requesting a Certificate from VeriSign or Other Internet Certificate Authority

4. Click **OK** to display the Create a Server Certificate form.
5. Complete the Create a Server Certificate form as show in Figure 333 on page 386 replacing the field values with your organization information.

   The options for the key size are determined by the IBM Cryptographic Access Provider (5769-ACx) licensed program installed in your system. This is the key size that generates your public and private keys.



Digital Certificate Manager

The system will create a public-private key pair and store the key pair in a key ring file.

Key size:	512 (bits)	
Key label:	VeriSign_Cert	(required)
Key ring path and file name:	/QIBM/USERDATA/ICSS/CERT/SERVER/VeriSign.	(required)
Key ring password:	******	(required)
Confirm password:	******	(required)

Certificate Information

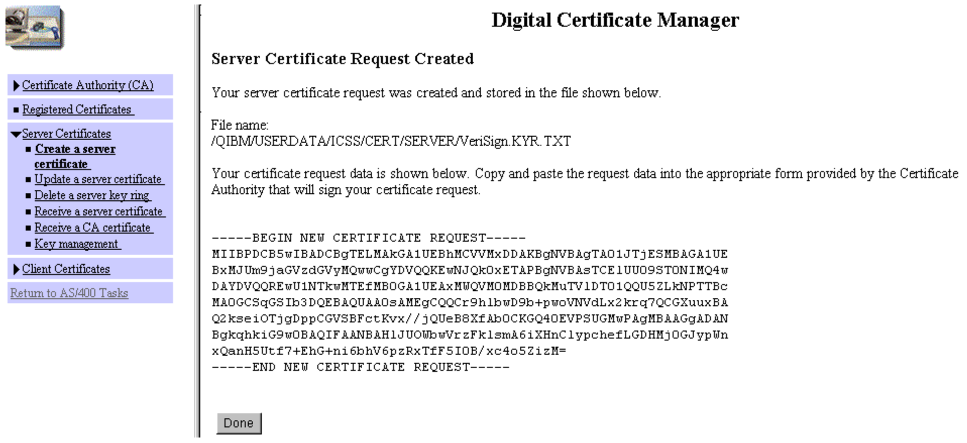
Server name:	AS400ABC.MYCOMPANY.COM	(required)
Organization unit:	ITSOROCH	
Organization name:	IBM	(required)
Locality or city:	Rochester	
State or province:	MIN	(required:minimum of 3 characters)
Country:	US	(required)
Zip or postal code:	55901	

Figure 333. Request a Server Certificate from an Internet CA

By default, the system inserts the fully qualified name of the AS/400 system into the system name field. Do not change this name. This is the name that describes your server. You can give the server any name, although the fully qualified TCP/IP host name is usually used for the server name.

6. Click **OK** to process the Create a Certificate Request form.

   You will receive the Server Certificate Request Created page as shown in Figure 334.



Digital Certificate Manager

Server Certificate Request Created

Your server certificate request was created and stored in the file shown below.

File name:
/QIBM/USERDATA/ICSS/CERT/SERVER/VeriSign.KYR.TXT

Your certificate request data is shown below. Copy and paste the request data into the appropriate form provided by the Certificate Authority that will sign your certificate request.

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBPDCB5wIBADCBgTELMAkGA1UEBhMCVVMxDDAKBgNVBAgTA01JTjESMBAGA1UE
BxMJUm9jaGVzdGVyMQwwCgYDVQQKEwNJQk0xETAPBgNVBAsTCElUU09ST0NIMQ4w
DAYDVQQREwU1NTkwMTEfMB0GA1UEAxMWQVM0MDBBQkMuTV1DT01QQU5ZLkNPTTBc
MA0GCSqGSIb3DQEBAQUAA0sAMEgCQQCr9hlbwD9b+pwoVNVdLx2krq7QCGXuuxBA
Q2kseiOTjgDppCGVSBFctKvx//jQUeB8XfAbOCKGQ40EVPSUGMwPAgMBAAGgADAN
BgkqhkiG9w0BAQIFAANBAHlJUOWbwVrzFklsmA6iXHnClypchefLGDHMj0GJypWn
xQanH5Utf7+EhG+ni6bhV6pzRxTfF5IOB/xc4o5ZizM=
-----END NEW CERTIFICATE REQUEST-----
```

Figure 334. Server Certificate Request Generated by DCM

Note: Do not click Done or close the browser yet. You need to cut and paste the certificate request when you submit the Certificate Signing Request to the Internet CA.

7. Copy the Server Certificate Request to your clipboard. Start at -----BEGIN NEW CERTIFICATE REQUEST----- and end at -----END NEW CERTIFICATE REQUEST-----. Click **Done** to close the page.

8. Follow your Internet CA procedures to paste the certificate request. For example, to request a certificate from VeriSign, follow the instructions that are described on the following Web site: http://www.verisign.com

   When the Internet CA is satisfied that you meet all of its requirements, it will e-mail the secure server certificate to you. You should receive it in three to five business days. Other certificates authorities have their own procedures.

10.6.1 Receiving a Server Certificate for This Server

After you receive the certificate from the Internet CA, you need to copy the signed server certificate to a text file that DCM can access when you perform the Receive server certificate task. Perform these steps:

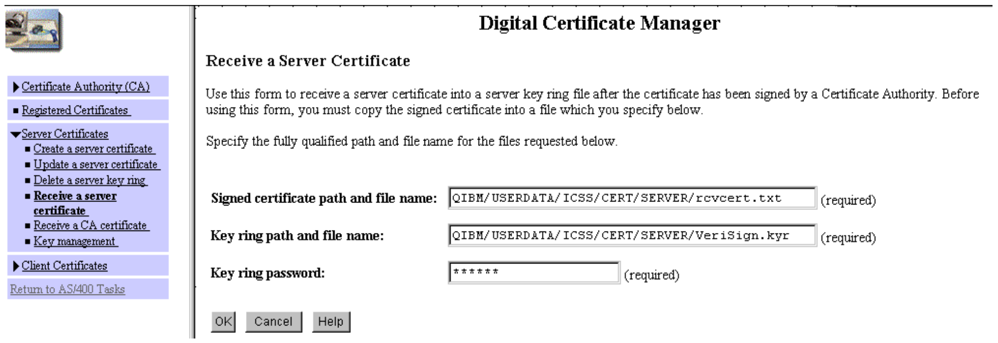
1. Copy the signed server certificate presented to you by the Internet CA to your clipboard.

   Include the -----BEGIN CERTIFICATE REQUEST----- and -----END CERTIFICATE REQUEST----- sections of the certificate

2. Paste the signed server certificate in your clipboard into an empty .txt file.
3. Save the file in your AS/400 system integrated file system. Use a mapped network drive and save the .txt file that contains the server certificate issued by the Internet CA in the following path (you can actually store this file where ever you want):

   `/QIBM/USERDATA/ICSS/CERT/SERVER/rcvcert.txt`

4. Back in DCM, click **Receive a server certificate** and complete the Receive a Server Certificate page (Figure 335).



Digital Certificate Manager

Receive a Server Certificate

Use this form to receive a server certificate into a server key ring file after the certificate has been signed by a Certificate Authority. Before using this form, you must copy the signed certificate into a file which you specify below.

Specify the fully qualified path and file name for the files requested below.

Signed certificate path and file name:	QIBM/USERDATA/ICSS/CERT/SERVER/rcvcert.txt	(required)
Key ring path and file name:	QIBM/USERDATA/ICSS/CERT/SERVER/VeriSign.kyr	(required)
Key ring password:	******	(required)

Figure 335. Receiving a Server Certificate Issued by an Internet CA

5. The Certificate Received page is displayed. You should now apply the certificate to the AS/400 servers. Select all the **QIBM_OS400**_xxx servers to enable SSL for Host Servers.

10.7 Downloading the SSL Java Packages

The core Java specification for 1.1.7 does not require the JDK/JVM to support SSL conversations. However, IBM recognizes the need to secure socket conversations using SSL, and therefore, provides two packages for enabling SSL in Java programs.

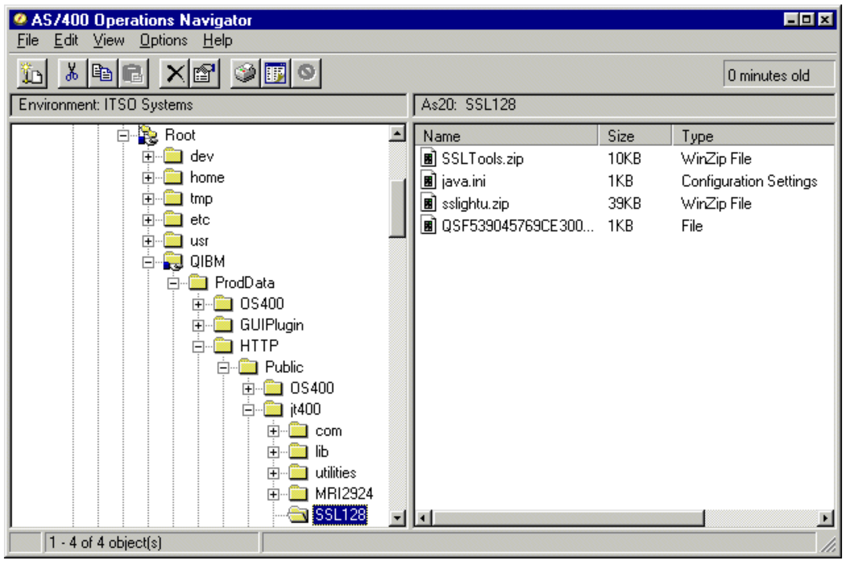
The following two archives are shipped with the 5769CEx products:

- **SSLTools.zip** — A utility set of classes used to create custom KeyRing class files
- **sslightu.zip** (with 5769CE3) or **sslightx.zip** (with 5769CE1 or 5769CE2) — This archive contains an SSL implementation for Java

Both of these archives are located in one of the following IFS directories:

- **5769CE1** — /QIBM/ProdData/HTTP/Public/jt400/SSL40
- **5769CE2** — /QIBM/ProdData/HTTP/Public/jt400/SSL56
- **5769CE3** — /QIBM/ProdData/HTTP/Public/jt400/SSL128

Using file transfer protocol (FTP), Client Access Shared Folders, Native SMB, or Operations Navigator (as shown in Figure 336), download the two zip files to your PC. In the subsequent sections, these zip files were downloaded to the C:\Programs\ directory.



Name	Size	Type
SSLTools.zip	10KB	WinZip File
java.ini	1KB	Configuration Settings
sslightu.zip	39KB	WinZip File
QSF539045769CE300...	1KB	File

Figure 336. The SSL Java Archives Viewed through Operations Navigator

When you deploy the program, you need to deploy the sslightx.zip or sslightu.zip archive with it. However, there is no need to deploy the SSLTools.zip archive.

10.8 Creating a KeyRing Class

Java uses a special class called KeyRing. This class stores additional CAs that your application will trust to communicate with through SSL.

The following certificates are built into the SSL packages shipped by IBM:

- VeriSign, Inc.
- Integrion Financial Network
- IBM World Registry
- Thawte Consulting
- RSA Data Security, Inc.

Therefore, if your server certificate was issued by one of the above certification authorities, you do not need to perform the subsequent steps.

If you are a working as your own CA, or have received a certificate from a certification authority that is not listed, you need to create a KeyRing.class file. The following steps help you achieve this:

1. Ensure that the AS/400 host servers have been restarted since assigning them a certificate. Use the `ENDHOSTSVR *ALL` and `STRHOSTSVR *ALL` commands if necessary.
2. Open a Java-enabled command prompt and add the SSLTools.zip and sslightu.zip or sslightx.zip classes to the CLASSPATH.
3. From your current directory, create a com/ibm/as400/access directory structure.
4. Enter the following command:

   ```
   java com.ibm.sslight.nlstools.keyrng com.ibm.as400.access.KeyRing connect
   systemname:9470
   ```

   This connects the SSL tool to port 9470, which is the default port for the Secure Socket version of the Central Server host server. You can use any SSL enabled socket, for example, 448 the secure data server port. For the AS/400 Toolbox for Java to communicate using SSL, the KeyRing class must reside in the com.ibm.as400.access package.
5. When prompted, enter `toolbox` for the password for the KeyRing class. This is the only value supported by the AS/400 Toolbox for Java for secure communications.
6. When the secure server is contacted, it causes the SSL tool to display the certificates known to the server and asks you to select one to add to the KeyRing class.

   Note: Select the CA certificate. Do not select the site certificate.
7. Once the `Done` message appears, you have successfully imported the CA certificate into the KeyRing class. The SSL Java classes can now use the class to communicate with any server that has a certificate issued by the imported CA.

   These steps are shown in Figure 337 on page 390.

```
C:>SET CLASSPATH=%CLASSPATH%;C:\Programs\SSLTools.zip
C:>SET CLASSPATH=%CLASSPATH%;C:\Programs\sslightu.zip
C:>md com
C:>md com\ibm
C:>md com\ibm\as400
C:>md com\ibm\as400\access

C:>java com.ibm.sslight.nlstools.keyrng com.ibm.as400.access.KeyRing
connect as20:9470
Password for com.ibm.as400.access.KeyRing.class:
toolbox
Connecting to as20:9,470
-------------------------- Server Certificate Chain --------------------------
Site Certificate - Number 0
Key : RSA/2048 bits
Subject: AS20.itsoroch.ibm.com, Rochester, Department 977,IBM ITSO, US
Issuer: ITSO Certification Authority, Rochester, Department 977, IBM ITSO, US
  Valid from: 4/6/99 6:50 PM
    Valid to: 4/6/00 6:50 PM
Fingerprint: E3:7E:0D:0F:13:39:B9:99:E0:EC:E0:6C:AB:2C:33:55

CA Certificate - Number 1
Key : RSA/2048 bits
Subject: ITSO Certification Authority, Rochester, Department 977,IBM ITSO, US
Issuer: ITSO Certification Authority, Rochester, Department 977,IBM ITSO, US
  Valid from: 4/6/99 6:43 PM
    Valid to: 4/7/02 6:43 PM
Fingerprint: 6D:F1:4C:F1:5D:E7:93:B5:8A:58:7A:69:F5:B4:92:02
------------------------------------------------------------------------------
Enter the number of the certificate to be added to com.ibm.as400.access.KeyRing
.class (q to quit):
1
Adding the CA Certificate - 1 to com.ibm.as400.access.KeyRing.class
Done.
```

Figure 337. Adding the CA Certificate to the KeyRing.class File

8. If you wish, you can now delete the SSLTool.zip file from your PC since it is only required to create a com.ibm.as400.access.KeyRing class.

Deployment Consideration

When you distribute the program, be sure to distribute this class. In addition, it *must* be included in the classpath prior to any AS/400 Toolbox for Java archives since this contains a KeyRing class.

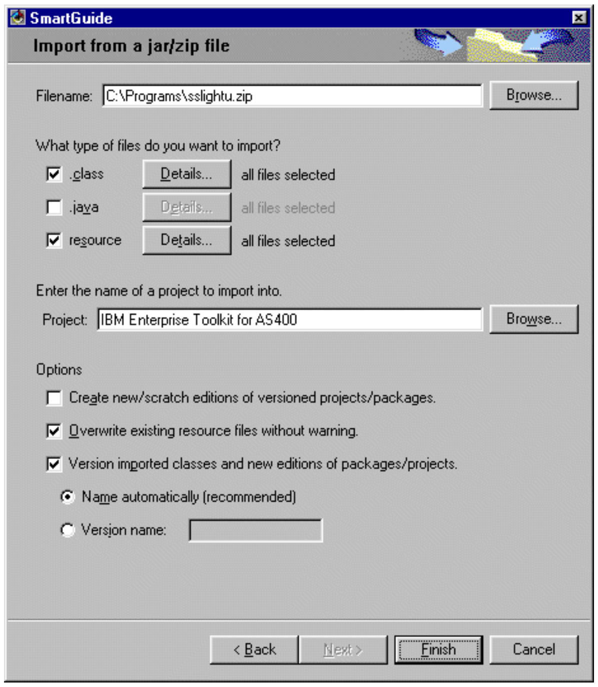
10.9 Modifying an Application to Use SSL with VisualAge 2.0

After downloading the sslightx.zip or sslightu.zip archive file, you need to import it into the VisualAge for Java environment. If it was necessary to create your own **c**om.ibm.as400.access.KeyRing class, you also need to import it into VisualAge for Java before you can run any modified applications. Once imported into VisualAge for Java, modifying a AS/400 Toolbox for Java program to use SSL is very simple.

10.9.1 Importing the Required Classes

Before attempting to use SSL or even importing these classes, be sure that you already followed the steps documented in Section 5.1, “Upgrading the AS/400 Toolbox Contained in VisualAge for Java 2.0” on page 213, to upgrade the AS/400 Toolbox for Java to Modification 2. Then, you can follow these steps to import the necessary classes into the VisualAge for Java Enterprise Edition 2.0 environment:

1. Start VisualAge for Java.
2. Select the **IBM Enterprise Toolkit for AS/400** project. If you are using a different edition of VisualAge, select the project containing the AS/400 Toolkit for Java Modification 2.
3. Create an Open Edition of this project.
4. Import the Java archive by selecting **File—>Import**. Check the **Import from a Jar file** option, and click **Next**.
5. Use the browse button to locate the sslightx.zip or sslightu.zip archive you previously downloaded. This is illustrated in Figure 338.


SmartGuide

Import from a jar/zip file

Filename: C:\Programs\sslightu.zip

What type of files do you want to import?

- [✓] .class — all files selected
- [ ] .java — all files selected
- [✓] resource — all files selected

Enter the name of a project to import into.

Project: IBM Enterprise Toolkit for AS400

Options

- [ ] Create new/scratch editions of versioned projects/packages.
- [✓] Overwrite existing resource files without warning.
- [✓] Version imported classes and new editions of packages/projects.
  - (●) Name automatically (recommended)
  - ( ) Version name:

Figure 338. Importing the SSL Support into VisualAge for Java

If you used the **SSLTools.zip** archive to create a customized KeyRing class, you need to perform the following additional steps:

1. Locate the existing com.ibm.as400.access.KeyRing class and delete it.
2. Import the new com.ibm.as400.access.KeyRing class by using the Import from a directory option. This is illustrated in Figure 339 on page 392.

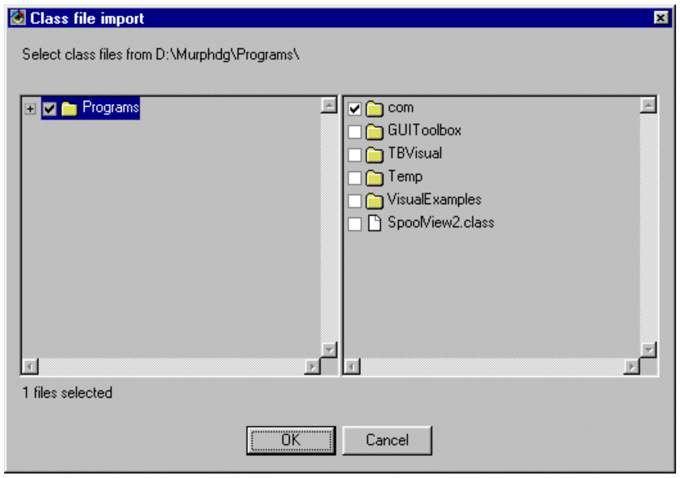


Figure 339. Importing a Modified KeyRing Class

You have now completed the all of the steps necessary to develop SSL-enabled AS/400 Toolbox for Java Modification 2 applications to communicate with AS/400 systems that run OS/400 V4R4 or later. Now, it is time to modify an application to use SSL.

10.9.2 Modifying the Program

For the AS/400 Toolbox for Java classes to use SSL to communicate with the AS/400 system, it is only necessary to change the AS400 objects to SecureAS400 objects. If you are using JDBC, set the secure property to true. This causes JDBC to use a SecureAS400 object.

In VisualAge for Java, you can use the Morph into feature to change an AS400 object to a SecureAS400 object. The following steps convert the TBVisual.RLFPExample program to use SSL:

1. Open the TBVisual.RLFPExample class in the Visual Composition Editor (VCE).
2. Right click on the **AS400** object and select the **Morph into** option, as illustrated in Figure 340 on page 393.

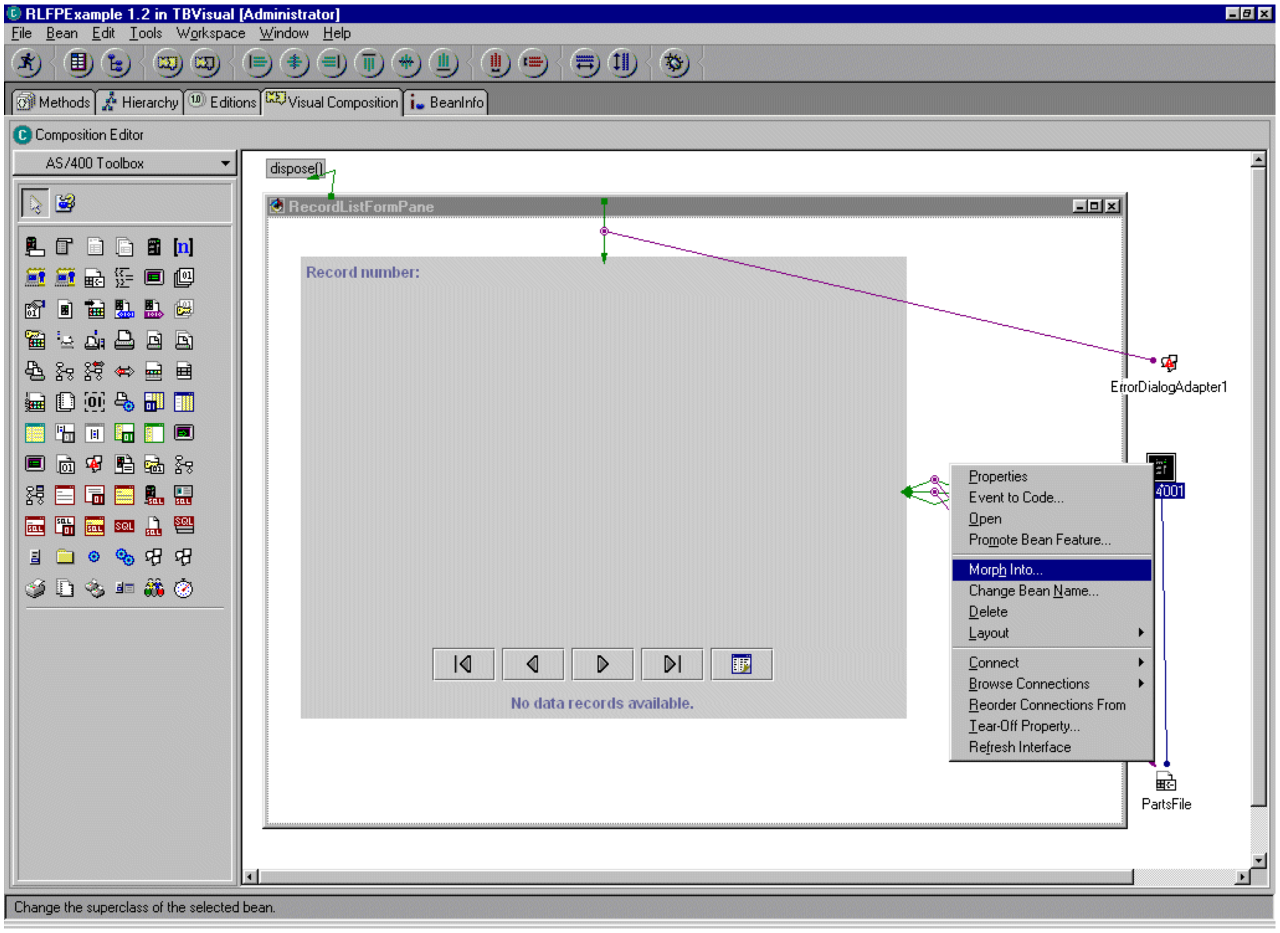
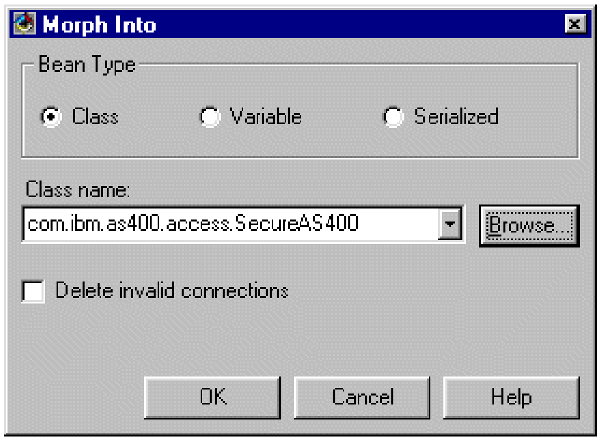


Figure 340. Morphing an AS400 Object

3. In the Morph into dialog, enter `com.ibm.as400.access.SecureAS400` and click **Next**, as shown in Figure 341.



Bean Type: [✓ Class] Variable Serialized

Class name: com.ibm.as400.access.SecureAS400

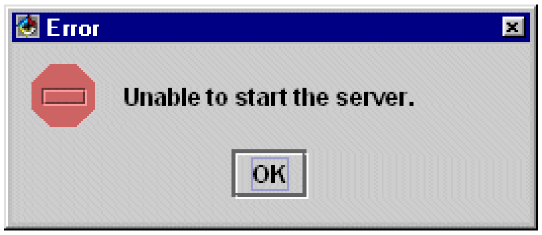
☐ Delete invalid connections

Figure 341. Morph the AS400 Object to a SecureAS400 Object

4. Once the morph operation is completed, the AS400 object is morphed to a SecureAS400. It uses SSL as opposed to normal socket communications. You need to repeat this step for all AS400 objects used in your application.

10.9.3 Testing the Changed Program

When it is run, the client application will not indicate that it is using SSL. However, should the program be unable to find the correct SSL-enabled server, a message will appear through the ErrorDialogAdaptor, as illustrated in Figure 342.



Unable to start the server.

Figure 342. The ErrorDialogAdaptor Message

If the application works and connects to the AS/400 system, you can use the AS/400 `netstat *cnn` command to verify which type of socket communication is being used.

10.9.4 Additional SSL Related Resources

For additional information, check out these resources:

- *HTTP Server for AS/400 Webmaster's Guide*, GC41-5434
- *Securing Your AS/400 from Harm on the Internet,* SG24-4929
- http://www.as400.ibm.com/toolbox
- http://publib.boulder.ibm.com/pubs/html/as400/ic2924/info/index.htm
  Click **Internet—>Digital certificate management**
- http://www.software.ibm.com/webservers/
- http://www.ibm.com/security
- http://www.ics.raleigh.com
- http://www.internet.ibm.com/commercepoint/registry/
- http://www.verisign.com/products/doc.html
- http://home.netscape.com/assist/security/ssl/index.html
- http://www.rsa.com

Appendix A. Example Programs

The Java 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 Version 2.0 Enterprise edition. OS/400 V3R2, V3R7, or later is required. To run the RMI example, V4R2 or later is required. To use any of the AS/400 Toolbox for Java Modification 2 examples, V4R2 or later is required. The following VisualAge for Java projects are available:

- **DaxProject** — Dax example Parts Ordering application. See Chapter 6, “Enterprise Access Builder for Data (DAX)” on page 267.
- **TeamLab**
  - ***TeamLab*** — Toolbox examples for JDBC stored procedures, DDM record level access, data queue, distributed program call, integrated file system, and print. See Chapter 3, “AS/400 Toolbox for Java” on page 89.
  - ***JDBCRmiEx*** — RMI example. See Chapter 6, “Enterprise Access Builder for Data (DAX)” on page 267.
  - ***PgmCall*** — Distributed Program Call JavaBean example. See Section 9.4, “Creating a Program Call JavaBean” on page 345.
  - ***TeamLabExtra***
    - *FancyLabel* — JavaBean Fancy Label example. See Chapter 9, “JavaBeans” on page 339.
    - *Demo* — FancyLabel example
    - *DpcxRpg2* — Program Call JavaBean program example
    - *DPCExample2* — Program Call JavaBean visual example
    - *DPCExampleDisplayAll* — Helper class for DPCExample2
- **Workshop** — Toolbox examples for JDBC. See Chapter 3, “AS/400 Toolbox for Java” on page 89.
- **TBVisual** — Toolbox examples using the GUI classes. See Chapter 4, “AS/400 Toolbox for Java — GUI Classes” on page 181.
  - *ISQLExSwing* — SQLResultSetTablePane example
  - *QBExample* — SQLQueryBuilderPane example
  - *RLFPExample* — RecordListFormPane example
  - *RLFPKeyedExample* — RecordListFormPane keyed access example
  - *RSFPExample* — SQLResultSetFormPane example
  - *RSTMExample* — SQLResultSetTableModel example

Important Information

These example programs have not been subjected to any formal testing. They are provided "as is." Use them *for reference only*. Please refer to Appendix D, “Special Notices” on page 413, for more information.

A.1 Downloading the Files from the Internet

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

The URL to access is: http://www.redbooks.ibm.com

Click on **Additional Materials**, and select the directory **SG242152**. In the SG242152 directory, click on **readme.txt**.

A.2 Setting Up VisualAge for Java

VisualAge for Java Enterprise edition, Version 2.0, requires the following software and hardware for development with the IDE:

- Windows 95 or Windows NT 4.0 with Service Pack 3
- TCP/IP communications protocol
- Pentium processor or higher recommended
- SVGA 800x600 display or higher (1024x768 recommended)
- 64MB RAM minimum (80MB recommended)
- Frames-capable Web browser
  - Netscape Navigator 4.04 or higher, or
  - Microsoft Internet Explorer 4.01 or higher
- Java Development Kit (JDK) 1.1 for deploying applications or
- JDK 1.1.2 for deploying applications using Swing components

The Java support classes described in the following sections are required.

A.2.1 AS/400 Toolbox for Java Classes

The example programs require that the AS/400 Toolbox for Java classes be inside the VisualAge for Java Integrated Development Environment. You must import these classes inside the IDE. Enterprise Edition simplifies this process. After you install VisualAge for Java 2.0 Enterprise edition, the Toolbox classes are available in the repository as part of the IBM Enterprise Toolkit for AS/400 project. To use the Toolbox classes, perform these steps:

1. From the workbench, click on **File** and click on **Quick Start**.
2. Click on **Features**, **Add Feature**, and **OK**.
3. Select **IBM Enterprise Toolkit for AS/400**, and click **OK**.

This adds the toolbox classes to your workspace. The IBM Enterprise Toolkit for AS400 is listed under All Projects.

The alternative is to perform these tasks:

1. Install LPP 5763-JC1 (5769-JC1 for V4R4) on an AS/400 system.
2. Download the classes to your workstation.
3. Import the classes into the VisualAge for Java IDE

A.2.2 IBM Enterprise Data Access Libraries

The example programs use IBM Enterprise Data Library supporting classes. To add them, complete these steps:

1. From the workbench, click **File** and **Quick Start**.
2. Click on **Features**, **Add Feature**, and **OK**.
3. Select **IBM Enterprise Data Access Libraries**, and click **OK**.

A.2.3 IBM Enterprise Access Builder Library

The DAX example programs use the IBM Enterprise Access Builder Library supporting classes. To add them, follow these steps:

1. From the workbench, click on **File** and **Quick Start**.
2. Click on **Features**, **Add Feature**, and **OK**.
3. Select **IBM Enterprise Access Builder Library**, and click **OK**.

Appendix B. AS/400 Source Listings

This appendix contains source listings for the following AS/400 programs used in the example programs:

- PARTS/PF
- SPROC2/SQLRPGLE
- SPROC3/SQLRPGLE
- DPCXRPG/RPGLE
- DQXRPG/RPGLE

B.1 PARTS/PF

```
A                                      UNIQUE
A          R PARTR
A            PARTNO         5S 0       COLHDG('Part Number')
A            PARTDS        25          COLHDG('Part Description')
A            PARTQY         5  0       COLHDG('Part Qty-on-Hand')
A            PARTPR         6  2       COLHDG('Part Price')
A            PARTDT          L         DATFMT(*ISO)
A                                      COLHDG('Part Shipment Date')
A          K PARTNO
```

B.2 SPROC2/SQLRPGLE

```
D*
D*  Defines PART ID As a Integer (Binary 4.0)
D*
D #PRTDS          DS
D  #PART                  1      4B 0
D #OPTDS          DS
D  #OPT                   1      4B 0
D*PARTNO                   1      2B 0
C     *ENTRY        PLIST
C                   PARM                    #OPTDS
C                   PARM                    #PRTDS
C*  Copy PART NUMBER to RPG Native Variable With Same Attributes Of
C*  Field In PARTS Master File (5,0) For Performance Issues
C                   Z-ADD     #PART         PART              5 0
C     #OPT          CASEQ     1             ONEREC
C     #OPT          CASEQ     2             ALLREC
C     #OPT          CASEQ     3             DELREC
C                   CAS                     BADOPT
C                   ENDCS
C*
C     ONEREC        BEGSR
C/Exec Sql Declare C1 Cursor For
C+   Select
C+   PARTNO,
C+   PARTDS,
C+   PARTQY,
C+   PARTPR,
C+   PARTDT
C+
C+   From PARTS                      -- From PART Master File
C+
C+   Where PARTNO  = :PART
C+
C+
C+  For Fetch Only                  -- Read Only Cursor
C/End-Exec
C*
C/Exec Sql
C+  Open C1
C/End-Exec
C*
C/Exec Sql
C+  Set Result Sets Cursor C1
C/End-Exec
```

```
C*
C                   RETURN
C                   ENDSR
C*
C     ALLREC        BEGSR
C/Exec Sql Declare C2 Cursor For
C+   Select
C+   PARTNO,
C+   PARTDS,
C+   PARTQY,
C+   PARTPR,
C+   PARTDT
C+
C+   From PARTS                      -- From PART Master File
C+
C+
C+  Order By PARTNO                -- Sort By PARTNO              me
C+
C+  For Fetch Only                 -- Read Only Cursor
C/End-Exec
C*
C/Exec Sql
C+  Open C2
C/End-Exec
C*
C/Exec Sql
C+  Set Result Sets Cursor C2
C/End-Exec
C                   RETURN
C                   ENDSR
C*
C     DELREC        BEGSR
C/Exec Sql
C+   Delete
C+
C+   From PARTS                      -- From PART Master File
C+
C+   Where PARTNO  = :PART
C+
C+
C/End-Exec
C*
C                   RETURN
C                   ENDSR
C*------------------------------------------------------------------
C* SUBROUTINE BADOPT
C*
C* AN UNRECOGNIZED OPTION PARAMETER WAS SET - RETURN '4' FOR
C* UNKNOWN.
C*
C*------------------------------------------------------------------
C     BADOPT        BEGSR
C                   MOVE      4             #OPT
C                   RETURN
C                   ENDSR
C*
C*
C*
```

B.3 SPROC3/SQLRPGLE

```
D* Option (1=Update/2=Add)
D #OPTDS          DS
D  #OPT                   1      4B 0
D*  Defines PART ID As an Integer (Binary 4.0)
D #PRTDS          DS
D  #PART                  1      4B 0
D*  Defines DESC As a String (25)
D #DSCDS          DS
D  #DESC                  1     25A
D*  Defines QTY As an Integer (Binary 4.0)
D #QTYDS          DS
D  #QTY                   1      4B 0
D*  Defines PRICE As a Float (zoned 6.2)
D #PRCDS          DS
D  #PRC                   1      4P 2
```

```
D*  Defines DATE As a Date (date 10)
D #DATDS          DS
D  #DAT                   1     10D
C     *ENTRY        PLIST
C                   PARM                    #OPTDS
C                   PARM                    #PRTDS
C                   PARM                    #DSCDS
C                   PARM                    #QTYDS
C                   PARM                    #PRCDS
C                   PARM                    #DATDS
C*  Copy PART NUMBER to RPG Native Variable With Same Attributes Of
C*  Field In PARTS Master File (5,0) For Performance Issues
C                   Z-ADD     #PART         PART              5 0
C                   MOVEL     #DESC         DESC             25
C                   Z-ADD     #QTY          QTY               5 0
C                   Z-ADD     #PRC          PRC               6 2
C                   MOVEL     #DAT          DAT              10
C     #OPT          CASEQ     1             UPDREC
C     #OPT          CASEQ     2             ADDREC
C                   CAS                     BADOPT
C                   ENDCS
C*
C     UPDREC        BEGSR
C/Exec Sql
C+   Update PARTS Set
C+   PARTDS = :DESC,
C+   PARTQY = :QTY,
C+   PARTPR = :PRC,
C+   PARTDT = :DAT
C+
C+   Where PARTNO  = :PART
C+
C/End-Exec
C*
C                   RETURN
C                   ENDSR
C*
C     ADDREC        BEGSR
C/Exec Sql
C+   Insert Into PARTS
C+   (PARTNO,
C+   PARTDS,
C+   PARTQY,
C+   PARTPR,
C+   PARTDT)
C+   Values
C+   (:PART,
C+    :DESC,
C+    :QTY,
C+    :PRC,
C+    :DAT)
C+
C/End-Exec
C*
C                   RETURN
C                   ENDSR
C*
C*---------------------------------------------------------------
C* SUBROUTINE BADOPT
C*
C* AN UNRECOGNIZED OPTION PARAMETER WAS SET - RETURN '3' FOR
C* UNKNOWN.
C*
C*---------------------------------------------------------------
C     BADOPT        BEGSR
C                   MOVE      3             #OPT
C                   RETURN
C                   ENDSR
C*
C*
```

B.4 DPCXRPG/RPGLE

```
H*DPCXRPG
H* This program is called from the client via the Distributed
H* Program Call API or as a stored procedure via ODBC. It
H* returns data to the client from the PARTS database file.
H*------------------------------------------------------------
H
FPARTS     UF A E           K DISK
C*------------------------------------------------------------
C* MAIN PROGRAM
C*
C* Take action depending on the 'option' parameter:
C*          Option    Action
C*             S      Retrieve a single record for supplied key
C*             A      Retrieve all records
C*             F      Fetch the next record based on cursor posn.
C*             E      End the program
C*             D      Delete a single record for supplied key
C*             U      Update/Add single record for supplied key / fields
C*------------------------------------------------------------
C     *ENTRY        PLIST
C                   PARM                    OPTION            1
C     PPARTNO       PARM                    PARTNO
C     PPARTDS       PARM                    PARTDS
C     PPARTQY       PARM                    PARTQY
C     PPARTPR       PARM                    PARTPR
C     PPARTDT       PARM                    PARTDT
C     *LIKE         DEFINE    PARTNO        PPARTNO
C     *LIKE         DEFINE    PARTDS        PPARTDS
C     *LIKE         DEFINE    PARTQY        PPARTQY
C     *LIKE         DEFINE    PARTPR        PPARTPR
C     *LIKE         DEFINE    PARTDT        PPARTDT
C     OPTION        CASEQ     'S'           ONEREC
C     OPTION        CASEQ     'A'           ALLREC
C     OPTION        CASEQ     'F'           NXTREC
C     OPTION        CASEQ     'E'           ENDPRG
C     OPTION        CASEQ     'D'           DELREC
C     OPTION        CASEQ     'U'           UPDREC
C                   CAS                     BADOPT
C                   ENDCS
C*--------------------------------------------------------------
C* SUBROUTINE - ONEREC
* This subroutine attempts to find the requested part in the
C* PARTS file. If the record is found, set the OPTION parameter
C* to 'Y', otherwise to 'X' to indicate record not found, then
C* return.
C*--------------------------------------------------------------
C     ONEREC        BEGSR
C* Return only one record
C     PARTNO        CHAIN     PARTR                              40        40
C     *IN40         IFEQ      '1'
C                   MOVE      'X'           OPTION
C                   ELSE
C                   MOVE      'Y'           OPTION
C                   ENDIF
C                   RETURN
C                   ENDSR
C*--------------------------------------------------------------
C* SUBROUTINE - ALLREC
C* This subroutine re-positions the cursor to the start of the
C* PARTS file anticipating subsequent calls to fetch the records
C* sequentially. If the SETLL operation fails, set the option
C* parameter to 'X', otherwise 'Y'.
C*--------------------------------------------------------------
C     ALLREC        BEGSR
C     *LOVAL        SETLL     PARTS                              50
C     *IN50         IFEQ      '1'
C                   MOVE      'X'           OPTION
C                   ELSE
C                   MOVE      'Y'           OPTION
C                   ENDIF
C                   RETURN
C                   ENDSR
C*--------------------------------------------------------------
C* SUBROUTINE - NXTREC
C* This subroutine retrieves the next sequential record in the
```

```
C* PARTS file. If the record is found, set the option parameter
C* to 'Y', otherwise 'X'.
C*---------------------------------------------------------------
C     NXTREC        BEGSR
C                   READ      PARTS                                  60     60=EOF
C     *IN60         IFEQ      '0'
C                   MOVE      'Y'           OPTION
C                   ELSE
C                   MOVE      'X'           OPTION
C                   ENDIF
C                   RETURN
C                   ENDSR
C*---------------------------------------------------------------
C* SUBROUTINE ENDPRG
C*
C* This subroutine terminates the program.
C*
C*---------------------------------------------------------------
C     ENDPRG        BEGSR
C                   MOVE      'Y'           OPTION
C                   SETON                                        LR
C                   RETURN
C                   ENDSR
C*---------------------------------------------------------------
C* SUBROUTINE - DELREC
C* This subroutine attempts to find the requested part in the
C* PARTS file. If the record is found, delete it, set the OPTION
C* parameter to 'Y', otherwise to 'X' to indicate record not found,
C* then return.
C*---------------------------------------------------------------
C     DELREC        BEGSR
C* Delete only one record
C     PARTNO        CHAIN     PARTR                              40
C     *IN40         IFEQ      '1'
C                   MOVE      'X'           OPTION
C                   ELSE
C                   DELETE    PARTR
C                   MOVE      'Y'           OPTION
C                   ENDIF
C                   RETURN
C                   ENDSR
C*---------------------------------------------------------------
C* SUBROUTINE - UPDREC
C* This subroutine attempts to update the requested part in the
C* PARTS file. If the record is found, update it, set the OPTION
C* parameter to 'Y', otherwise add it, set the OPTION parameter to
C* 'X' to indicate the record was added, then return
C*---------------------------------------------------------------
C     UPDREC        BEGSR
C* Update only one record
C     PARTNO        CHAIN     PARTR                              40
C     *IN40         IFEQ      '1'
C                   Z-ADD     PPARTNO       PARTNO
C                   MOVEL     PPARTDS       PARTDS
C                   Z-ADD     PPARTQY       PARTQY
C                   Z-ADD     PPARTPR       PARTPR
C                   MOVEL     PPARTDT       PARTDT
C                   WRITE     PARTR
C                   MOVE      'X'           OPTION
C                   ELSE
C                   MOVEL     PPARTDS       PARTDS
C                   Z-ADD     PPARTQY       PARTQY
C                   Z-ADD     PPARTPR       PARTPR
C                   MOVEL     PPARTDT       PARTDT
C                   UPDATE    PARTR
C                   MOVE      'Y'           OPTION
C                   ENDIF
C                   RETURN
C                   ENDSR
C*---------------------------------------------------------------
C* SUBROUTINE BADOPT
C*
C* An unrecognised option parameter was set - return 'U' for
C* unknown.
C*
C*---------------------------------------------------------------
C     BADOPT        BEGSR
C                   MOVE      'U'           OPTION
```

```
C                   RETURN
C                   ENDSR
```

B.5 DQXRPG/RPGLE

```
*DQXRPG
*
* This is a never-ending-program that runs in the background
* as a batch job. It checks the data queue DQINPT for
* any queue entries received. Once an entry arrives in the
* data queue, the program retrieves and processes it.
*
* This program should be submitted with the SBMJOB command
* and terminated with ENDJOB OR WRKACTJOB commands, or by
* placing an entry starting with 'E' on the DQINPT data queue.
*-----------------------------------------------------------
H
FPARTS     UF A E           K DISK
*------------------------------------------------------------
* DATA STRUCTURES
*
* DATAI - input data record 6 bytes
* DATAO - output data record 48 bytes
*------------------------------------------------------------
D DATAI           DS
D  OPTION                 1      1
D  INPNO                  2      6  0
D  IARTDS                 7     31
D  IARTQY                32     34P 0
D  IARTPR                35     38P 2
D  IARTDT                39     48D
D DATAO           DS
D  RESULT                 1      1
D  PARTNO                 2      6  0
D  PARTDS                 7     31
D  PARTQY                32     34P 0
D  PARTPR                35     38P 2
D  PARTDT                39     48D
*------------------------------------------------------------
* CONSTANTS
*
* DQINPT - data queue used for receiving input records
* DQOUPT - data queue used for sending records
* APILIB - library name
*-----------------------------------------------------------
D DQINPT          C                   CONST('DQINPT    ')
D DQOUPT          C                   CONST('DQOUPT    ')
D LIBL            C                   CONST('APILIB    ')
*------------------------------------------------------------
* MAIN PROGRAM
*
* Loop on read to data queue. Action depends on the 'option'
* flag:
*         Option    Action
*            S      Retrieve a single record for supplied key
*            A      Retrieve all records in file
*            E      End the program
*            D      Delete a single record for supplied key
*            U      Update/Add single record for supplied key / fields
*-----------------------------------------------------------
C                   EXSR      RCVDQ
C     OPTION        DOWNE     'E'
C                   EXSR      READR
C                   EXSR      RCVDQ
C                   ENDDO
C                   SETON                                        LR
*-------------------------------------------------------------
* SUBROUTINE RCVDQ
*
* This subroutine performs the QRCVDTAQ function. Notice that
* the wait parameter is set to a negative value to force it
* to wait until a queue entry is available.
*
C     RCVDQ         BEGSR
C                   MOVE      DQINPT        QUEUEI           10
C                   MOVE      LIBL          LIBLD            10
```

```
C                   Z-ADD     48            FLDDL             5 0
C                   Z-ADD     -9            WAIT              5 0
C                   CALL      'QRCVDTAQ'
C                   PARM                    QUEUEI
C                   PARM                    LIBLD
C                   PARM                    FLDDL
C                   PARM                    DATAI
C                   PARM                    WAIT
C                   ENDSR
*-----------------------------------------------------------------
* SUBROUTINE - READR
* This subroutine retrieves the part number from the data queue
* DQINPT, searches the data base file PARTS using the part number
* just received. If the record is found, send the record to the
* data queue DQOUPT. If option 'A' is received, send all records
* to the data queue DQOUPT.
*
* The 'result' flag is set as follows
*         Result    Meaning
*            Y      Record found and being returned
*            X      Record not found or eof
*-----------------------------------------------------------------
C     READR         BEGSR
*
C     OPTION        IFEQ      'A'
* Return all records in the file
C     *LOVAL        SETLL     PARTS
C                   READ      PARTS                                   60      60=EOF
*
C     *IN60         DOWEQ     '0'
C                   MOVE      'Y'           RESULT
C                   EXSR      SNDDQ
C                   READ      PARTS                                   60
C                   ENDDO
*
C                   MOVE      'X'           RESULT
C                   EXSR      SNDDQ
*
C                   ELSE
*
C     OPTION        IFEQ      'D'
* Delete one record
C     INPNO         CHAIN     PARTR                              98
*
C     *IN98         IFEQ      '1'
C                   MOVE      'X'           RESULT
C                   ELSE
C                   DELETE    PARTR
C                   MOVE      'Y'           RESULT
C                   ENDIF
C                   EXSR      SNDDQ
*
C                   ELSE
*
C     OPTION        IFEQ      'U'
* Update record if found / Add record if not found
C     INPNO         CHAIN     PARTR                              98
*
C     *IN98         IFEQ      '1'
C                   Z-ADD     INPNO         PARTNO
C                   MOVEL     IARTDS        PARTDS
C                   Z-ADD     IARTQY        PARTQY
C                   Z-ADD     IARTPR        PARTPR
C                   MOVEL     IARTDT        PARTDT
C                   WRITE     PARTR
C                   MOVE      'X'           RESULT
C                   ELSE
C                   MOVEL     IARTDS        PARTDS
C                   Z-ADD     IARTQY        PARTQY
C                   Z-ADD     IARTPR        PARTPR
C                   MOVEL     IARTDT        PARTDT
C                   UPDATE    PARTR
C                   MOVE      'Y'           RESULT
C                   ENDIF
C                   EXSR      SNDDQ
*
C                   ELSE
* Return only one record
```

```
C     INPNO         CHAIN     PARTR                             98
*
C     *IN98         IFEQ      '1'
C                   MOVE      'X'           RESULT
C                   ELSE
C                   MOVE      'Y'           RESULT
C                   ENDIF
*
C                   EXSR      SNDDQ
C                   ENDIF
*
C                   ENDIF
C                   ENDIF
*
C                   ENDSR
*-----------------------------------------------------------------
* SUBROUTINE SNDDQ
*
* This subroutine performs the QSNDDTAQ function.
*
C     SNDDQ         BEGSR
C                   MOVE      DQOUPT        QUEUEO           10
C                   MOVE      LIBL          LIBLD
C                   Z-ADD     48            FLDDL
C                   CALL      'QSNDDTAQ'
C                   PARM                    QUEUEO
C                   PARM                    LIBLD
C                   PARM                    FLDDL
C                   PARM                    DATAO
C                   ENDSR
```

Appendix C. GUI Builder Code

This appendix contains the code for the GUI Builder examples.

C.1 SystemStatusEngine.java

This section shows the SystemStatusEngine.java code after completing the task in Section 5.5.4, “Modifying the Databean to Retrieve Data from the AS/400 System” on page 229.

```
import com.ibm.as400.ui.framework.java.*;
import com.ibm.as400.access.*;
import com.ibm.as400.opnav.Monitor;

public class SystemStatusEngine extends Object implements DataBean {
  private String m_sSystemASPPercent;
  private String m_sCpuUtilization;
  private String m_sSystemASPSize;
  private String m_sPermanentAddressesUsed;
  private String m_sTemporaryAddressesUsed;
  private AS400 m_sAS400;

public SystemStatusEngine(AS400 anAS400) {
  m_sAS400 = anAS400;
  }

  public Capabilities getCapabilities() {
    return null;
  }

  public String getCpuUtilization() {
    return m_sCpuUtilization;
  }

  public String getPermanentAddressesUsed() {
    return m_sPermanentAddressesUsed;
  }

  public String getSystemASPPercent() {
    return m_sSystemASPPercent;
  }

  public String getSystemASPSize() {
    return m_sSystemASPSize;
  }

  public String getTemporaryAddressesUsed() {
    return m_sTemporaryAddressesUsed;
  }

  public void load() {
    try {
      SystemStatus aStatus = new SystemStatus (m_sAS400);
      m_sSystemASPPercent = new Float(aStatus.getPercentSystemASPUsed()).toString() + " %";
      m_sCpuUtilization = new Float(aStatus.getPercentProcessingUnitUsed()).toString() +
        " %";
      m_sSystemASPSize = new Integer(aStatus.getSystemASP()).toString() + " MB";
      m_sPermanentAddressesUsed = new
        Float(aStatus.getPercentPermanentAddresses()).toString() + "%";
      m_sTemporaryAddressesUsed = new
        Float(aStatus.getPercentTemporaryAddresses()).toString() + "%";
    } catch (Exception e) {
      Monitor.logThrowable(e);
    }
  }

  public void save() {
  }

  public void verifyChanges() {
  }

}
```

C.2 SystemStatusManager

This section shows the SystemStatusManager.java source code after completing the task in Section 5.5.6, “Adding an Operations Navigator Plug-in” on page 232.

```
import com.ibm.as400.opnav.*;
import com.ibm.as400.access.*;
import com.ibm.as400.ui.framework.java.*;

class SystemStatusManager extends Object implements ActionsManager {
  private ObjectName[] initObjs;
  private ObjectName dragDropObj;

public void initialize(ObjectName[] arg1, ObjectName arg2) {
  initObjs = arg1;
  dragDropObj = arg2;
}

public ActionDescriptor[] queryActions(int arg1) {
  ActionDescriptor[] actions = new ActionDescriptor[0];
  String objType = null;

  try {
    objType = initObjs[0].getObjectType();
    if (objType.equals("AS4")) {
      if ((arg1 & CUSTOM_ACTIONS) == CUSTOM_ACTIONS) {
        actions = new ActionDescriptor[1];
        ActionDescriptor act = new ActionDescriptor(1);
        act.setText("System Status");
        act.setHelpText("Loads the ITSO System Status Plugin");
        act.setVerb("ITSOSysSts");
        actions[0] = act;
      }
    }
  }
  catch (Exception e) {
    Monitor.logThrowable(e);
  }
  return actions;
}

public void actionSelected(int arg1, java.awt.Frame arg2) {
  if (arg1 == 1) {
    try {
      AS400 theMachine = (AS400)initObjs[0].getSystemObject();
      SystemStatusEngine theSystemEngine = new SystemStatusEngine(theMachine);
      theSystemEngine.load();
      DataBean[] dbeans = {theSystemEngine};
      PanelManager pm = null;
      try {
        pm = new PanelManager("SystemStatus","SystemResources",dbeans,arg2);
      }
      catch (DisplayManagerException e){
          e.displayUserMessage(arg2);
        }
      pm.setVisible(true);
    }
    catch (Exception e) {
      Monitor.logThrowable(e);
    }
  }
}

}
```

C.3 SystemStatus Registry

This section show the Windows registry after completing the task in Section 5.5.7, “Modifying the Windows Registry” on page 236. Some additional comments have been added to clarify the entries.

```
REGEDIT4
; Define the primary registry key for the plugin

[HKEY_CLASSES_ROOT\IBM.AS400.Network\3RD PARTY EXTENSIONS\ITSO.SystemStatusPlugin]
; You have written an extension to the shell, so use EXT for the type of
; plugin. If a ListManager interface had been implemented then you would
; have set this to Plugin
"Type"="EXT"
"MinimumRISCRelease"="ANY"
"MinimumIMPIRelease"="NONE"
"ProductID"="NONE"
"ServerEntryPoint"="NONE"
; The next value point to the base directory or archive used to find any
; required user class definitions.
"JavaPath"="C:\\L04\\Student\\"
"JavaMRI"=""
; Although no DLL is used you are required to add a registry entry. It will
; never be used or accessed but it MUST exist.
"NLS"="itso.dll"
"NameID"=dword:00000000
"DescriptionID"=dword:00000000

;-------------------------------------------------------------------
; Register a context menu handler for the new folder and its objects

[HKEY_CLASSES_ROOT\IBM.AS400.Network\3RD PARTY
EXTENSIONS\ITSO.SystemStatusPlugin\shellex\AS/400
Network\*\ContextMenuHandlers\{1827A857-9C20-11d1-96C3-00062912C9B2}]
; Java class is the classname of the ActionManager implementor class.
"JavaClass"="SystemStatusManager"
```

C.4 SystemStatusEngine.java

This is the source code for SystemStatusEngine.java after completing the task in Section 5.5.10, “Modifying the SystemStatusEngine DataBean” on page 239.

```
import com.ibm.as400.ui.framework.java.*;
import com.ibm.as400.access.*;
import com.ibm.as400.opnav.Monitor;

public class SystemStatusEngine extends Object
  implements DataBean
{
  private String[] m_sSystemPoolNumber;
  private ItemDescriptor[] m_idSystemPoolNumber;
  private String[] m_sSystemPoolName;
  private ItemDescriptor[] m_idSystemPoolName;
  private String[] m_sSystemPoolSize;
  private ItemDescriptor[] m_idSystemPoolSize;
  private String[] m_sDBFaulting;
  private ItemDescriptor[] m_idDBFaulting;
  private String[] m_sNonDBFaulting;
  private ItemDescriptor[] m_idNonDBFaulting;
  private String m_sSystemASPPercent;
  private String m_sCpuUtilization;
  private String m_sSystemASPSize;
  private String m_sPermanentAddressesUsed;
  private String m_sTemporaryAddressesUsed;
  private AS400 m_sAS400;

  public SystemStatusEngine(AS400 anAS400) {
    m_sAS400 = anAS400;
  }

  public Capabilities getCapabilities() {
    return null;
  }

  public String getCpuUtilization() {
    return m_sCpuUtilization;
  }

  public ItemDescriptor[] getDBFaultingList() {
    return m_idDBFaulting;
  }
```

```
public String[] getDBFaultingSelection() {
  return m_sDBFaulting;
}

public ItemDescriptor[] getNonDBFaultingList() {
  return m_idNonDBFaulting;
}

public String[] getNonDBFaultingSelection() {
  return m_sNonDBFaulting;
}

public String getPermanentAddressesUsed() {
  return m_sPermanentAddressesUsed;
}

public String getSystemASPPercent() {
  return m_sSystemASPPercent;
}

public String getSystemASPSize() {
  return m_sSystemASPSize;
}

public ItemDescriptor[] getSystemPoolNameList() {
  return m_idSystemPoolName;
}

public String[] getSystemPoolNameSelection() {
  return m_sSystemPoolName;
}

public ItemDescriptor[] getSystemPoolNumberList() {
  return m_idSystemPoolNumber;
}

public String[] getSystemPoolNumberSelection() {
  return m_sSystemPoolNumber;
}

public ItemDescriptor[] getSystemPoolSizeList() {
  return m_idSystemPoolSize;
}

public String[] getSystemPoolSizeSelection() {
  return m_sSystemPoolSize;
}

public String getTemporaryAddressesUsed() {
  return m_sTemporaryAddressesUsed;
}

public void load() {
  try {
    SystemStatus aStatus = new SystemStatus (m_sAS400);
    int numOfPools = aStatus.getPoolsNumber();
    java.util.Enumeration thePools = aStatus.getSystemPools();

    m_sSystemASPPercent = new Float(aStatus.getPercentSystemASPUsed()).toString() + " %";
    m_sCpuUtilization = new Float(aStatus.getPercentProcessingUnitUsed()).toString() +
      " %";
    m_sSystemASPSize = new Integer(aStatus.getSystemASP()).toString() + " MB";
    m_sPermanentAddressesUsed = new
        Float(aStatus.getPercentPermanentAddresses()).toString() + " %";
    m_sTemporaryAddressesUsed = new
        Float(aStatus.getPercentTemporaryAddresses()).toString() + " %";

    m_sSystemPoolNumber = new String[numOfPools];
    m_idSystemPoolNumber = new ItemDescriptor[numOfPools];
    m_sSystemPoolName = new String[numOfPools];
    m_idSystemPoolName = new ItemDescriptor[numOfPools];
    m_sSystemPoolSize = new String[numOfPools];
    m_idSystemPoolSize = new ItemDescriptor[numOfPools];
    m_sDBFaulting = new String[numOfPools];
    m_idDBFaulting = new ItemDescriptor[numOfPools];
    m_sNonDBFaulting = new String[numOfPools];
    m_idNonDBFaulting = new ItemDescriptor[numOfPools];
    int i = 0;
```

```
        while (thePools.hasMoreElements()) {
          SystemPool tempPool = (SystemPool) thePools.nextElement();

          m_sSystemPoolNumber[i] = new Integer(tempPool.getPoolIdentifier()).toString();
          m_idSystemPoolNumber[i] = new ItemDescriptor ("PNum"+ m_sSystemPoolNumber[i],
            m_sSystemPoolNumber[i]);
          m_sSystemPoolName[i] = tempPool.getPoolName();
          m_idSystemPoolName[i] = new ItemDescriptor ("PName" + m_sSystemPoolNumber[i],
            m_sSystemPoolName[i]);
          m_sSystemPoolSize[i] = new Integer(tempPool.getPoolSize()).toString() + " K bytes";
          m_idSystemPoolSize[i] = new ItemDescriptor("PSize" +
            m_sSystemPoolNumber[i],m_sSystemPoolSize[i]);
          m_sDBFaulting[i] = new Float(tempPool.getDatabaseFaults()).toString() + " page
            faults per sec " +
            new Float(tempPool.getDatabasePages()).toString() + " pages brought in per min";
          m_idDBFaulting[i] = new ItemDescriptor("DBFault" + m_sSystemPoolNumber[i],
            m_sDBFaulting[i]);
          m_sNonDBFaulting[i] = new Float(tempPool.getNonDatabaseFaults()).toString() + " page
            faults per sec " +
            new Float(tempPool.getNonDatabasePages()).toString() + " pages brought in per
            min";
          m_idNonDBFaulting[i] = new ItemDescriptor("NonDBFault" + m_sSystemPoolNumber[i],
            m_sNonDBFaulting[i]);
          i ++;
          }
        } catch (Exception e) {
        Monitor.logThrowable(e);
      }
    }


    public void save() {
    }

    public void setDBFaultingList(ItemDescriptor[] items) {
      m_idDBFaulting = items;
    }

    public void setDBFaultingSelection(String[] select) {
      m_sDBFaulting = select;
    }

    public void setNonDBFaultingList(ItemDescriptor[] items) {
      m_idNonDBFaulting = items;
    }

    public void setNonDBFaultingSelection(String[] select) {
      m_sNonDBFaulting = select;
    }

    public void setSystemPoolNameList(ItemDescriptor[] items) {
      m_idSystemPoolName = items;
    }

    public void setSystemPoolNameSelection(String[] select) {
      m_sSystemPoolName = select;
    }

    public void setSystemPoolNumberList(ItemDescriptor[] items) {
      m_idSystemPoolNumber = items;
    }

    public void setSystemPoolNumberSelection(String[] select) {
      m_sSystemPoolNumber = select;
    }

    public void setSystemPoolSizeList(ItemDescriptor[] items) {
      m_idSystemPoolSize = items;
    }

    public void setSystemPoolSizeSelection(String[] select) {
      m_sSystemPoolSize = select;
    }

    public void verifyChanges(){
    }

  }
```

C.5 SystemStatusManager

This is the source code for the SystemStatusEngine.java file after completing the task in Section 5.5.11, “Modifying the SystemStatusManager” on page 241. The only change is mofiying the name of the initial panel to be displayed.

```
import com.ibm.as400.opnav.*;
import com.ibm.as400.access.*;
import com.ibm.as400.ui.framework.java.*;


class SystemStatusManager extends Object implements ActionsManager {
  private PanelManager pm = null;
  private SystemStatusEngine theSystemEngine = null;
  private ObjectName[] initObjs;
  private ObjectName dragDropObj;

public void actionSelected(int arg1, java.awt.Frame arg2) {
  if (arg1 == 1) {
    try {
      AS400 theMachine = (AS400)initObjs[0].getSystemObject();
      theSystemEngine = new SystemStatusEngine(theMachine);
      theSystemEngine.load();
      DataBean[] dbeans = {theSystemEngine};

      try {
        pm = new PanelManager("SystemStatus","SystemStatus",dbeans,arg2);
      }
      catch (DisplayManagerException e){
          e.displayUserMessage(arg2);
      }
      pm.setVisible(true);
    }
    catch (Exception e) {
      Monitor.logThrowable(e);
    }
  }
}

public void initialize(ObjectName[] arg1, ObjectName arg2) {

  initObjs = arg1;
  dragDropObj = arg2;
}

public ActionDescriptor[] queryActions(int arg1) {

  ActionDescriptor[] actions = new ActionDescriptor[0];
  String objType = null;

  try {
    objType = initObjs[0].getObjectType();
    if (objType.equals("AS4")) {
      if ((arg1 & CUSTOM_ACTIONS) == CUSTOM_ACTIONS) {
        actions = new ActionDescriptor[1];
        ActionDescriptor act = new ActionDescriptor(1);
        act.setText("System Status");
        act.setHelpText("Loads the ITSO System Status Plugin");
        act.setVerb("ITSOSysSts");
        actions[0] = act;
      }
    }
  }
  catch (Exception e) {
    Monitor.logThrowable(e);
  }
  return actions;
}
}
```

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 VisualAge for Java or the AS/400 Toolbox for Java. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java 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.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

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.

E.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 417.

- *Building AS/400 Applications with Java*, SG24-2163
- *AS/400 Client/Server Performance Using the Windows Clients*, SG24-4526
- *Securing Your AS/400 from Harm on the Internet,* SG24-4929
- *Application Development with VisualAge for Java Enterprise*, SG24-5081
- *VisualAge for Java Enterprise Team*, SG24-5245
- *Building AS/400 Internet Based Applications with Java*, SG24-5337

E.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at `http://www.redbooks.ibm.com/` for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037

E.3 Other Publications

These publications are also relevant as further information sources:

- *HTTP Server for AS/400 Webmaster's Guide*, GC41-5434
- *Object Oriented Technology: A Manager's Guide*, SH20-9092
- *JavaBeans for Dummies,* SR23-7895
- Flanagan, David. *Java in a Nutshell*. Indianapolis, IN: Sams.net Publishing, 1996 (ISBN 1-56592-183-6)

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** `http://www.redbooks.ibm.com/`

  Search for, view, download, or order hardcopy/CD-ROM redbooks from the redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

  Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at `http://w3.itso.ibm.com/` and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access `MyNews` at `http://w3.ibm.com/` for redbook, residency, and workshop announcements.

IBM Redbook Fax Order Form

Please send me the following:

Title	**Order Number**	**Quantity**

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

AFP	advanced function printing
APA	all points addressable
AWT	Abstract Windowing Toolkit
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
JFC	Java Foundation Classes
JIT	Just in Time Compiler
JVM	Java Virtual Machine
MI	Machine Interface
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OOP	Object Oriented Programming
PTF	Program Temporary Fix
RAD	Rapid Application Development
RMI	Remote Method Invocation
SCS	SNA Character Set
SLIC	System Licensed Internal Code
SSL	secure sockets layer
TIMI	Technology Independent Machine Interface
UML	Unified Methodology Language
URL	Universal Resource Locator
VCE	Visual Composition Editor
WWW	World Wide Web

Index

ITSO Redbook Evaluation

Building AS/400 Client/Server Applications with Java
SG24-2152-02

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to `redbook@us.ibm.com`

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

SG24-2152-02

Printed in the U.S.A.

Building AS/400 Client/Server Applications with Java

SG24-2152-02