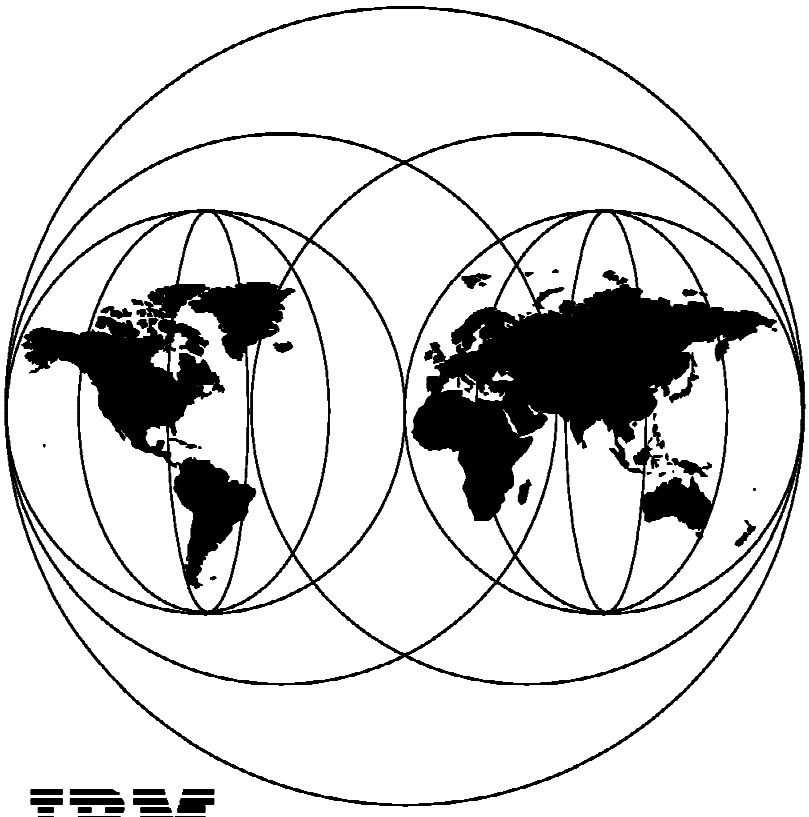
SG24-2152-00

Accessing the AS/400 System with Java

August 1997



IBM

International Technical Support Organization
Rochester Center

International Technical Support Organization

SG24-2152-00

Accessing the AS/400 System with Java

August 1997

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix E, "Special Notices" on page 277.

First Edition (August 1997)

This edition applies to OS/400 Version 3, Release Number 2 and OS/400 Version 3, Release Number 7

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

Contents

Preface

In the past year, Java has become the hot new programming language. The reasons for Java's popularity are its portability and its ability to produce Internet enabled applications. This redbook is intended for anyone who wants to design and build Java applications that access the AS/400 system. We cover how you can use Java and the AS/400 system to build client/server applications and Internet based applets. We provide many practical programming examples with detailed explanations of how they work. These examples are also available for download from our Internet site. This redbook will give you a fast start on your way to using Java and the AS/400 system.

The Team That Wrote This Redbook

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

Bob Maatta is a Senior Software Engineer from the United States at the International Technical Support Organization, Rochester Location. He writes extensively and teaches IBM classes worldwide on all areas of AS/400 client/server. Before joining the ITSO 2 years ago, he worked in the U.S. AS/400 National Technical Support Center as a Consulting Market Support Specialist. He has over 20 years of experience in the computer field and has worked with all aspects of personal computers for the last 10 years.

Marshall Dunbar has been a Sr. Systems Analyst with Data Processing Services, Inc., in Indianapolis, Indiana for the last 11 years. Marshall has been a primary developer on DPS' Distribution and Financial software, DPS/9000. Working on the R&D team, Marshall has primarily dealt with PC to AS/400 integration using various tools. Recently, Marshall has implemented a Graphical User Interface for DPS/9000 and been intimately involved in DPS' Internet development and consulting efforts.

Stuart Foster has worked for IBM U.K. for 11 years. He started working in the mid-range S/370 arena with VM and VSE customers before moving to work on the AS/400 system six years ago as an application development specialist. He has been a pioneer of object technology implementations on the AS/400 system in the U.K. predominately using the VisualAge Smalltalk for AS/400 product set at a large number of AS/400 Business Partners and customers. Most recently, Stuart has been leading the work with AS/400 Business Partners to integrate Lotus Notes with their existing applications.

Paul Holm is an Advisory Software Engineer from the United States at the Partner's In Development Organization in Rochester, MN. He has focused exclusively on AS/400 Object Oriented Technology for the past five years. Before joining Partners in Development, he was a developer for DB2/400. His areas of expertise include DB2/400, Java, Smalltalk, and Object Oriented Analysis and Design.

Cheryl Pflughoeft is an advisory programmer on the Object Oriented Team in AS/400 Partners in Development (PID). Her current job is to assist business partners with their migration to Object Oriented Technologies focusing on Java. Prior to this, she was the coordinator for the PID early involvement programs for

ILE C, ILE RPG, and ILE COBOL. Cheryl started with IBM Rochester programming in-house manufacturing and warehouse applications on the S/3. She was part of the team that converted existing RPG applications to the S/38. Cheryl then moved on to the development team for Query/36 and Query/400 products, later becoming team leader for the Query/400 product.

Kevin Roberts is a computer engineering student, focusing in software engineering at the University of Michigan and expects to graduate in the spring of 1998. Kevin had the opportunity to work on the Object Oriented Team in AS/400 Partners in Development department at IBM as a co-op for seven months. There he focused on Java, JavaBeans, and accessing the AS/400 system with Java. He is currently back at the University of Michigan working for the School of Dentistry as a webmaster and system administrator.

Roger Wong is an IBM System Engineer and Sales Specialist in Hong Kong and the South China region. Roger has a lot of practical experience in AS/400 System Management, LAN technology, and Application Development. He has been working on the AS/400 Web, serving with Internet Connection for OS/400 since 1995. Roger supports IBM major Finance and Security accounts in South China. He has played a leading role in promoting IBM technology in the South China region.

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

Phil Coulthard
IBM Toronto Laboratory

Karen Eikenhorst
IBM Rochester Laboratory

Anita Leung
IBM Toronto Laboratory

Gary Mullen-Schultz
IBM Rochester-Partners in Development

Clif Nock
IBM Rochester Laboratory

Pramod Patel
IBM Toronto Laboratory

Schuman Shao
IBM Rochester Laboratory

Pamela Tse
IBM Toronto Laboratory

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

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

- Send us a note at the following address:

  `redbook@vnet.ibm.com`

Chapter 1. Object-Oriented Technology Overview

The purpose of this redbook is to introduce VisualAge for Java and to describe how it can be used to access the AS/400 system.

Java is an object-oriented language. In this chapter, object-oriented principles are reviewed but not explained in full detail. For a full introduction to object technology, one of the best books on the subject is *Object-Oriented Technology: A Managers Guide* by David A Taylor, published by Addison-Wesley (ZH20-9092, and ISBN 0-201-56358-4).

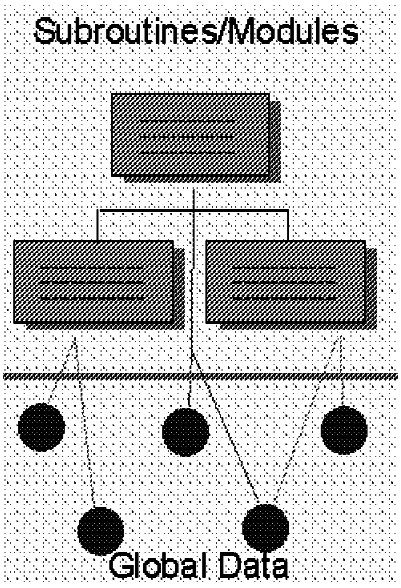
This chapter contains information about the following subjects:

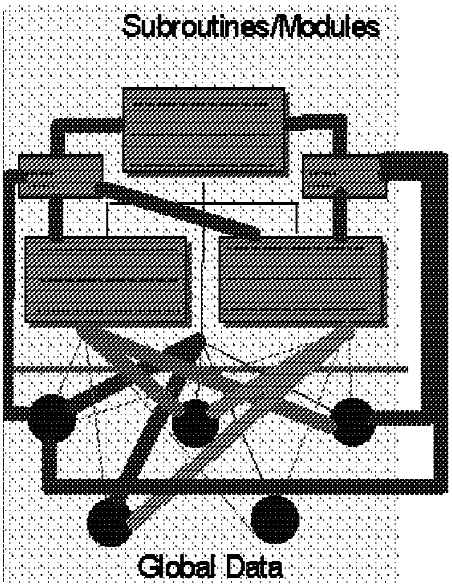
- Objects
- Classes
- Class relationships
- Polymorphism
- Benefits of object-oriented technology

1.1.1 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 increase in speed and power, the implementation of object-oriented technology becomes increasingly important.

Let's just take a moment to review the traditional application development scenario. I wonder how many of you recognize this scenario.





When our applications were designed about 20 years ago, they were designed to segregate the procedures from the data 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 comes in. This request is to extend the application a little. And the second request is to fix a small bug that has appeared. Maybe we were lucky this time as the impact on our total application was just to make a couple of minor modifications to the code modules, and to make a couple of extra links from a module to the database. But 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 would have great difficulty in retaining anything similar to the original design. But given that our programmers and designers have moved on two or three times from the original team, it is easy to see how the third picture in this little scenario has come about. I really do wonder how many of you recognize the picture on the right? Is this your application?

But wait a minute. From 20 years ago, we have moved from 2GL to 3GL, 3GL to 4GL, 4GL to case, case to uppercase and lowercase. Each of these transitions has made an incrementally better impact on software quality and design, and on programmer productivity. But as an industry, we are still left trying to maintain these creaking systems, and trying to add real value to business and to provide competitive advantage with Web based applications and so on. The industry has been looking for a new way to develop applications that better simulates the real world, not by splitting data and function apart and trying to mesh it back together again as we have been doing, but by keeping the data and procedures together

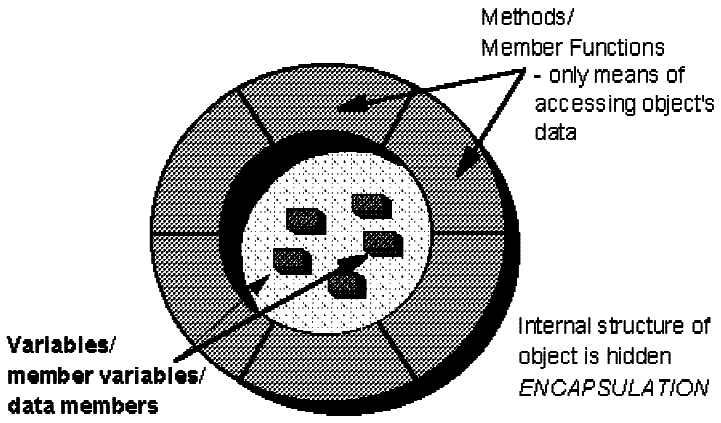
from analysis, through design, all the way to coding. This way of building systems is the object-oriented way. Let's take a look at what this actually means.

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 us, objects have a private side. The private side of an object is how it performs things, and it can do them in any way 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.

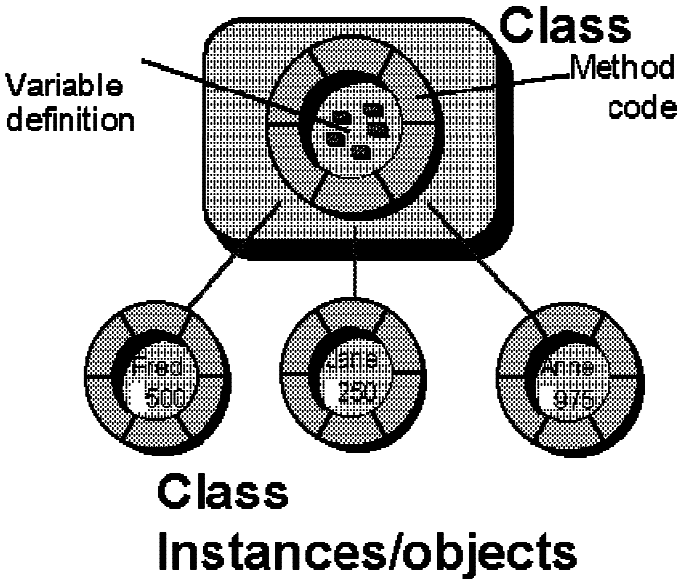


1.3 Classes

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, and is able upon receiving a message to perform any operation for which it has methods. It may also call upon other instances, either 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.

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.



To conclude,

- 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.

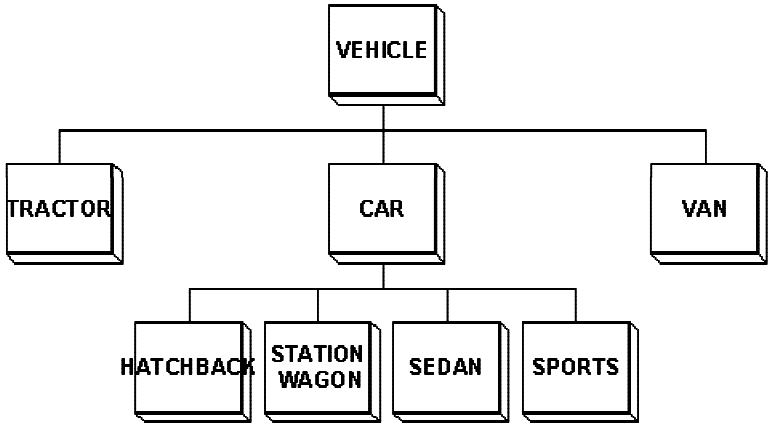
  The objects that belong to a class, called instances of the class, contain only their particular values for the variables.

1.4 Class Relationships

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

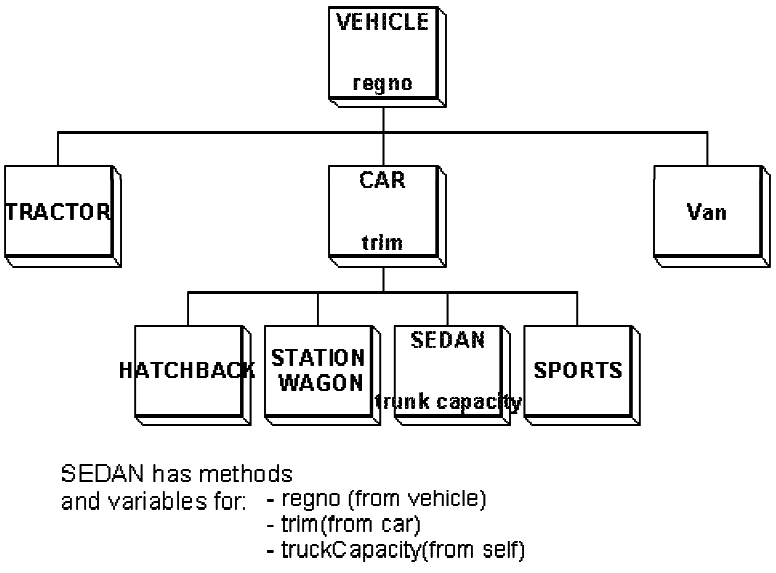
1.4.1 Specialization

By declaring one class to be a special case, or subclass of another, the subclass inherits all the method and variable definitions of its superclass. In the following class hierarchy figure, `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, Saloon, Estate, Sports). These last four classes are subclasses of `car` because they are a type of car.



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 the diagram below 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 the following figure, 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).



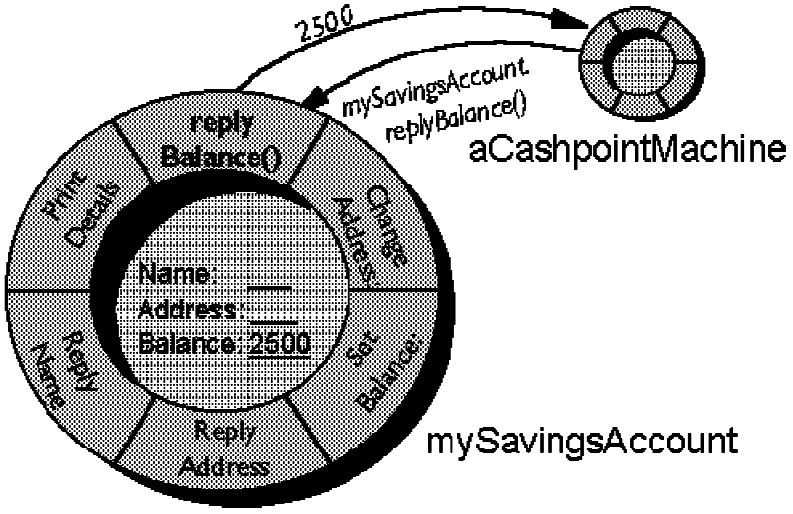
1.4.2 Composition

Classes can also be defined as components of one another. A laser printer might 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 might consist of a specified set of departments, several divisions can be combined into a business unit, and a company might 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.

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



1.5 Polymorphism

Understanding how object-oriented software works leads to realizing its vast benefits. One most important benefit 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 have been 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.

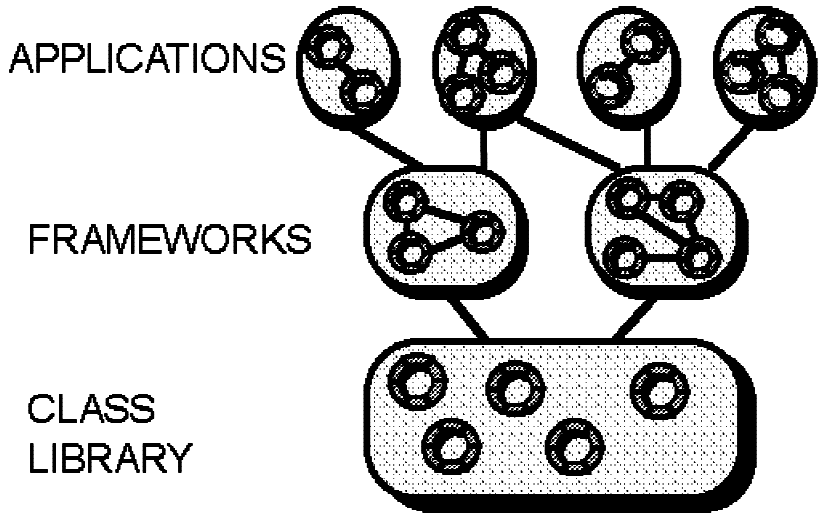
Closely related to polymorphism is the concept of dynamic binding. This idea stresses that because the sender of a message does not know anything about its receiver, determining the identity of that receiver can be left until the program is actually running. The advantage of dynamic binding is that it leaves all 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.

1.6 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 comes not from programming faster, but from programming less. The critical factor is to build up an inventory of reusable class definitions such that new applications can be constructed largely by recombining existing classes. The more reuse that is implemented, the greater the benefit.

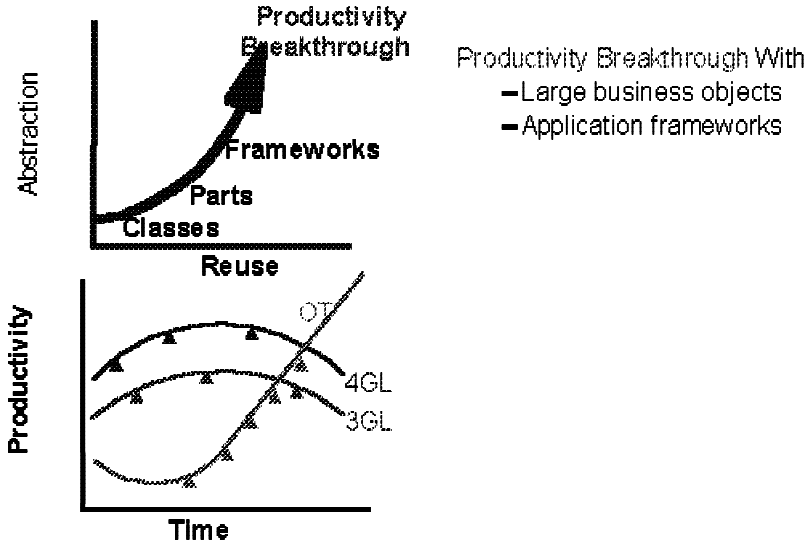
Encapsulation allows the building of entities that can be depended upon to behave in certain ways, and to know certain information. Such entities can be reused in every application that can make use of this behavior and knowledge. While it is possible to construct entities that are useful in many situations,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

Components 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 make use of components need not 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 out the common elements from the disparate pieces into one, and to 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 have coded similar pieces again and again.



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 will fit 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 will be maintained and extended in the future. And if an application-specific object has potentially a broader utility, one should consider designing it as a component that can be reused by other applications.



A large by-product of the reuse concept is increased quality. If 90 percent of a new application consists of proven, existing components, only the remaining 10 percent of the code has to be tested from scratch. This, in turn, leads to an increase in maintenance ease. If there are only 10 percent as many defects to begin with, there are a lot fewer bugs to chase down 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 covers VisualAge for Java with specific emphasis on the components most likely to be used by an AS/400 development team. It covers the following topics:

- The VisualAge family
- VisualAge Java overview
- Integrated Development Environment (IDE)
  - Java support
  - Navigating within VisualAge for Java
  - Visual Composition Editor
  - Team development
  - Applet viewer
  - Editor/Debugger/SmartGuides
  - Proxy builder
- The Enterprise Access Builders (EAB)
  - Data Access Builder (DAX)
- System requirements
- Summary

This chapter discusses various processes and windows that you use in the development of windows and applications using VisualAge for Java. All development for this redbook was performed using the Windows 95 client of the Enterprise Edition of VisualAge for Java. 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.

2.1 The VisualAge Family

VisualAge for Java is one of the newest member 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 Generator (4GL)
- VisualAge for COBOL
- VisualAge for RPG
- VisualAge for C + +
- VisualAge for Smalltalk
- VisualAge for Basic
- VisualAge for e-business
- VisualAge for PacBase
- VisualAge Financial Foundation
- VisualAge 2000
- VisualAge WebRunner

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
- 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.

VisualAge for Java is available in three versions:

- Entry
  - Free
  - One hundred class limit
  - Does not support the AS/400 Toolbox for Java
- Professional Edition
  - Includes the Integrated Development Environment (IDE)
  - Supports the AS/400 Toolbox for Java
- Enterprise Edition
  - Includes all Professional Edition support
  - Supports the AS/400 Toolbox for Java
  - Includes the Enterprise Access Builders
  - Team support will be included in the future

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

Visual Composition Editor component, which is common across all the 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

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), which 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 the need to run the compile statement (JavaC) from the command line. All applications can be 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 are able to debug code on the fly, spotting errors in their code with the debugger, changing it, and continuing without bringing the running application down...all within the VisualAge for Java IDE.

VisualAge for Java is an open IDE and 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 WWW. The JavaBeans support in VisualAge for Java also enables a developer to take an existing JavaBean (for example, from the WWW), import it 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 Borlands JBuilder).

Version 1 of VisualAge for Java supports JDK 1.1 (the most recent 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 should be able to run without change on the native AS/400 Java Virtual Machine when it becomes available.

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 datasources, 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.

VisualAge for Java comes with the following core components:

- Integrated Development Environment:
  - Hierarchy browser
    - Projects
    - Packages
    - Classes
    - Methods
  - Editor
  - Debugger
  - Applet viewer
  - Team support (Enterprise edition)
  - Java class libraries
  - Visual Composition editor
- Enterprise Access Builders (EAB)
  - Data access builder
  - CICS access builder
  - RMI builder
  - C + + builder

All of the preceding components utilize the JDK 1.1 and Java Virtual Machine Support of VisualAge for Java.

The AS/400 Toolbox for Java is not part of the core VisualAge for Java product and needs to be ordered separately. In a feature release of VisualAge for Java Enterprise Edition, the Toolbox classes will be included; see Chapter 8, “Java on AS/400 System” on page 209 for more information. The Enterprise Access Builders are part of the Enterprise Edition of VisualAge for Java.

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 metadata 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 will provide 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 is releasing 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 (known previously as the native method interface in JDK 1.0) 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 is, software that can be developed independently of the countries or languages of its users, and then 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, greatly improving 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 is the utility package containing 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 is the abstract windowing package that allows for 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, 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 1.x (currently 1.1.1 at June 1997) at:

`http://java.sun.com:80/products/jdk/1.1/docs/index.html`.

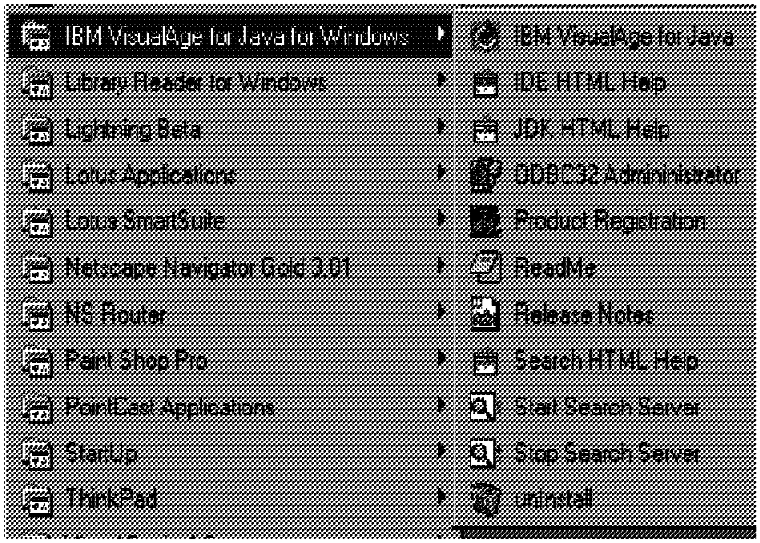
2.3.2 Navigating within VisualAge for Java

This section of the chapter 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
  - Unresolved problems
- Browsers:
  - Project
  - Package
  - Class

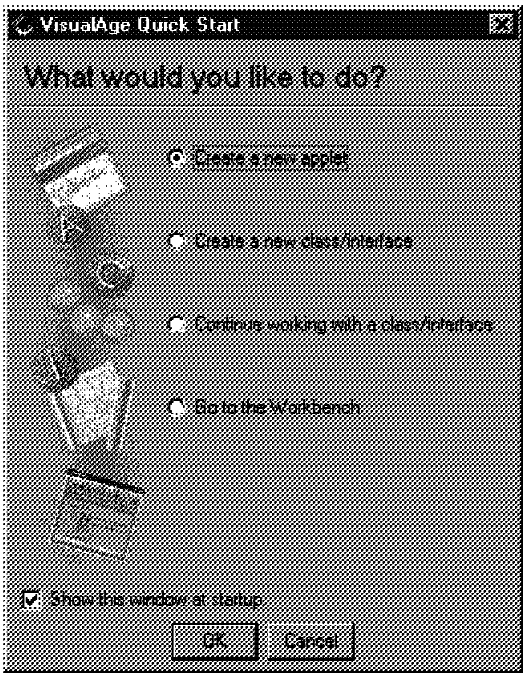
2.3.2.1 Starting VisualAge for Java

During the installation of VisualAge for Java, an item is added to the Windows 95 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.



During the startup process, VisualAge for Java loads the development image. As this image can be 8MB or larger (typically in the 15MB-25MB range), the startup process can take one to two minutes because the entire image must be loaded into memory. The development image is also known as the workspace and these two terms are used interchangeably in this chapter.

If this is the first time VisualAge for Java has been started, the first window displayed is the Quick Start window.



The Quick Start window provides a single point to perform most of the simple tasks. However, as you get more experienced using VisualAge for Java, you may decide to stop this window from appearing at startup.

Select **Go to Workbench** and press OK to go to the Workbench window.

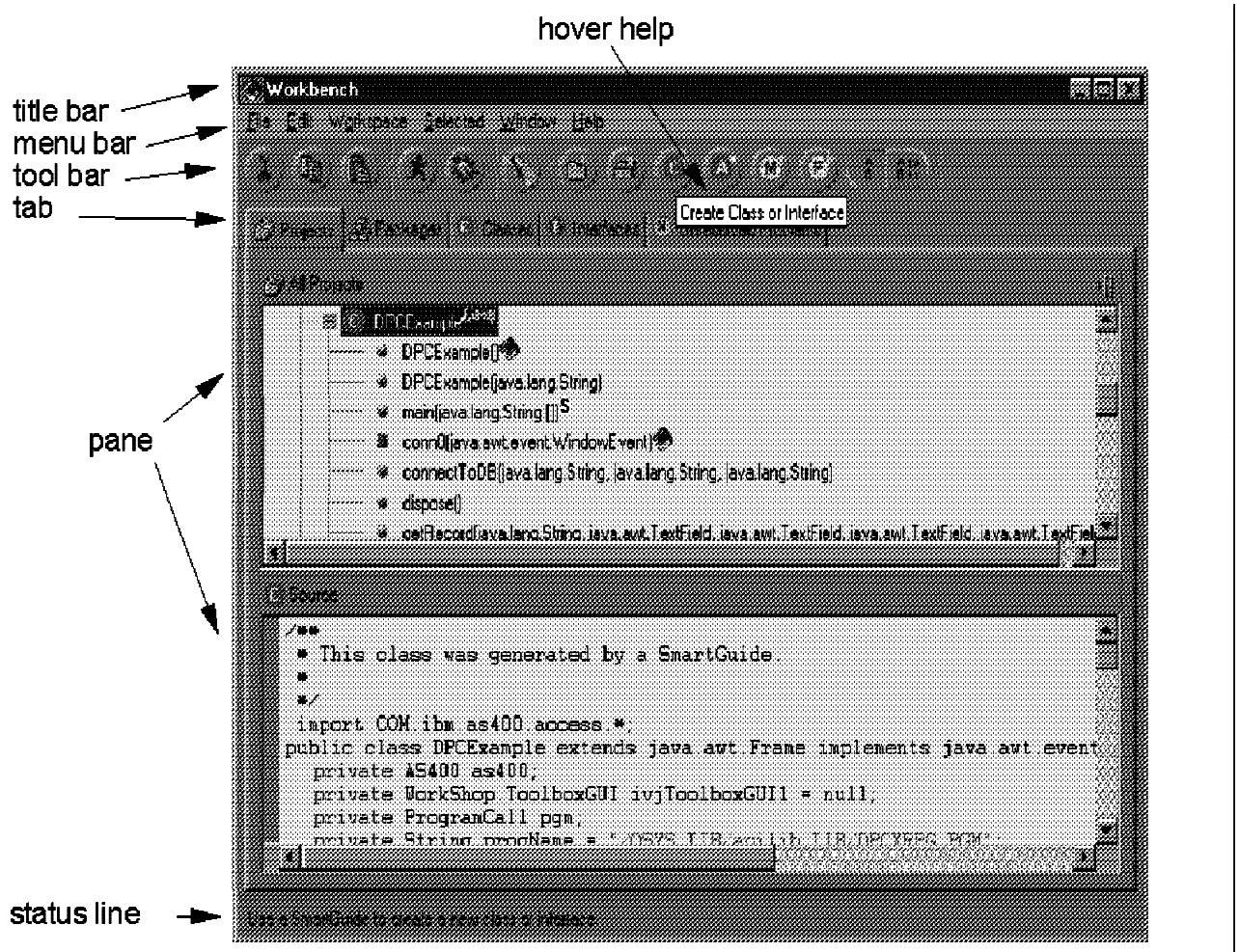


Figure 1. VisualAge for Java Workbench

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
- Menu bar
- Tool Bar:

  Provides fast access to menu items.
- Notebook tabs:

  Provides views of the four fundamental components of VisualAge for Java (projects, packages, classes, and interfaces) as well as a tab for displaying any unresolved problems.
- Panes:
  - Hierarchy pane:

Typically displays the component being browsed in context with its containing 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.

- 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
      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:

Figure 2. Project

Figure 3. Package

Figure 4. Class

Figure 5. Interface.

Figure 6. Executable class

2.3.2.3 Workbench Window

In the following Workbench window (Projects tab), the workshop project has been expanded to show its packages. One of these packages, the Workshop Package, has been expanded to show its classes and interfaces (classes only in this case). One of these classes, the DPCExample Class, has been expanded to show its methods. One of its methods, the connectToDB (java.lang.String, java.lang.String, java.lang.String) method, has been selected and its source is shown in the source pane.

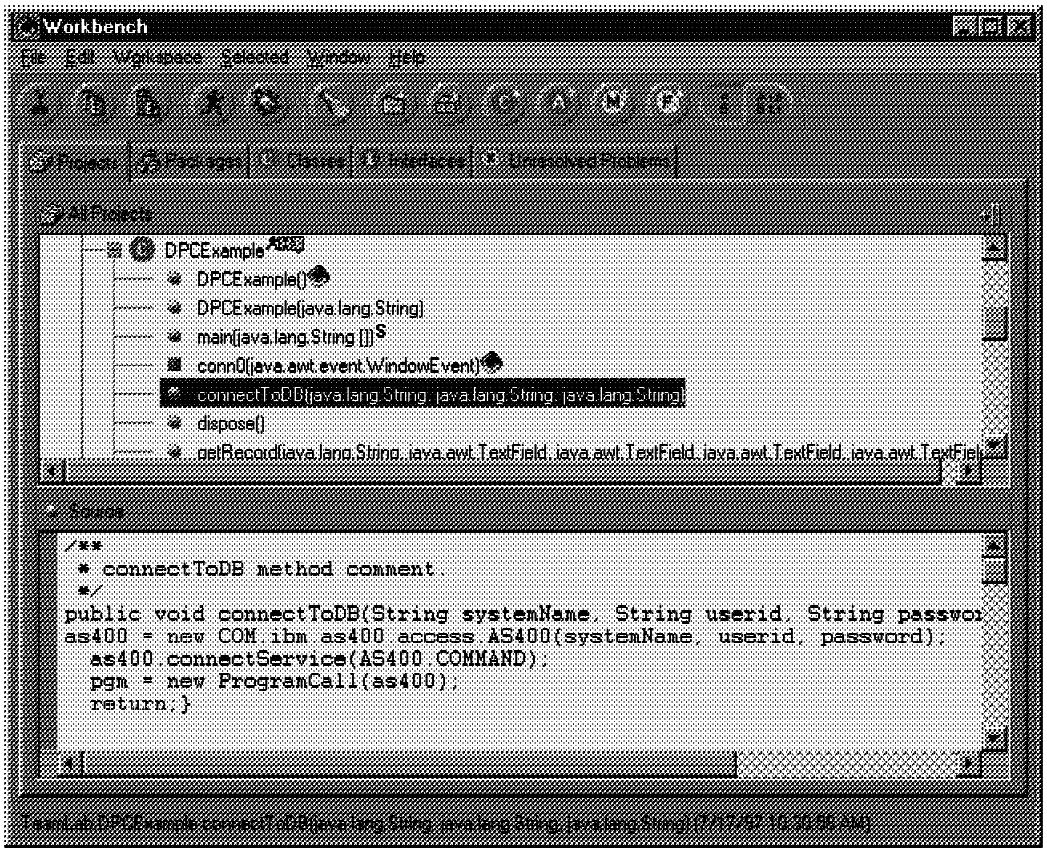


Figure 7. VisualAge for Java Workbench

2.3.2.4 Component Browsers

The next section discusses the four component browsers used extensively within VisualAge for Java (project, package, class, and interface). Each of the browsers is displayed from the Workbench window by selecting a component in the Hierarchy Browser Pane of the Workbench (for example, a package) and then selecting Open from its pop-up menu.

2.3.2.5 Project Browser

The project browser displays details on all the components within the project, including the packages, classes, interfaces, methods, and method source across the first three different views (tabs). The final tab on all the browsers is an Editions tab, which displays the version/edition information about this component enabling the developer (even in the Professional Edition of VisualAge for Java) to manage multiple versions/editions of packages/classes/interfaces/methods.

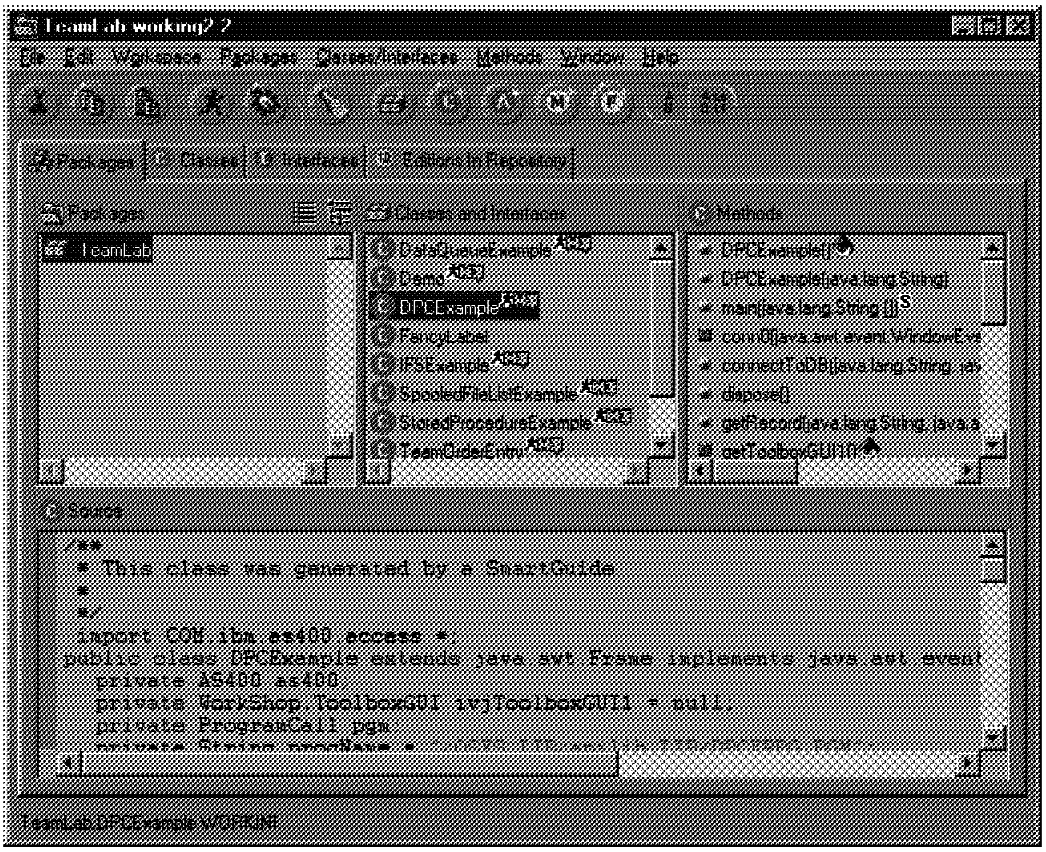


Figure 8. VisualAge for Java Project Browser

2.3.2.6 Package Browser

The package browser displays details on all the components within the package, including the classes, interfaces, methods, and method source across the first two different views (tabs). As with the project browser, the package browser has the editions tab to help manage multiple editions of the package.

The most used view shows the contained classes in the hierarchy in tree format.

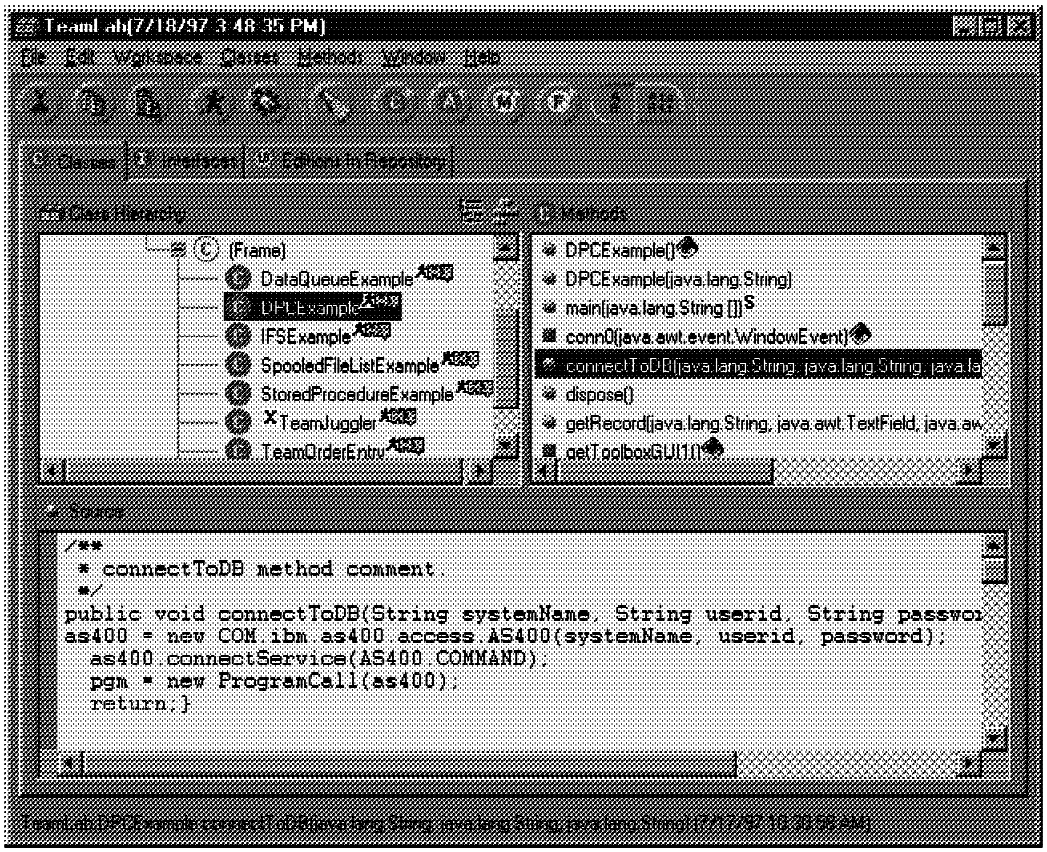


Figure 9. VisualAge for Java Class Hierarchy

There is also a graphical view of the classes contained in this package, although a high resolution screen is required to gain maximum benefit from this particular view.

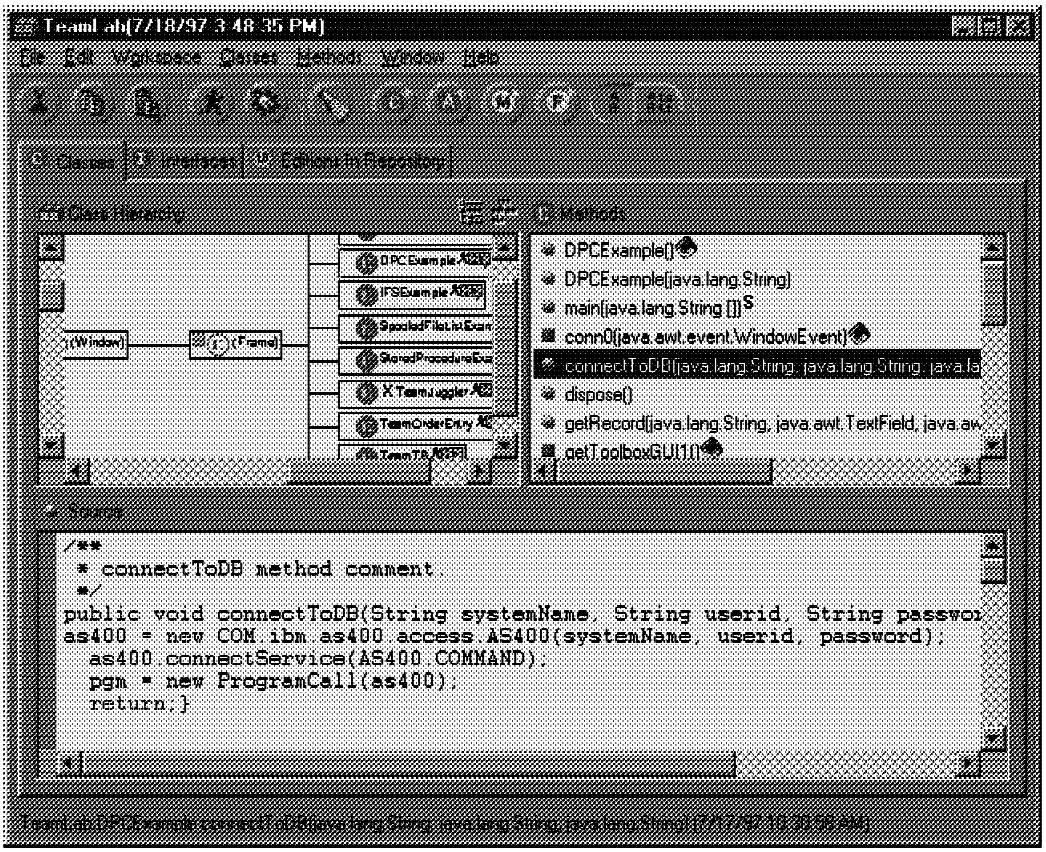


Figure 10. VisualAge for Java Class Hierarchy

2.3.2.7 Class Browser

The class browser is a little different in its implementation when compared with the project and package browsers. The class browser still displays all the sub-components (methods) it contains, the method source in the lower pane, and an editions tab for managing multiple editions of the components. But in addition, there are four extra tabs:

- The Hierarchy tab displays the position of the class in the hierarchy showing all the super-classes, both in tree and graphical format.
- The Editions in Repository tabs shows the editions available for the class in the repository.
- The Visual Composition tab is used primarily for the design of visual classes.
- The Bean Info tab displays information about the features that have been defined for the class (if any), and which also allows the Bean Info to be modified.

A lot of work is performed using the Visual Composition builder and this is discussed in the next sections. To open this browser, you select a class, click the right mouse, and select open.

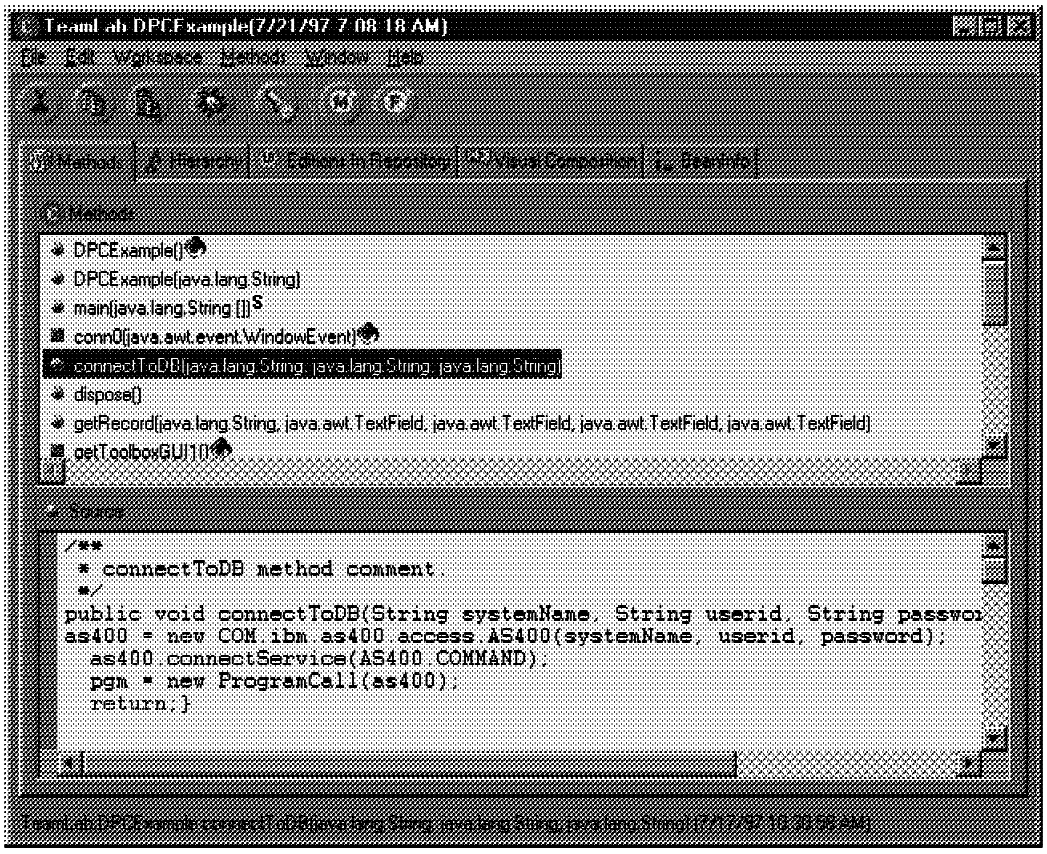
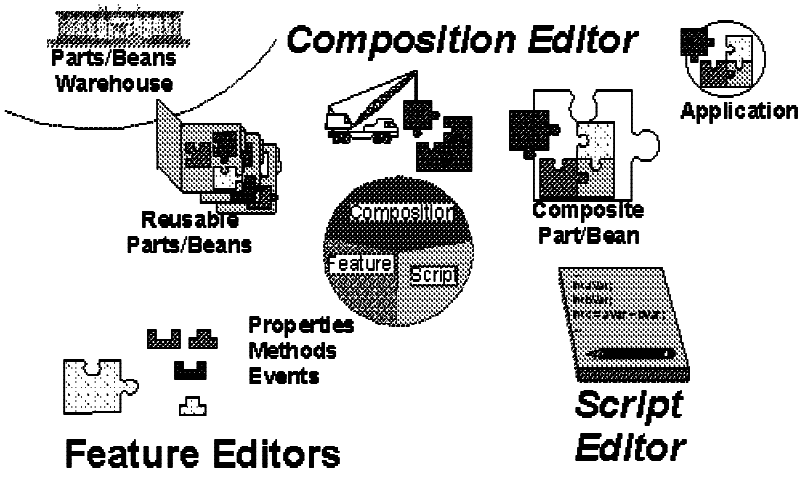


Figure 11. VisualAge for Java Class Browser

2.3.3 How It Fits Together

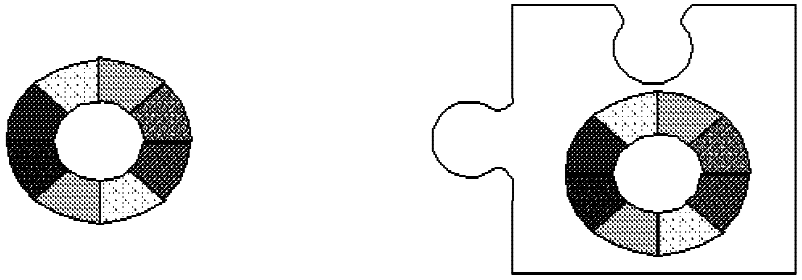


VisualAge for Java uses three basic components to build reusable JavaBeans and to use JavaBeans that may have been built by other tool vendors. These three components are the Visual Composition editor, the Features editor, and the Script editor.

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 Visual Composition editor to connect multiple beans together to perform the required function.

The product can also be used to develop reusable beans, or to 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

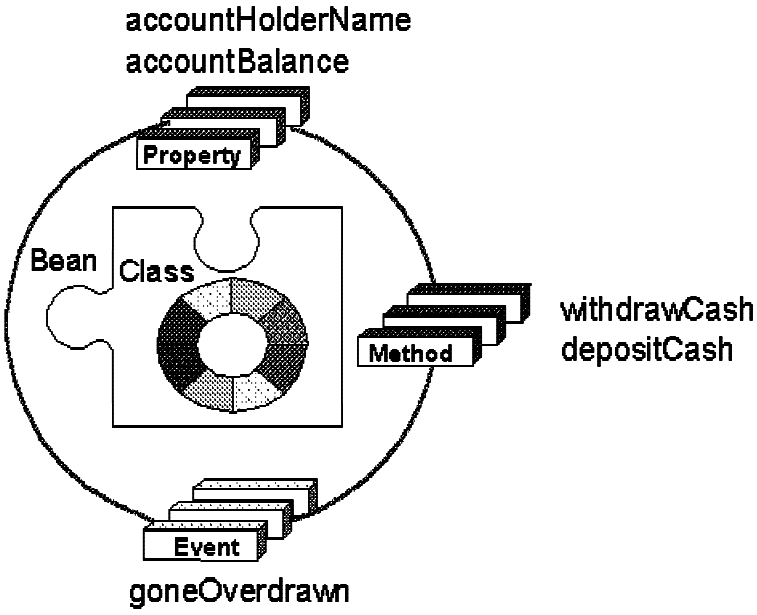


As discussed in Chapter 1, “Object-Oriented Technology Overview” on page 1, a class is a template for objects that have similar behavior (methods) and data elements (variables, properties). To use classes in visual builders (for example, VisualAge for Java, Symantic Cafe) the class needs to have features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

JavaBeans add standardized features and object introspection mechanisms to classes, allowing builder tools to query components (classes or groups of classes) about their properties, behavior, and events, thus allowing 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:

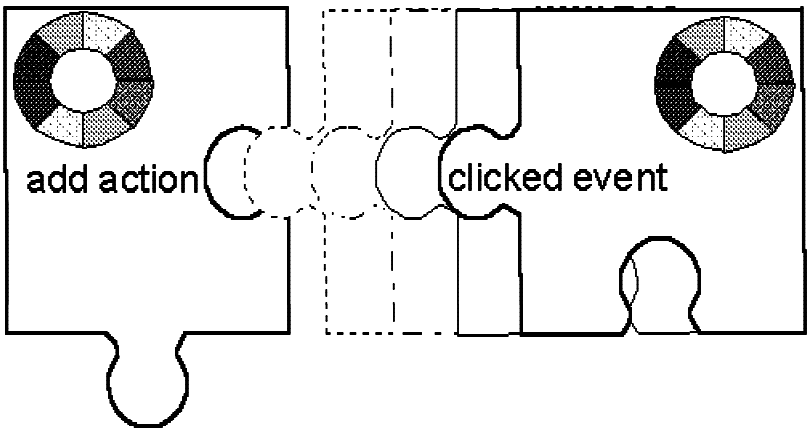
- Introspection - allowing a builder tool to analyze how a bean works.
- Events - allowing beans to fire events, and informing builder tools about both the events they can fire and the events they can handle.
- Properties - allowing beans to be manipulated programatically.
- Methods - allowing beans to perform functions implemented by the underlying classes methods.
- Customization - allowing a user to alter the appearance and behavior of a bean.
- Persistence - allowing beans that have been customized in an application builder to have their state saved and restored.



In the preceding diagram, an account class has been defined with functions/methods and variables. In addition to the account class definition, bean features have been defined for the following definitions:

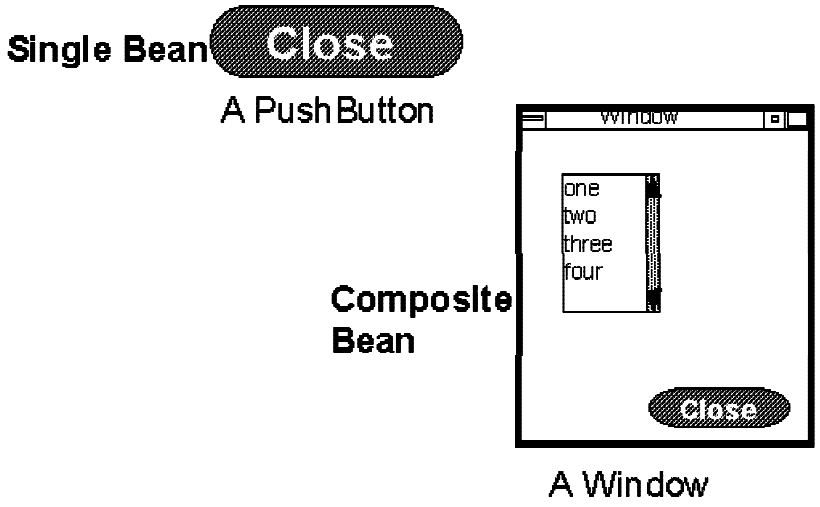
- Variable/Property
  - AccountHolderName
  - AccountBalance
- Method
  - WithdrawCash
  - DepositCash
- Event
  - GoneOverdrawn

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 listening) and takes its appropriate action (sends a letter informing the account holder of the account status and of the charges that have been applied).



In the preceding figure, there are two classes packaged as beans. The rightmost bean (for example, a push button) has a connection to the leftmost bean (for example, a listbox) and when the clicked event occurs, the listbox performs the add function. In VisualAge for Java, this connection is made through a series of simple steps that do actually 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 the preceding figure, the push button is a single bean, whereas the window is a composite bean made up of a push button and a list.

The preceding figure represents single and composite beans that are visual. Similar concepts apply to non-visual classes/beans. For example, an array contains a number of strings.

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

The following table summarizes the types of connections that the Composition editor provides. The return value is supplied by the connection's normalResult property.

If you want to..	Use this connection type	Color	Return Value
Cause one data value to change 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	Violet	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 bidirectional dark blue line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source. When your 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 unidirectional 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 bidirectional 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 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.

In the preceding discussion, the source and target points of a connection have been introduced.

A connection is directional; 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 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, so the connection line might 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.

2.3.3.2 Building a Sample Application Window

The objective of this section of the chapter 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 where ordering them in a parts ordering application.

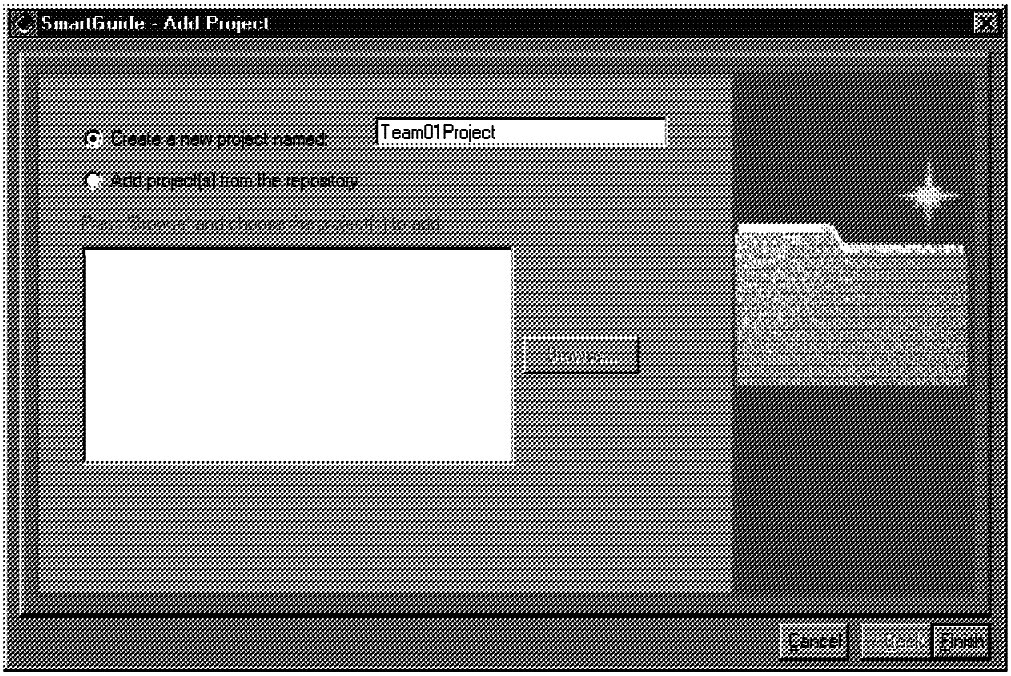
Earlier in this chapter, VisualAge for Java was started and you navigated past the Quick Start window to the Workbench window.

```
From the Workbench window,
      select the Selected menu item
           then select the Add Project... submenu item
```

Note: In all future scripts, selected sub-menu items are formatted the same as this: Selected--Add Project...

The SmartGuide - Add Project window is shown:

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



A project named Team01Project is created; you are returned to the Projects tab of the Workbench window, and the Team01Project is highlighted.

Open the Team01Project: Bring up the Team01Project's pop-up menu and select Open:

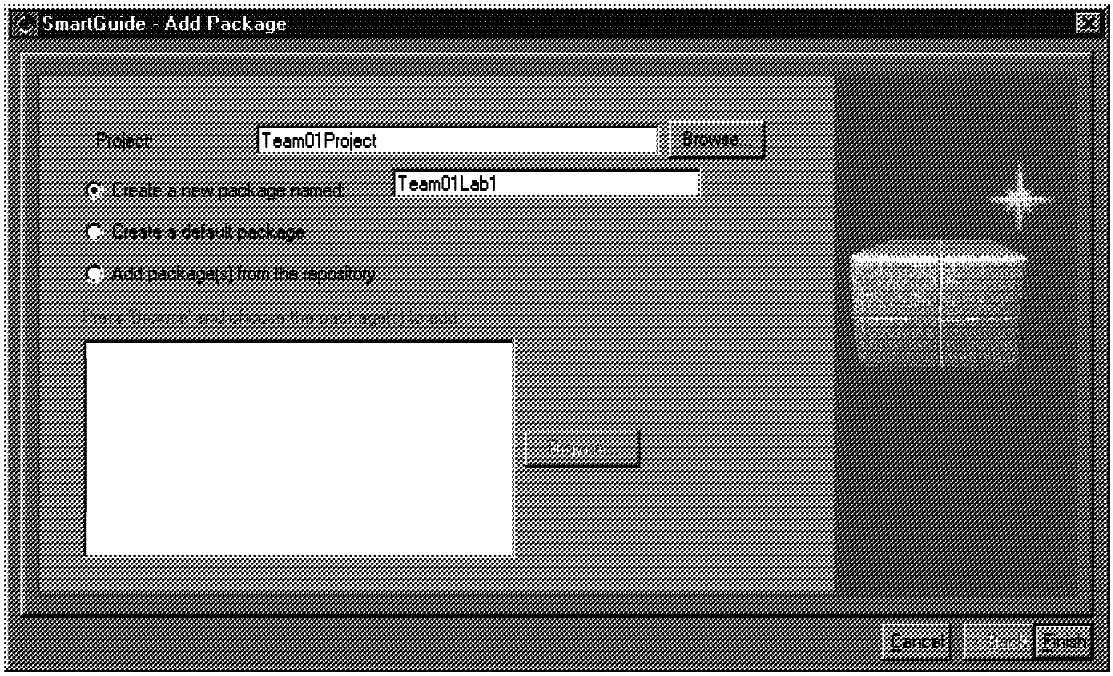
The Team01Project window opens. The title of this window is Team01Project(dd/mm/yy hh:mm::ss am). The time stamp element of the window title is an indication of the date/time when this edition of the project was created. If the Team01Project window does not open up maximized, then maximize it.

Add a New Package (Team01Lab1) to the Team01Project Project

```
To add a new package, click the right mouse button in
the Packages pane and select add package from the pop-up menu.
```

Select the Packages--Add Package... menu item from the Team01Project window.

Enter Team01Project as the name of the project to which this class is added (this is the default). Enter Team01Lab1 as the package name and click Finish to create it.

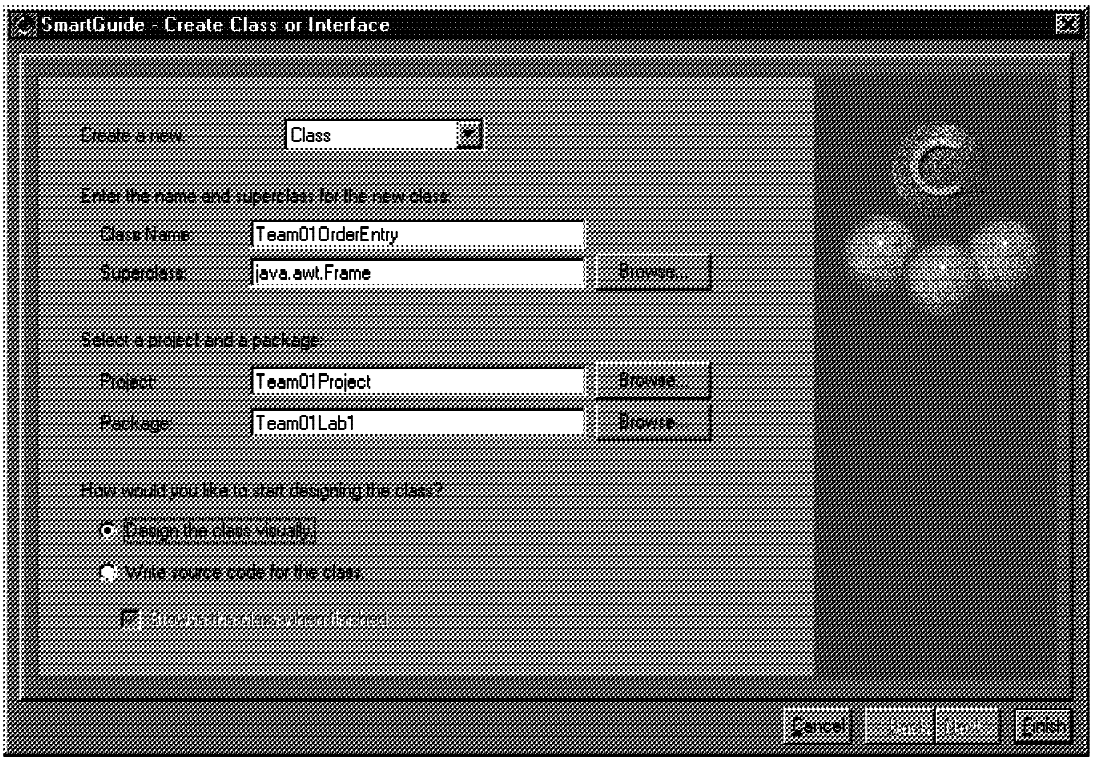


A new package, Team01Lab1, is created in the Team01Project. The new package is shown highlighted in the Packages pane of the Team01Project window.

Add a New Class (Team01OrderEntry) to the Team01Lab1 Package: Select the Classes/Interfaces--New Class/Interface... menu item. Enter Team01OrderEntry as the class name. Enter java.awt.Frame as the superclass name.

Notice that class names are case sensitive. You are creating a visual class, and most visual classes have java.awt.Frame as their superclass.

Select the Design the class visually radio button. Click Finish to create the class.



A new class, Team01OrderEntry, is created in the Team01Lab1 package in the Team01Project project. The new class is shown in the classes and interfaces pane of the Team01Project window, and the Visual Composition editor for the Team01OrderEntry class is opened and in focus.

Maximize the Team01Lab1.Team01OrderEntry(dd/mm/yy hh:mm:ss am) window.

The title of the window is Team01Lab1.Team01OrderEntry(dd/mm/yy hh:mm:ss am). The suffix time-stamp element of the window title is an indication of the date/time when this edition of the class was created, and the prefix shows you which package.class you are working on.

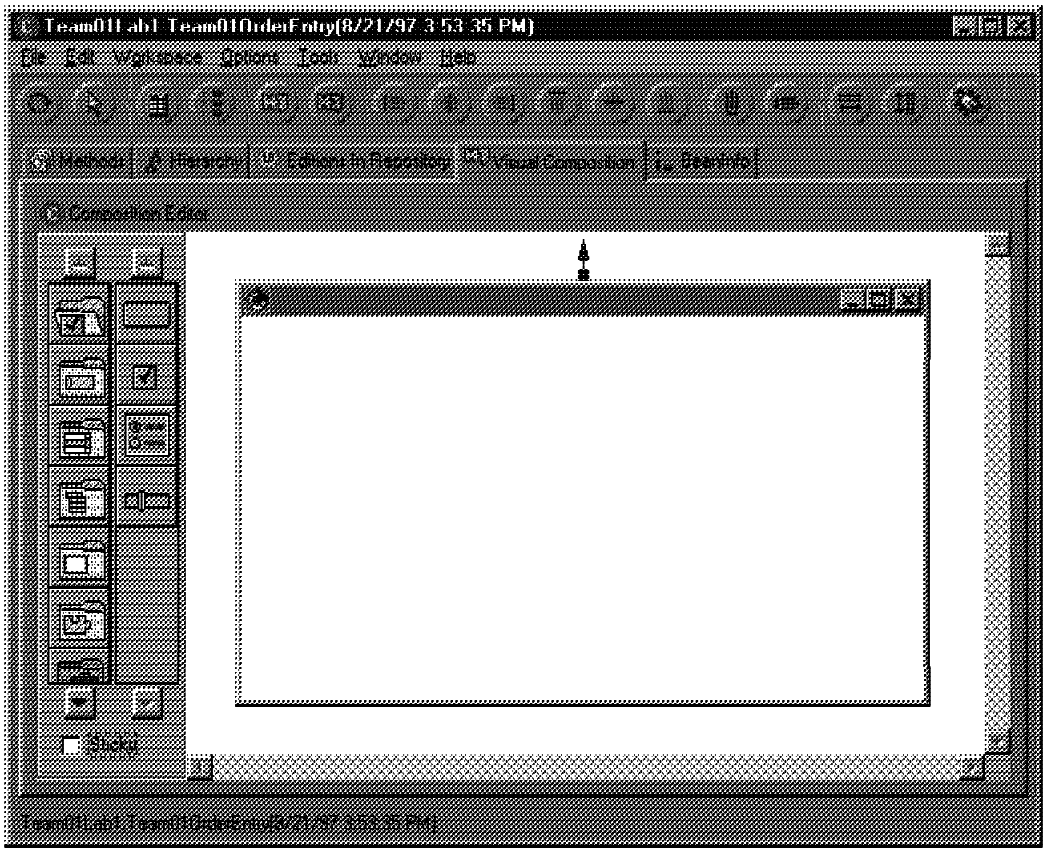
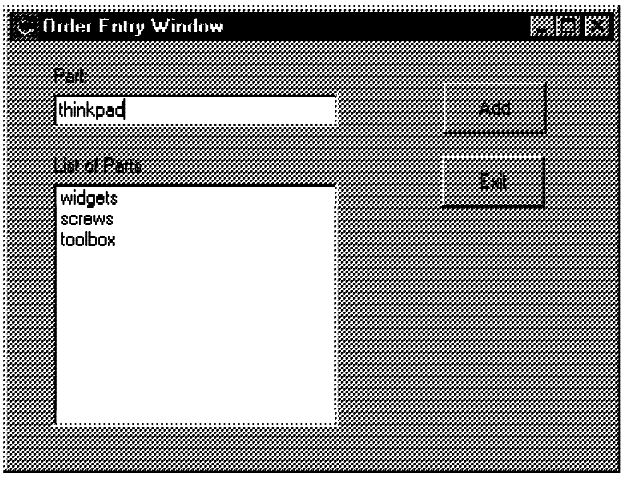


Figure 12. Visual Composition Editor

Take a moment to review the preceding window to see the various components on the Visual Composition editor.

- Window: The window being built, usually in the top left corner of the free-form surface.
- Free-form surface: The white space surrounding the window being built. The free-form surface is usually used to drop non-visual parts (for example, a timer) that you want to utilize in your class but that you do not want to show to the end user at run time.
- Parts palette: The area on the left of the Visual Composition Editor window that contains:
  - Parts categories (the leftmost column/scrollbar) - a container for parts.
  - Parts (the rightmost column/scrollbar) - the parts.
  - Sticky: The check-box at the bottom of the parts palette. The sticky check box enables you to load the cursor with a part and to perform multiple drops of that part onto the free-form surface or window.

Add the Visual Components to the Window: The completed window at run time for this section looks similar to this:



Note: Do not be too concerned with the placement and alignment of parts as you are building the window. Later, we will make it look good. Also increase the size of the window at this point. This makes it easier to add parts. To size a part, click on it to select it. There is a block in each corner, which indicates that it is selected. These are called re-size handles. Move the mouse pointer over one of these re-size handles and press and hold the left mouse button to drag the part to its desired size.

Now we build the Graphical User Interface by selecting parts from the parts palette and placing them on the window. Use the completed window shown previously as a guide. Add:

- One TextField
- Two buttons
- One list
- Two labels

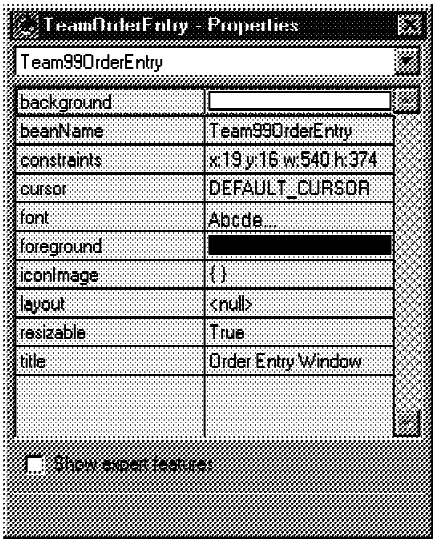
Note: Use the hover help to recognize the parts in the parts palette. Move the mouse pointer over the top of the part and view the online help.

Move the cursor over the parts palette and left-click on the data entry category. Left-click on the TextField part. This loads the cursor with the TextField. Move the cursor over the window, near to the left edge (about 10% in) and 20% down. Left-click to drop the TextField into position. Now left-click on the Buttons category. Left-click on the Button part. Move the cursor to the right side of the window, about 25% in from the right edge and 20% down and drop a button with a left-click, and then drop another button just below the first one. Left-click on the lists category and then on the list part.

Move the cursor to the left side of the window about 10% in and 50% up. Left-click to drop the List. Left-click on the Data Entry category, left-click on the Label part.

Move the cursor to just above the TextField that you dropped earlier and left-click to drop it. Add another TextField just above the list that you dropped earlier.

Make the Window Look Good: Move the cursor over the label just above the TextField. Hold the Alt key and left-click (Alt-left). You can now type in the text that is shown on the label. Type **Part** and then left-click on the free-form surface to stop editing the label text. Move the cursor over the other label and Alt-left. Type **List of Parts**. Move the cursor over the top button and Alt-left. Type Add. Move the cursor over the bottom button and Alt-left. Type Exit. Move the cursor to the free-form surface and left-click to stop editing. Move the cursor over the Window title bar and double-click to open the Properties Dialog for the window. Left-click in the value column of the title row and type in **Order Entry Window** for the title of the window. Close this window by click on the x in the upper right corner.



Move the cursor over the TextField and left-click. Move the cursor over the list and then left-click while holding down the Ctrl key (Ctrl-left). Both parts are now selected. You can tell a part is selected by the re-size handles. When you have two or more parts selected, the alignment smarticons are enabled.

This allows you to align left, center, right, top, middle, bottom, space horizontally, vertically, same width, and height.

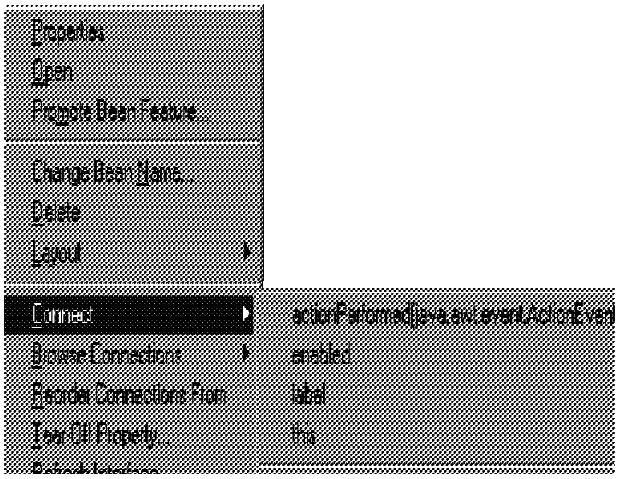
Note: To learn the function of a smarticon, move the mouse over it and view the help text.

Move the cursor to the bottom right re-size handle of the list. Click and hold the left mouse button, and drag the re-size handle to make the list wider. Release the button when you are happy with the width of the list. Notice that the TextField stretches as well. Now select the Match Width smarticon. This makes

the TextField the same width as the list. The list is the part of reference because it has the darker re-size handles. Still with the list and Textfield selected, click on the Align-Left smarticon to align their left sides. Move the cursor to the free-form surface and left-click to deselect the part. You now know how to align and size parts. Using your own style and GUI building skills, align the various parts to make the window look cool! Test the application. It does not have any function but you can see how it looks. Click on the Test smarticon and then click on run on the Command Line Argument window to test the application.

You receive a message saying "Generating run-time code". This is the Visual Composition editor saving the layout information into Java code. The developed window is shown. Review it and close the window. You return to the VisualAge Visual Composition window.

Add the Function: Move the cursor over the Exit button and left-click to select it. With the cursor still over the Exit button, right-click to bring up the Buttons pop-up menu, select Connect and action.actionPerformed(java.awt.event.ActionEvent).



Selecting Connect brings up the features available to you as defined on the button JavaBean. The action.actionPerfomed feature 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.

The spider is shown. The spider allows you to connect parts (beans) together. Move the spider to the window title bar and left-click. The connection target pop-up window is displayed.

What happens when the Exit button is pressed? You want the window to be closed/disposed.

Select dispose() from the pop-up window. A green connection is displayed between the Exit button and the window.

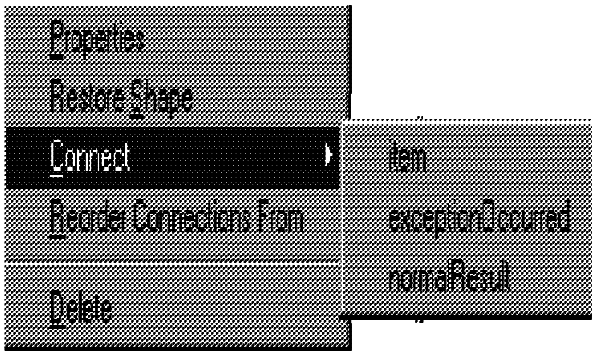
You are now ready to visually perform the function to add text entries from the TextField to the list. Move the cursor over the Add button and left-click to select

it. With the cursor still over the Add button, right-click to bring up the Buttons pop-up menu, select Connect, and action.actionPerformed. Move the spider to the list and left-click. The connection target pop-up window is displayed.

What happens when the Add button is pressed? You want the text/string entered in the TextField to be added to the list. Select add(java.lang.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 have told VisualAge that when the Add button is pressed, something is added to the list...but you have not specified what is added. You can do this now.

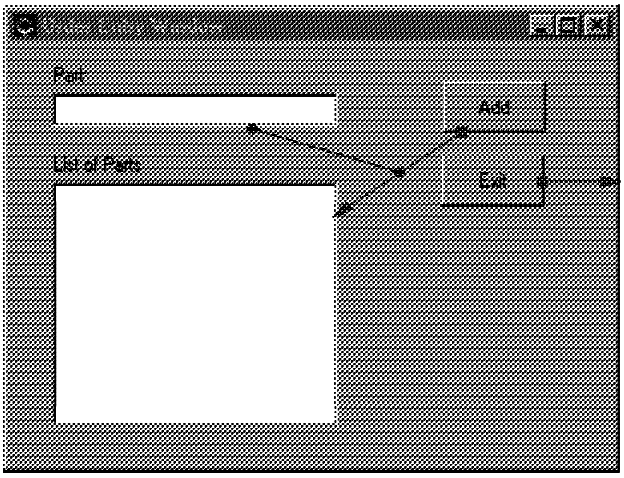
Move the cursor over the dashed green connection from the Add button to the list. Left-click over the connection. Selection handles are shown along the connection to show that it has been selected. With the cursor still over the connection (but not on a selection handle) right-click to bring up the Connections pop-up menu and select Connect--Item.



Move the resulting spider over the TextField. Left-click and select Text. A purple arrow joins the TextField to the green connection. Do not forget that VisualAge colors each connection depending on its type. You have now completed the window for this section.

Test it out by selecting the Test smarticon. Enter some values in the TextField and check if the Add button adds them to the list. Test the Exit button.

This is how our completed VisualAge window looks; your window may look similar.



Version Your Application: Left-click on the Hierarchy tab. The class hierarchy is displayed showing the Team01OrderEntry class and its super-classes. Left-click on Team01Lab1.Team01OrderEntry to select it. Select the Classes--Version... menu item. Make sure the Automatic radio button is selected and click on the Finish button.

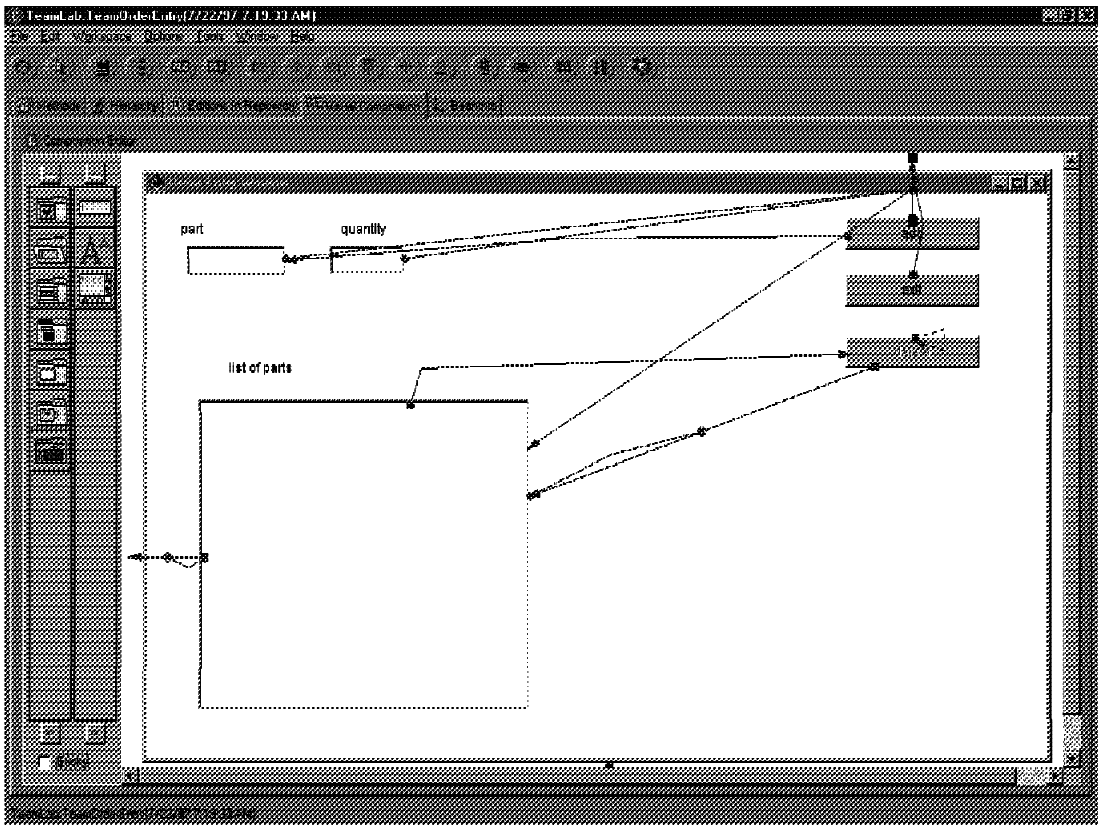
Your class is now versioned. You can change the class at any time but you can also go back to this version of the Team01OrderEntry class whenever you need to.

2.3.3.3 Extending the Application

The application is now extended and the following actions are performed:

- Add a quantity field.
- Modify the behavior so that the Add push button invokes a script to concatenate the part and quantity details and then displays them in the list.
- Add a Delete button to delete existing entries in the list.
- Enable/disable the Delete button when an item is selected/deselected in the list.
- Add a Java script breakpoint and modify code when the breakpoint is invoked.

At the end of this section of the chapter, the completed development window should look similar to this:



Add Standard GUI Parts: Click on the Visual Composition tab to get back to the Visual Composition Editor.

- Add a Delete button to the window.

  Disable the Delete button. To modify a component's properties, double left-click on the component to bring up its Properties window. Check on expert features, then single left-click inside the value column for the property name you want to modify (to bring focus to the value) and then change the value as appropriate. Change the enabled property to false and the label to Delete.

- Add a TextField that allows the quantity of parts to be input (called the Quantity TextField later).

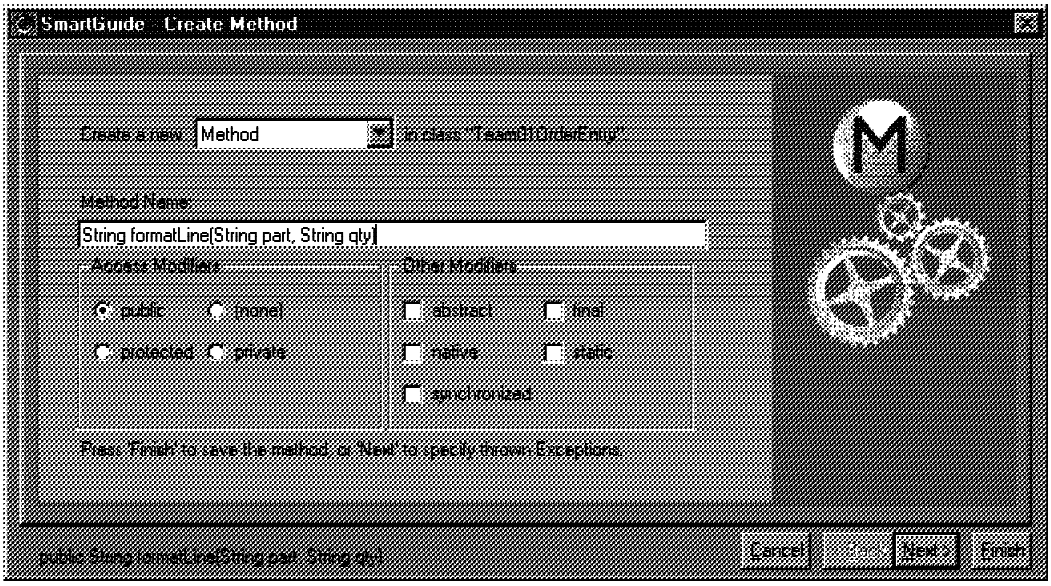
  Place it level with and a little to the right of the part TextField. You may need to move some components around, and you may even have to make the window a little larger. To re-size any component, single left-click and hold down over a re-size handle and drag the mouse to the required size. Release the mouse button to end re-sizing.

- Add a label part and change its text property to Quantity.

Delete Connections: You can add items from the TextFields to the list using a script in this section. Therefore, the current connection from the Add button (action.actionPerformed) to the list (add(java.lang.String)), taking the Text property from the TextField, is removed. Move the cursor to the green connection from the Add button to the list (the one just described). Single left-click over the connection and re-size handles should appear on it. If they do not appear, you are not exactly over the connection; move the cursor and try again if this is the case. Press the Delete key and select OK when prompted by

the confirmation message. The connection and any connections it was supporting are deleted.

Write a New Java Method to Add Part/Quantity Text to the List: To add both the part and quantity text to the list, write a script to concatenate the two TextFields together. Single left-click on the Methods tab and select the Methods--New Method... menu item. In the Method Name entry field of the Method Properties window, enter String formatLine (String part, String qty) and select the Finish button.



A new method called formatLine is created that takes two String variables (part and quantity) and returns a string (the concatenated string).

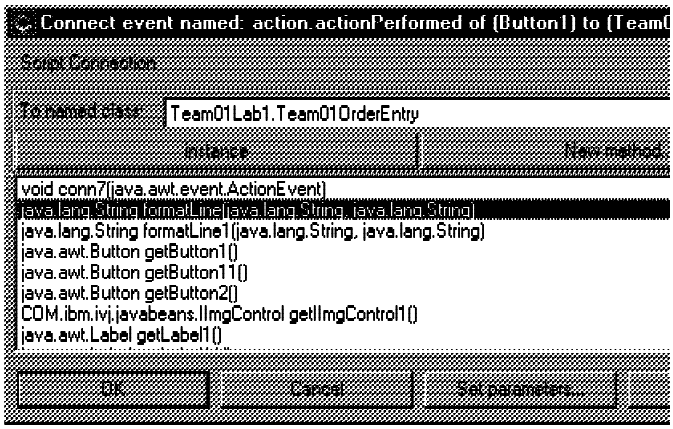
```
            /**
* This method was created by a SmartGuide.
* @return java.lang.String
* @param part java.lang.String
* @param qty java.lang.String
*/
public String formatLine(String part, String qty) {
return;
}
```

Modify the formatLine method source as shown in the following example and use Ctrl-S to save the modified method.

```
            /**
* This method was created by a SmartGuide.
* @return java.lang.String
* @param part java.lang.String
* @param qty java.lang.String
*/
public String formatLine(String part, String qty) {
return part + " " + qty;
}
```

Single left-click on the Visual Composition tab to return to editing the window.

Make the Connections to Add the Part/Quantity to the List: Start the connection from the action.actionPerformed event of the Add button and drop the spider over the free-form surface. The free-form surface is outside the window that you are building. The connection target pop-up menu appears. Select the Event to Script... menu item. The following window is shown that lists all of the scripts you can connect to for the class being developed.



Select the java.lang.String formatLine(java.lang.String, java.lang.String) script and select OK.

A light green dashed connection is shown from the Add button to the free-form surface. A dashed connection means that the connection requires parameters that have not yet been supplied.

- Connect the part parameter for the preceding light green dashed connection to the text property of the part TextField.
- Connect the quantity parameter for the preceding light green dashed connection to the text property of the quantity TextField.
- Connect the normalResult parameter for the preceding light green dashed connection to the addItem(java.lang.String) method of the list.
- Try testing the application to see if you can add parts to the list using the script that you just created.
- Return back to the Visual Composition editor.

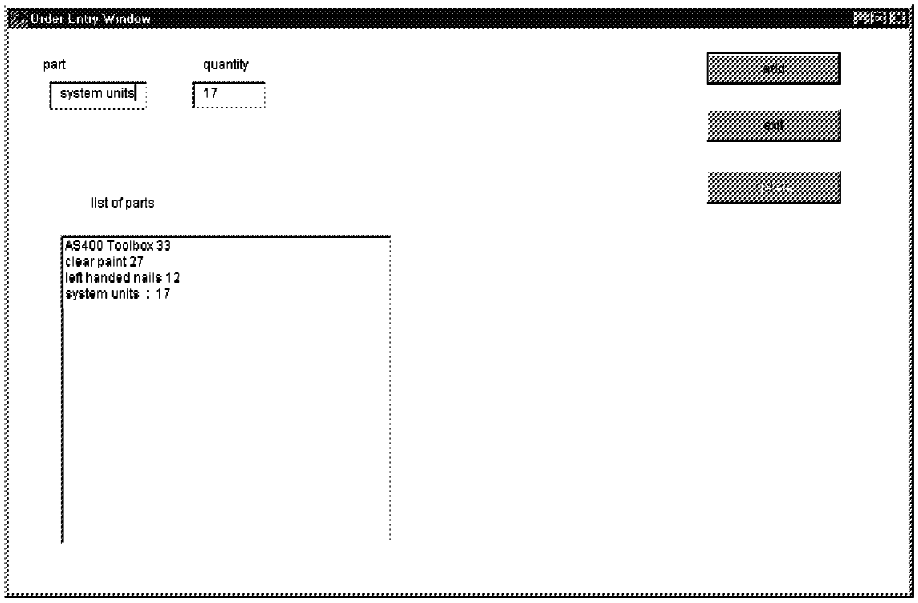
Make the Connections for the Delete Button

- Connect the action.actionPerformed event of the Delete button to the remove(java.lang.String) action of the list. There is a dashed green line connection from the Delete button to the list.
- Click on the list to select it.
- Connect the selectedItem property of the list to the item parameter of the connection between the Delete button and the list. The green connection now should become a solid green line.
- Connect the itemStateChanged event (that is, an event is fired when the selected item changes) of the list to the enabled() action of the Delete button. There is a dashed green connection between the list and the delete button.

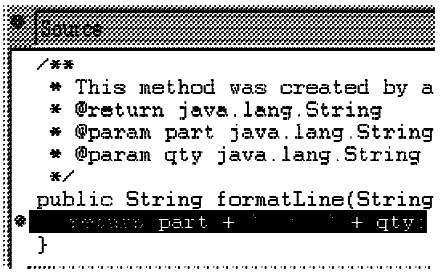
- Double-click on the dashed green connection to open it. Click on Expert features and then use the Set parameters button to set the value to **True**. This enables the button when an item is selected in the list.
- Connect the action.actionPerformed event of the Delete button to the enabled() action of the Delete button. You can leave the parameter default to false; this disables the button.

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

Debugging Code, Setting Breakpoints, and Changing Code "on the Fly": With the Test window still running, return to the class browser/editor Methods tab page. Modify the code of the formatLine method so that the line return part + " " + qty; now reads return part + " : " + qty. Save the part with Ctrl-S. Add another part/quantity item and see that the code you changed was used to add this new part. Your test window should look similar to this:



With the Test window still running, return to the Class browser/editor Methods tab page. Move the cursor to the method source pane on the return part + " : " qty; line. Right-click the mouse and select **Insert/Remove Breakpoint**. A blue breakpoint marker is shown. 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.



Add another part/quantity item to the running test window. The debugger window is shown.

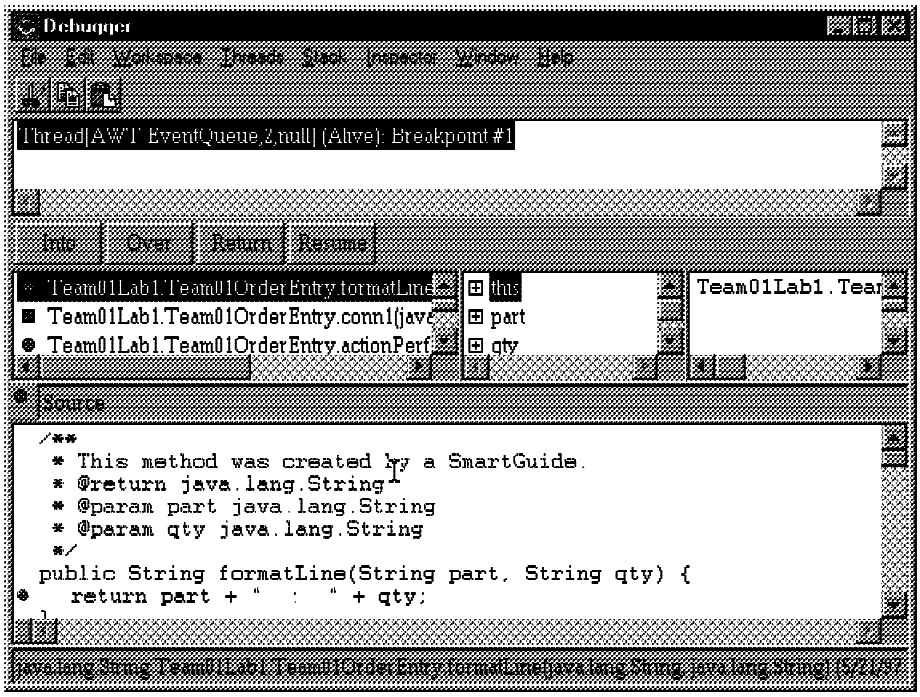
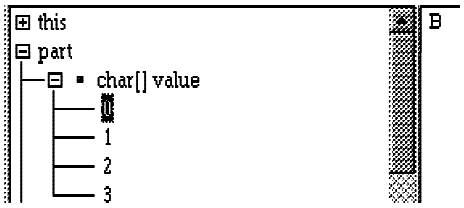


Figure 13. VisualAge for Java Debugger

The code has stopped prior to executing the statement. The uppermost pane shows the current thread when the debugger was invoked. In the three middle panes, the left-hand pane shows the call stack with the most recent method at the top (call stack pane), the center pane shows the variables that are accessible (variables pane), and the right pane shows the value of the currently selected variable (variable value pane). The bottom pane shows the current line in the current method (method source pane). Single left-click on the part variable in the variables pane. The variable value pane is updated and displays the string value of whatever you typed into your part TextField. 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 (this assumes you typed in a part with two or more characters). The variable value pane shows you the values of the first two characters you typed into your part TextField.



Now single left-click on the Team01Lab1.Team01OrderEntry.connx(java.awt.event.ActionEvent) entry in the call stack pane (notice your entry is similar to conn1). You see that the variables pane, variable value pane, and method source pane are all updated. In the method source pane, the actual code that called the current method is highlighted (this.formatLine(getTextField12().getText(), getTextField11().getText());). Navigate to the Methods pane of the Team01OrderEntry Class editor/browser and traverse the list of methods and select the connx (for example, conn1) method. This is the method that you are currently looking at in the debugger. Return to the debugger window. Select the top entry in the call stack pane, Team01Lab1.Team01OrderEntry.formatLine(java.lang.String). Now modify the code so that the string " : " now reads " :- ". Ctrl-S to save the method. Select the Resume button. The debugger window blanks out as that thread has now run to completion. Close the debugger window and navigate back to the running test window. Your part/quantity entry has been added and with the " :- " separator between the part and quantity.

Anywhere you have a method source window, you can modify the method, save it, and it runs immediately with the updated code. Now select an entry in the list of the running test window. The Delete button is now enabled. Test it out by deleting the entry. The entry is deleted and the Delete button is disabled (until you select another entry in the list).

Close it Down and Version: Close the Team01Lab1.Team01OrderEntry Class editor/browser. Version the class, either accepting the default version name or enter your own. Save the workspace.

2.3.4 Team Development

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

Important Information

Team support is not available in the currently released product, but is planned to be made available by the end of 1997.

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 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 checkpointing components within a development cycle.

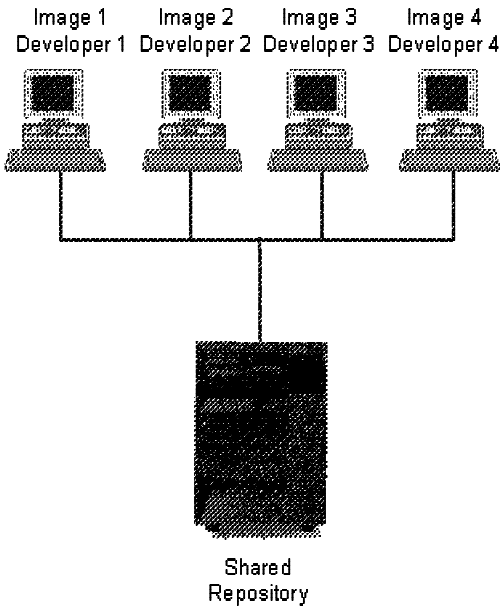
With the Enterprise Edition, multiple developers can, if appropriate, work on any component (project, package, class, or method) concurrently. In a normal check-in, check-out philosophy, this is impossible but within the VisualAge for Java Enterprise Edition, this can be achieved. Despite this flexibility, 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 shares a common repository allowing all the work to be shared and accessed concurrently, online and in real time.

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

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

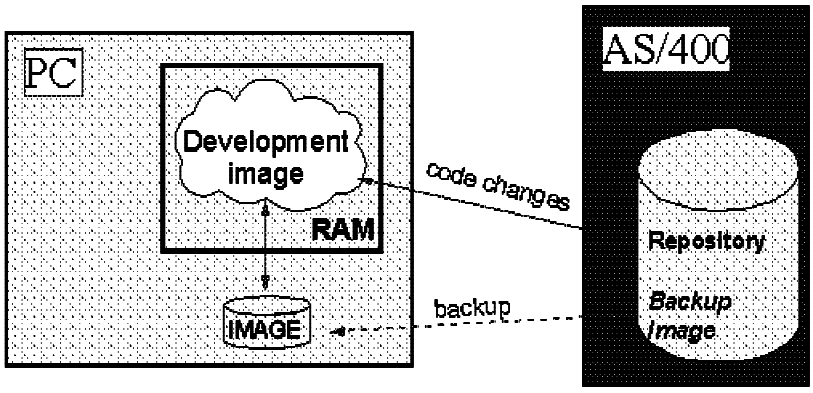
The configuration of VisualAge for Java places a development image/workspace on the client and a repository on the client/file server in the Professional Edition. In the Enterprise Edition, the repository has to be placed on a shared file server. The repository holds a copy of every version of every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



In a team development environment using the AS/400 IFS as the file server for the repository code, changes made by a developer to any component get 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.

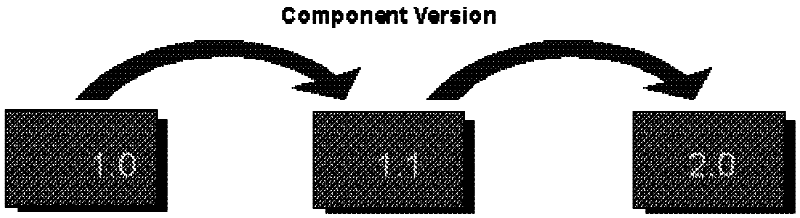
When a developer starts VisualAge for Java, the image/workspace gets copied from disk into memory and it is this copy of the image that the developer works with when adding/deleting/changing components. It is vital that the developer saves this "in-memory" image to disk on a regular basis (for example, 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 take an hour or two.

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



The team development facilities enable the versioning and editing 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. Therefore, in the following diagram, there are three separate versions of the component. The developer can assign each version a unique name and 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 and the developer explicitly versions these components. Methods are the exception and these get versioned automatically every time a change is made to them.

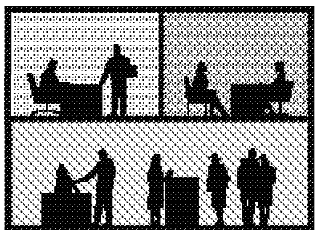


The big question is..."If a component is a version and a version is just another name for a frozen component that cannot be changed, then how do I change a versioned component"? The answer to this is to create an edition of the component. An edition of a component is editable, but the original version of the component remains in the repository should the developer need to go back to it at any time. Therefore, the process for creating, freezing, and changing a component (let's say Class A) is as follows;

- Create Class A (it gets created as an edition):
  - Write methods.
  - Define variables.
- Version Class A as Class A 1.0:
  - Class A is frozen and cannot be changed.
- Edition Class A:
  - Class A can now be edited again, but version 1.0 is still available should it need to be restored.
- Version Class A as Class A 2.0

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
—15/01/96 10:01:31
EDITIONING
—Making an editable entity from a version



2.3.5 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 having to boot 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), whereas 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 APIs.

VisualAge for Java has an applet creation SmartGuide that is accessed through its smarticon on the toolbar.

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), it is advisable to write this as a separate thread. Again, the SmartGuide provides the option of creating the applet to use its own thread.

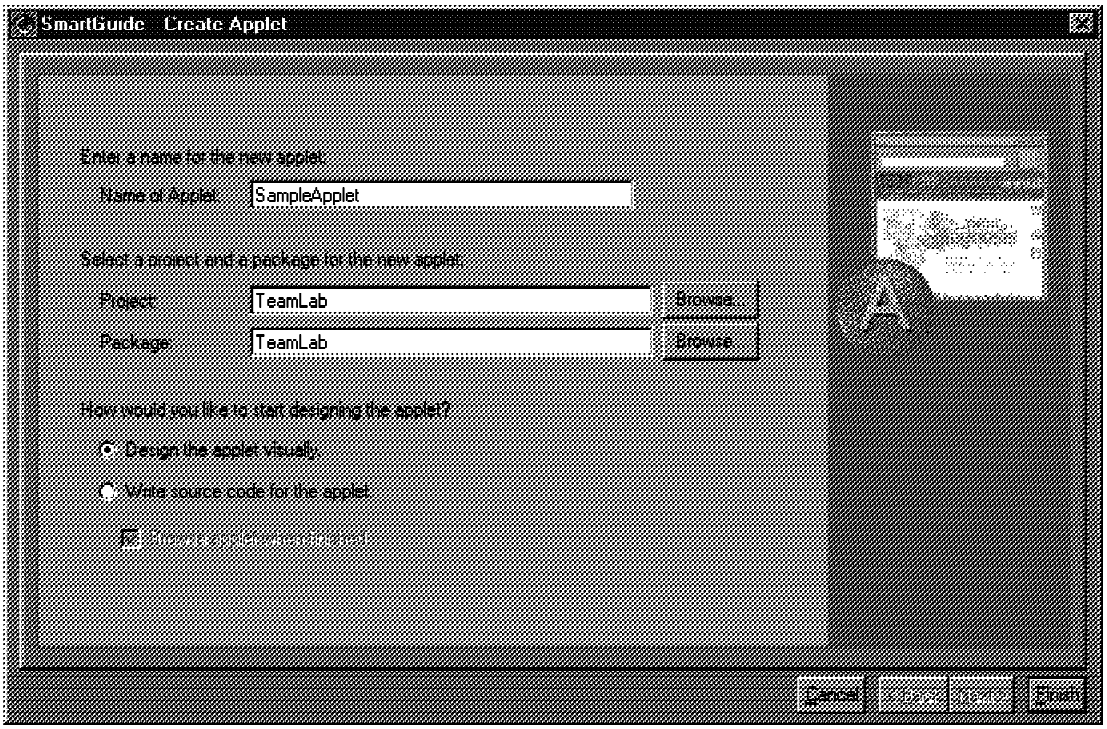


Figure 14. SmartGuide - Create Applet

After the applet has been created from the SmartGuide, use the class browser to view it and to see its place in the class hierarchy. As you expect, the applet inherits from java.applet.Applet and its required methods are generated also (init(), start(), stop(), destroy(), paint()). In the following example, you can see the destroy() method.

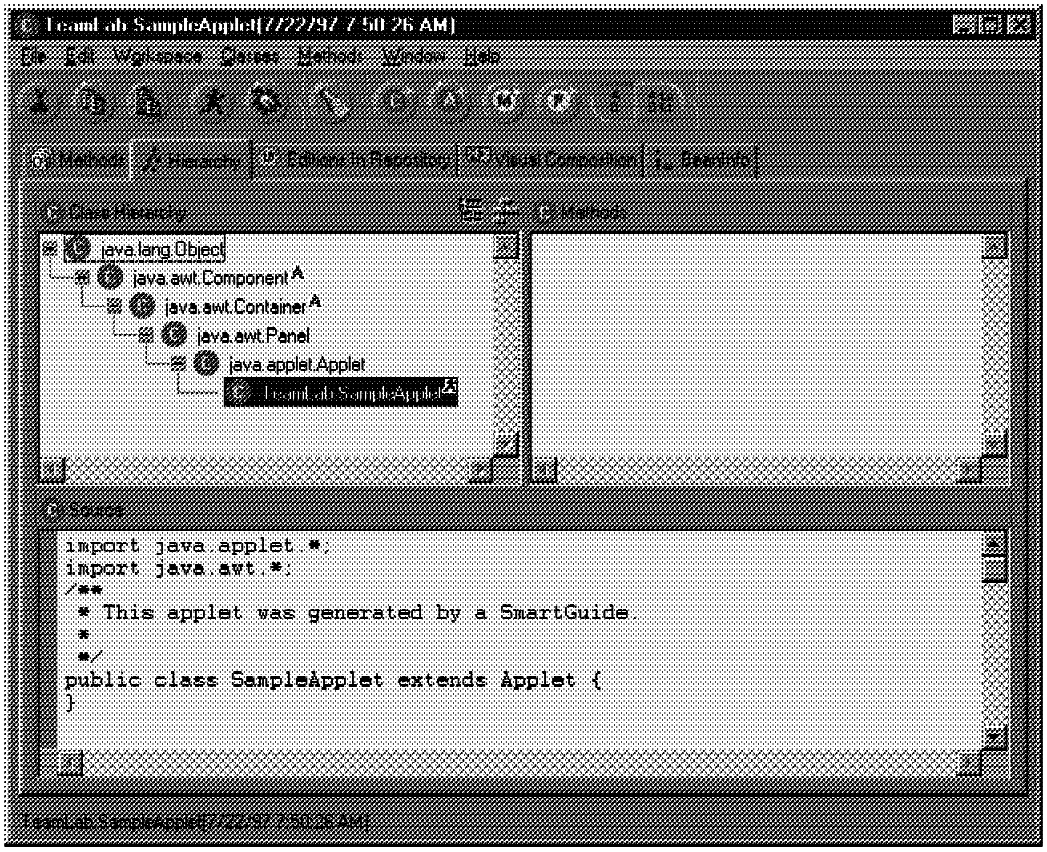


Figure 15. Sample Applet

From the applets pop-up menu, select the run--In Applet Viewer menu item to run the applet in the applet viewer. Outside of the IDE, an HTML file is required to wrapper 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. As part of the applet viewer, the Settings window is displayed that asks the developer to input these HTML settings prior to the applet running.

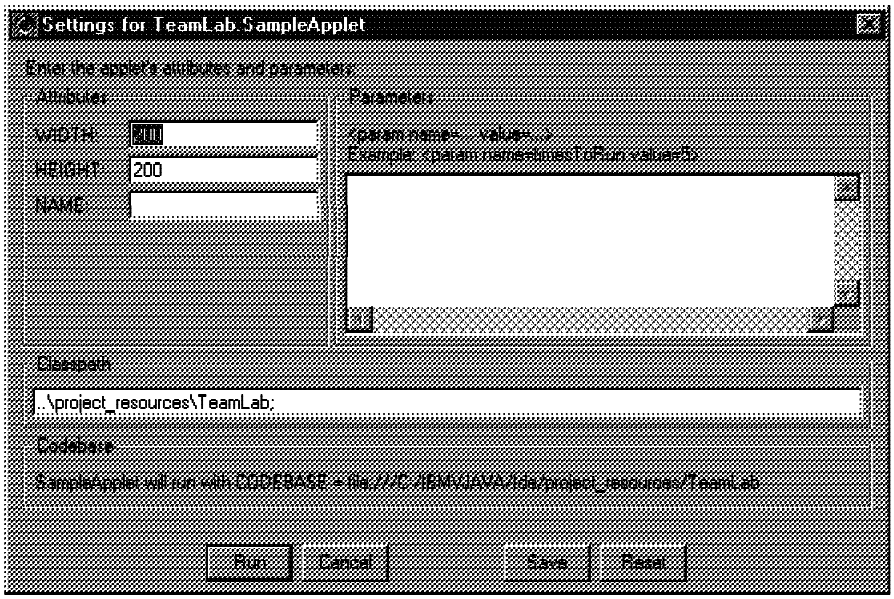


Figure 16. Run Applet

2.3.6 Editor/Debugger/SmartGuides

In an object-oriented application development environment, developers need to perform many similar tasks as procedural developers, but in addition, they perform a number of different tasks as part of a RAD 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).

- Add or change a method:

  Adding or changing a method is probably the most important task of an application developer as 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:

  Where ever 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, in any Method Source pane, the following code can be entered, 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, to hop over methods, to hop into methods, to run methods to completion, to interactively patch code, and to 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 recompile the entire class that contains the method being changed.

- Maintain project database:

  The team development environment has already been introduced in this chapter, and it is this team development environment that 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. If, for example, 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.

  In the following figure, the Workbench Window is displayed showing the ConnectToDB(java.lang.String, java.lang.String). In the method pane, a class can be defined/changed, a method can be defined/changed, a breakpoint can be set, code is syntax checked, and code snippets can be evaluated. In the following example, a breakpoint has been set on the as400.connectService(AS400.COMMAND); line, and this is indicated by the (blue) dot in the left margin.

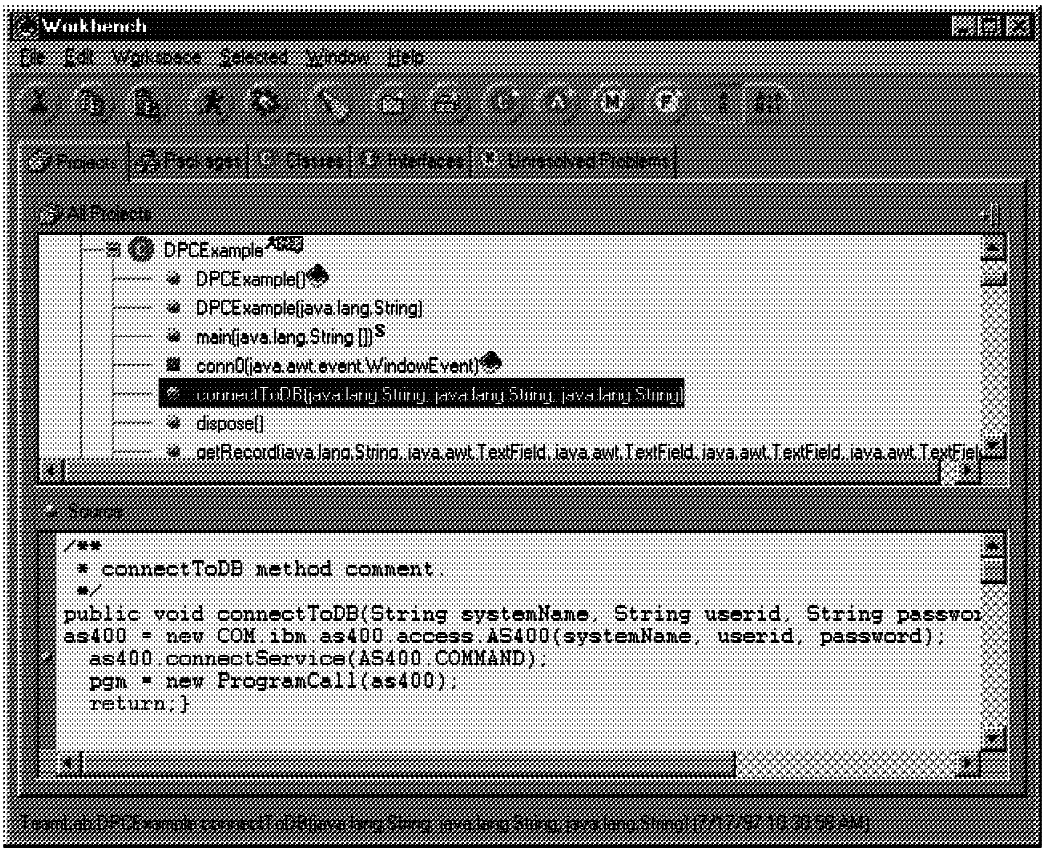


Figure 17. VisualAge for Java Workbench

2.3.6.1 The Editor Pane

The editing pane (elsewhere called the Method Source pane) allows the developer to;

- Perform editing operations.
- Undo/Redo:

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

  A developer can highlight some text and then select Search from the pop-up menu to search the workspace for references to or declarations of the highlighted text.
- 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 have been 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 or not.

- Editor setup:

  The editor has some default settings and these can be modified. The default settings are as follows:

  - Browser Font - Serif 10
  - Comment - Red
  - Default Text - Courier 10
  - Error - Red
  - Keyword - Blue
  - Literal String - Green

2.3.6.2 The Debugger

As you work in the integrated development environment, you need not launch a special debugger virtual machine or start the virtual machine in 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 button () on any tool bar.

You can use the debugger to step through code and inspect and change variables. As well, you can 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 pane (the threads pane) 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.

The middle part is divided into three panes that give more details of the current state of processing the code. From left to right, they are:

- Stackframes pane
- Variables pane:

  A text pane that displays the current value of a selected variable in the variables pane.
- Source pane

The stackframes pane (or thread stack pane) displays a stack trace as a list of stackframes. Each stackframe corresponds to a method that was called. Stackframes are in reverse chronological order (the most recent stackframe is the top item). The debugger lets you manipulate thread execution by dropping to a particular stackframe in the stackframes pane. 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.

The Source pane displays the source of the selected method.

The Variables pane displays a list of all the locally visible variables for the current stackframe. If you select a variable, its current value is displayed in the text/variable pane.

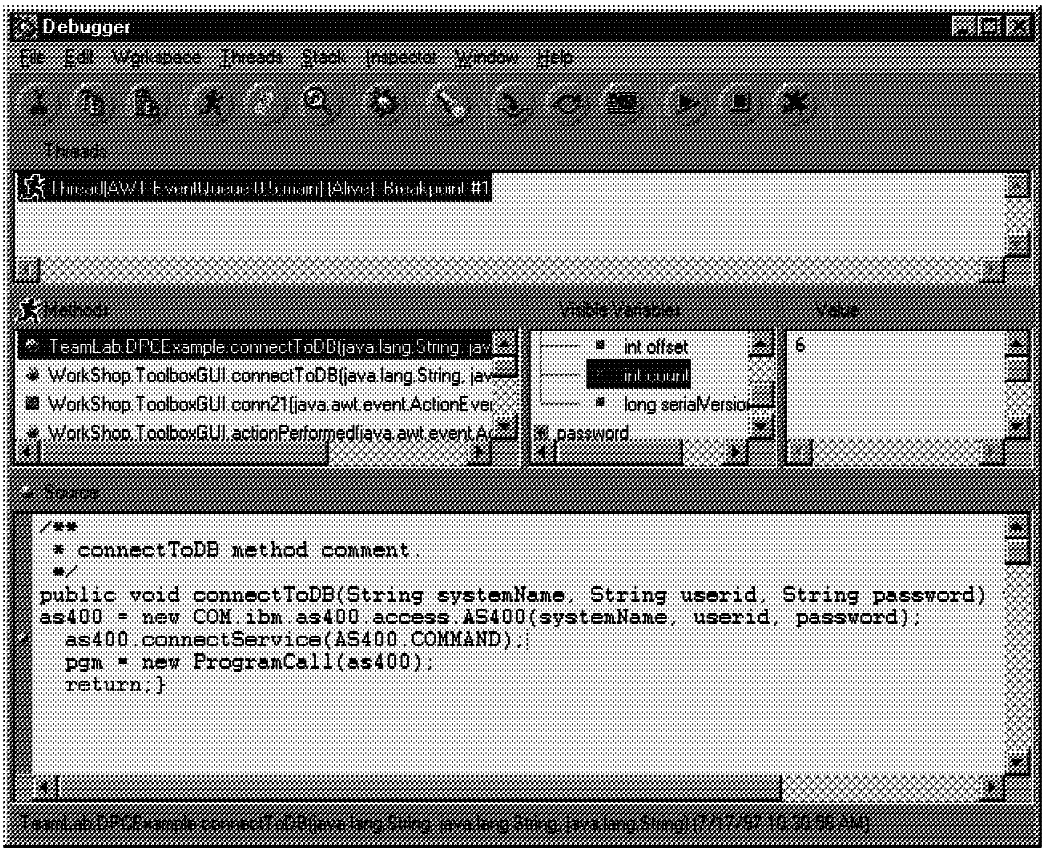


Figure 18. VisualAge for Java Debugger

Stepping through Methods: With the debugger's navigation buttons, 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 has hit a dead end. In this case, you must first drop the stackframe that throws the exception to reset the current status of processing.

- Into:

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

- Over:

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

- Return:

Executes all statements in the method that is currently selected in the stackframe's pane until the method is about to return and then stops. All local variables are updated.

- Resume:

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

2.3.6.3 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.

Using the Inspector: As an example, open an inspector on a string array object by copying the following code to a page in the scrapbook.

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

Select the code and select Inspect from the pop-up menu. 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 (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.

Changing the Value of an Object: You can change the value of fields while you are inspecting an object. Follow 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 you want in the field.
3. Select Save from the pop-up menu.

The expression in the Value pane is evaluated and 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.

2.3.6.4 Other VisualAge for Java Windows

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.

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

Hello World in Under a Minute:

1. Select Scrapbook from the Window pull-down menu.
2. Type:

   ```
   System.out.println("Hello World!");
      Select the line of code that you typed
      in Step 2.
   ```

3. Select Run from the pop-up menu.

The console (the standard output device) appears and displays the string Hello World!. The example works. The code was automatically compiled by the built-in Java compiler and then 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.

2.3.6.5 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::

- Is this a class or an interface?
- 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 Visual Composition Editor (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?

With future releases of VisualAge for Java, it is likely that more SmartGuides will be implemented, particularly in the area of AS/400 access to complement the AS/400 Toolbox classes.

There are a number of SmartGuides including class creation, interface creation, method creation, and applet creation.

2.3.7 Proxy Builder

The VisualAge for Java development environment includes a Java proxy builder that allows the development of JavaBeans to enable a local JavaBean to access another JavaBean located in another Java Virtual Machine (local or remote) by using a Java proxy object.

In VisualAge for Java, the RMI access builder can generate proxy code for a JavaBean in such a way that this JavaBean can be made accessible remotely through the builder-generated proxy code. A client-side server proxy, server-side server proxy, and supporting interface code are generated for each user JavaBean. A distributed client/server application can easily be created using these proxies. A client application can use the generated client-side server proxy as if it were a local object even though service requests to the client-side server proxy are actually sent over to the user JavaBean through RMI.

Note: In this release, you can only create servers out of JavaBeans that are generated by the C + + access builder. The tool to create distributed access for user-written JavaBeans is not yet available.

To enable a Java application to access a Java server over RMI:

1. Create or modify the packages.
2. Create the JavaBeans.
3. Import the JavaBeans into the Enterprise access builders.
4. Generate the distribution proxies as JavaBeans.
5. Export the generated JavaBeans.
6. Import the generated JavaBeans into the IDE.
7. Write the business logic.
8. Assemble the client.
9. Assemble the server.

10. Build the application.
11. Deploy the application.
12. Run the application.
13. Regenerate code.

2.4 Enterprise Access Builders (EAB)

The Enterprise Access Builders provides a graphical method to organize, create, and package parts generated by the following sub-components:

- Enterprise Access Builder for data (Data Access Builder)
- Enterprise Access Builder for CICS (CICS Access Builder)

These sub-components produce JavaBeans for access to transactions and databases.

The following operations are available from the Enterprise Access Builders:

- Create a package to organize parts into Java packages.
- Create data access parts to provide access to the Data Access Builder.
- Create CICS parts to access CICS transactions using CICS ECI.
- Create Jar file to package multiple parts into a JAR file.

In this chapter, we focus only 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 or 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 in.

- Quick and simple to use:

  You can simply specify a database table name and 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, please refer to Chapter 5, “Enterprise Access Builder For Data (DAX)” on page 135.

2.5 System Requirements

The current release of VisualAge for Java has the following system requirements:

- Processor:

  32-bit processor (Pentium or higher, or compatible processor)

- Display:

  SVGA 800x600 minimum (1024 x 768 recommended)

- Operating system:

  Windows NT 4.0, Windows 95, or OS/2 Warp 4.0

- Other software:

  – DAX with DB2 requires DB2 2.1.2.
  – Support for other databases through ODBC is also available. An ODBC driver is required and is not shipped with VisualAge for Java.
  – TCP/IP

- Memory:

  32M minimum (64M recommended)

- Disk space:

  – EAB = 55MB
  – EAB + IDE = 90MB
  – Swap space = 30MB

2.6 Summary

In summary, VisualAge for Java is a full member and newest member of the VisualAge family. It allows application developers to develop applications and Web-based applets 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, enabling 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 should be able to run without change on the native AS/400 Java Virtual Machine when it becomes available.

The IDE enables a developer to build and run applications, applets, and code snippets interactively without the need to run the compile statement (JavaC) from the command line. All applications can be 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 and an applet viewer. Because you can interactively modify code and run it without compilation, developers are able to debug code on the fly, spot errors in their code with the debugger, change it, and then continue without bringing the running application down...all within the VisualAge for Java IDE.

VisualAge for Java is an open IDE and developers can easily import and export Java source and class files as well as JavaBeans, which may have been purchased by the company or made available on the WWW. The JavaBeans support in VisualAge for Java also enable a developer to take an existing JavaBean (for example, from the WWW), import it into VisualAge for Java, modify the bean, and then export it again for use within another JDK 1.1 compliant development environment (for example, Symantic Cafe and Borlands 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, CICS transactions, and other programs. Secondly, 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 will run on OS/2 Warp Version 4.0, Windows NT 4.0, or Windows 95.

Chapter 3. AS/400 Toolbox for Java

This chapter covers the AS/400 Toolbox for Java. It covers 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
- DDM Record Level Access Example
- Distributed Program Call Example
- Data Queues Example
- Print Example
- IFS Example

3.1 Introduction to the AS/400 Toolbox for Java

The AS/400 Toolbox for Java is a set of enablers that supports an internet programming model. It provides familiar client/server programming interfaces for use by Java applets and applications. The toolbox does not require additional client support over and above what is provided by the Java Virtual Machine and JDK.

The AS/400 Toolbox for Java is currently available as an *open* beta driver from IBM. It can be downloaded from `http://www.as400.ibm.com/products/software/javatool/javatool.htm` .

The toolbox provides support similar to functions available when using the Client Access/400 APIs. It uses sockets 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 but 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 are provided:

- AS/400 object/infrastructure/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

The base API package requires a Java Virtual Machine supporting JDK 1.1. The class files for JDK 1.1, and the base API package must be available at runtime for applications and applets that are built using the base API package.

3.1.1 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 mapped drive.

3.1.2 AS/400 Host Servers

The AS/400 host servers must be running. Use the AS/400 command STRHOSTSVR *ALL to start the host servers.

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

3.1.3 AS/400 Object/Infrastructure/Sign-On

An AS400 object manages:

- 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 sockets 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 resource 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 userid 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 were 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 destructed. 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 userid 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 userid can be used. The default userid is used when a userid is not provided by the Java program. The default userid can be set either by the Java program or through the user interface. If the default userid has not been established, the sign-on dialog allows the user to set the default userid. Once the default userid is established for a given AS/400 system, the sign-on dialog does not allow the default userid 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

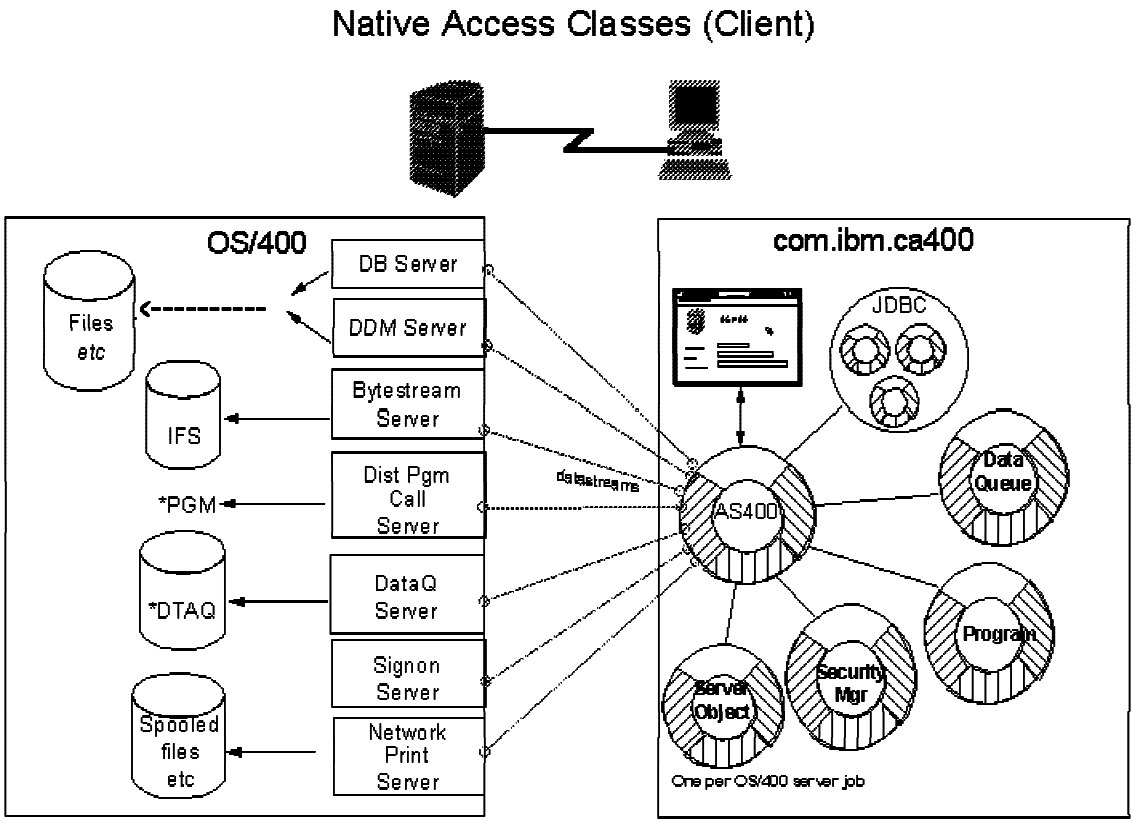


Figure 19. Java Host Server Overview

This set of interfaces provides the infrastructure needed to create and maintain sockets 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/400s and sign-on information for the system. These classes use the sign-on server and the central server to interact with the AS/400.

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 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 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

This set of classes 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. Each class implements the AS400DataType interface that defines common functions for converting data between representations.

Public Class	Description
AS400Bin2	Convert between a signed two-byte AS/400 number and a Java Integer object.
AS400Bin4	Convert between a signed four-byte AS/400 number and a Java Integer object.
AS400UnsignedBin2	Convert between an unsigned two-byte AS/400 number and a Java Integer object.
AS400UnsignedBin4	Convert between an unsigned four-byte AS/400 number and a Java Long object.
AS400Float4	Convert between a signed four-byte floating point AS/400 number and a Java Float object.
AS400Float8	Convert between a signed eight-byte floating point AS/400 number and a Java Double object.
AS400PackedDecimal	Convert between a byte packed decimal AS/400 number and a Java BigDecimal object.
AS400ZonedDecimal	Convert between a zoned decimal AS/400 number and a Java BigDecimal object.
AS400Text	Converts character data between an EBCDIC code page and character set (CCSID), and unicode.
AS400ByteArray	Convert between two byte arrays. This is useful because the converter correctly zero-fills and pads the target buffer.
AS400Array	Allows the Java program to work with an array of data types.
AS400Structure	Allows the Java program to work with a structure of data types.

3.2.3 Record Level Conversions

This set of classes represent AS/400 data to a Java program in a way similar to how it is defined on the AS/400 system. These classes provide a way to build aggregates of data.

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 the following diagram, the JDBC driver provided by the toolbox can be used instead of a JDBC/ODBC bridge driver.

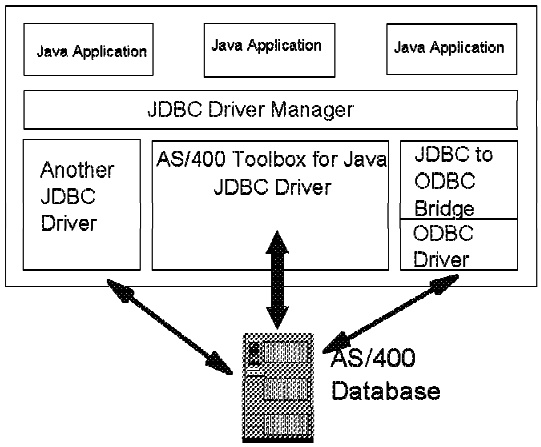


Figure 20. JDBC Interface to AS/400 System

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.

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 a 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 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. To use the host DDM server through a TCP/IP interface, some special PTFs are required. At the time this redbook was published, the required PTF numbers were SF42337 (V3R2) and SF42338 (V3R7). Check the AS/400 Toolbox for Java documentation for the latest PTF numbers and set up instructions.

The following public classes are defined and implemented:

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 (IFS)

The file system classes allow access to file objects that are in the AS/400 integrated file system. The original intent was to extend the file classes that are in the Java.io package but 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/write data from/to any specified location in the file. These classes use the Bytestream server to access the AS/400 system.

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

The 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 below. 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:

- SF42334 and SF42515 (V3R2)
- SF42316 and SF42516 (V3R7)

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 this support. A list of AS/400 messages that were generated when the command was run can be retrieved after the command completes. These classes use the Distributed Program Call server to access the AS/400 system.

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 this 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.

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 put 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 following public classes are defined and implemented:

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 Does the AS/400 System Fit 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 your 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 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 AS/400 objects or resources. User ID and password are prompted if one was not provided by the user. The user ID and password are verified on the AS/400 system and 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 userid that is passed when the connection is made. No authorities are adopted.

Note: At the initial release of the AS/400 Toolbox for Java, the AS/400 system has a SSL(secure sockets layer) interface,but the toolbox cannot take advantage of this encryption support until the OS/400 servers are enhanced to use secure sockets. Application developers must use caution when sending data through the network because there is no encryption of data. When SSL is available in the servers, the AS/400 Toolbox for Java is changed to take advantage of this support. Using a digital signature to secure the AS/400 Toolbox for Java code is also being investigated.

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 run time and 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 and 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/Restore Considerations

The Java classes and applets can reside in the IFS. There are no unique considerations for save and restore of these entities.

3.3.4 Install 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 run time. The JDK has set a precedent for how class files are accessed at run time. 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 (through the HTTP server), the class files are loaded from that AS/400 system. The applet can access only that 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 such 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 FFDC), severe errors, and other places a message can help the user recover from the error. Errors will only be logged if logging is turned on by the Java program.

3.4 Introduction to Application Examples

The remainder of this chapter covers application examples. It covers the following examples:

- AS/400 Database access:
  - JDBC
  - JDBC stored procedures
  - DDM Record Level Access
  - via Distributed Program Call
  - via 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 retrieval of a records from a PARTS file on the AS/400 system. Some examples allow an update to the current record in the window. All support the display of the entire PARTS file in a list box.

The PARTS file is defined as:

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 is 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 widget used in all the examples is from Taligent. The MultiColumnListbox class widget is available from Taligent's Web site, http://www.taligent.com. Version 1.0 is used.

Complete listings of all RPG and DDS source code can be found in Appendix B, “AS/400 Source Listings” on page 219. Listings for the Java source code can be found in Appendix C, “AS/400 Toolbox Example Java Code” on page 225.

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 is easy to use. Being an industry standard allows the AS/400 developer to use generic Java applets and objects and point them to the AS/400 system for data storage and retrieval. Additionally, the Java developer can use the 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 need not 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 precompiled SQL statements.
- CallableStatement - runs SQL stored procedures.
- ResultSet - provides access to a table of data generated by running a 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.

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

- Register the AS/400 JDBC driver.
- Connect to the database.
- Define and Prepare SQL statements.
- Execute SQL statements.
- Obtain and process results of the statements.
- Close the statements.
- 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*, SG24-4526-01.

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**; it caches 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 thus 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 on 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 utilized where possible. The properties control record blocking, package caching, and extended dynamic support. Selected properties and their settings are listed in the following table and other non-performance properties can be found in the Toolbox documentation.

Performance Properties				
Property	Description	Required	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, or else the row that is updated is not necessarily the current row.	No	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.	No	8, 16, 32, 64, 128, 256, 512	32
Prefetch	Specifies whether to prefetch data upon executing a SELECT statement. This increases performance when accessing the initial rows in the ResultSet.	No	True, 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 a 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`, then a package name must be set using the `package` property.	No	True, 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`.	No	SQL package	""
Package library	Specifies the library for the SQL package. This property has no effect unless the `extended dynamic` property is set to `true`.	No	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`.	No	True, False	False

Performance Properties				
Package clear	Specifies whether to clear SQL packages when they become full. Clearing a 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`.	No	True, 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`.	No	True, False	True
Package error	Specifies the action to take when SQL package errors occur. When a SQL package error occurs, the driver optionally throws a SQLException or posts 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`.	No	Exception, warning, none	Warning
Lazy close	Specifies whether to use lazy close support. Lazy close support provides a mechanism for deferring close requests until the next explicit communication to the server. This normally improves performance, but can cause concurrency problems in rare cases.	No	True, False	True

3.5.3 A JDBC Application Example

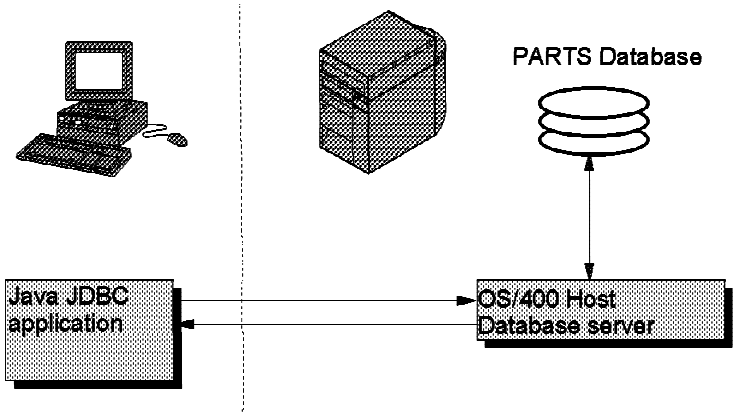


Figure 21. JDBC Application

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 a SQL result set. The JDBC support handles all data conversions.

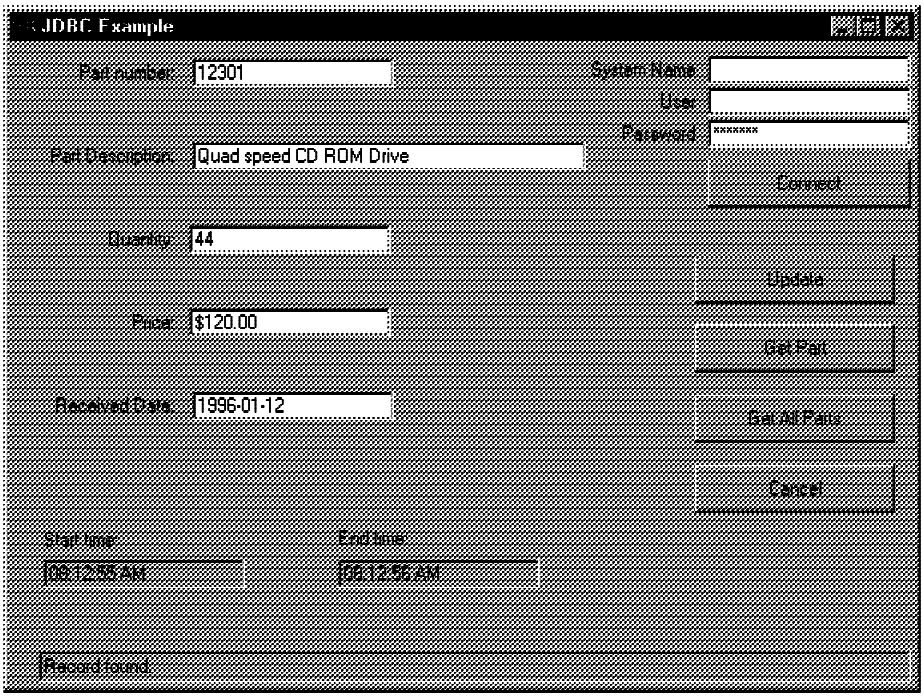


Figure 22. JDBC Application Get Part

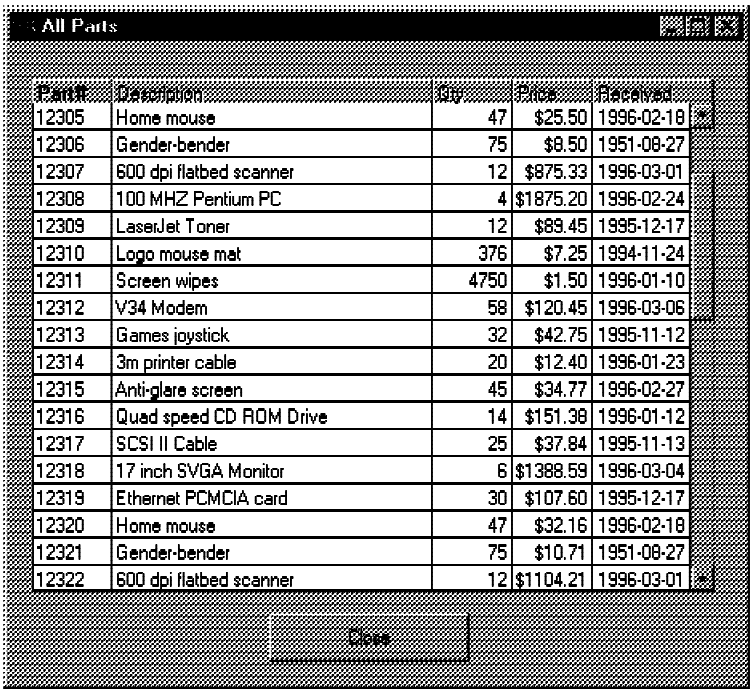


Figure 23. JDBC Application All Parts

A single part record can be retrieved and updated or all part records can be displayed in a list box. Two classes drive the application: JDBCExample and JDBCExampleDisplayAll.

The JBCDExample class creates the main application window, connects and disconnects the database, prepares the SQL statements, gets and displays a

single part record. 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 ConnectToDB Method

This 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.

```
/**
 * Connect to the AS/400 using JDBC
 */
public String connectToDB(String systemName, String userid, String password) {
  try {

    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",userid,
      password);

    psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE
      PARTNO = ?");
    psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS");

  } catch (Exception e) {showException(e); return "Connect Failed."; };


  return "Connected to AS/400.";
}
```

Class: JDBCExample Method: ConnectToDB

Let's dissect the method:

```
java.sql.DriverManager.registerDriver
         (new COM.ibm.as400.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",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, specifying use of ISO format for date fields, and error messages are to contain all available information.

```
psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE
                 PARTNO = ?");
psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS");
```

These statements create two 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.

3.5.4.2 GetRecord Method

This method is called when the **Get Part** button is pressed. A string parameter containing the part number is passed, along with the TextField objects that are used to display values of other fields in the part record.

```
/**
 * Retrieve a single record, using partNo as the key.
 * @param partNo java.lang.String
 * @param partDesc java.awt.TextField
 * @param partQty java.awt.TextField
 * @param partPrice java.awt.TextField
 * @param partDate java.awt.TextField
 */
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));
     rs = psSingleRecord.executeQuery();

    if (rs.next()) {
      partDesc.setText(rs.getString("PARTDS"));
      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("");
      partPrice.setText("");
      partDate.setText("");
      return "Record not found.";
    }
  } catch  (Exception e) {e.printStackTrace();showException(e); return null; }

  return "Record found.";
}
```

Class: JDBCExample Method: getRecord

Let's dissect the method:

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

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

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

Uses the PreparedStatment method, setInt to set the value of parameter 1 to the 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. Because this is the first 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"));
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:

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

3.5.4.3 UpdateRecord Method

This method is called when the **Update** 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.

```
public String updateRecord (String partno, String partdesc, String partqty,
  String partprice, String partdate) {

  // strip the leading dollar sign if it exists...
  String tempPrice = partprice.indexOf('$') == 0 ? partprice.substring(1) :
                     partprice;

  try {
    java.sql.Statement s = dbConnect.createStatement();
    String updatestring = "UPDATE PARTS SET PARTDS = '" + partdesc + "', PARTQY =
      " + partqty + ", PARTPR = " + tempPrice + ", PARTDT = '" + partdate +
      "' WHERE PARTNO = " + partno;
    s.executeUpdate(updatestring);
  } catch (Exception e) {showException(e); return "Update failed";}

  return "Record Updated.";
}
```

Class: JDBCExample Method: updateRecord

Let's dissect the method:

```
String tempPrice = partprice.indexOf('$') == 0 ? partprice.substring(1) :
   partprice;
```

Strips the "$" character off the price field if it exists in the first position of the String.

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

Creates a dynamic SQL statement object called s. We choose not to use a PreparedStatement for the update function. A statement object is used to show the use of an ad hoc SQL statement. Performance improves if the PreparedStatement is used, assuming the update occurs more than one time during the user session.

```
String updatestring = "UPDATE PARTS SET PARTDS = '" + partdesc + "',
PARTQY = " + partqty + ", PARTPR = " + tempPrice + ", PARTDT = '" + partdate
+ "' WHERE PARTNO = " + partno;
```

Builds 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.

```
s.executeUpdate(updatestring);
```

Runs the SQL update statement.

3.5.4.4 Dispose Method

This method is called when the application window is closed.

```
public void dispose () {
  try {
    psSingleRecord.close();
    psAllRecord.close();
    dbConnect.close();
    System.exit(0);
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  if (ivjDisplayAll != null) ivjDisplayAll.dispose();
  super.dispose();
return;
}
```

Class: JDBCExample Method: Dispose

Let's dissect the method:

```
psSingleRecord.close();
psAllRecord.close();
```

Releases the PreparedStatement's database and JDBC resources immediately. This also closes the current ResultSet.

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

Disconnects from the AS/400 system.

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

If the JDBCExampleDisplayAll class is instantiated, call its dispose method to shut it down.

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

Call the super class' dispose method to make sure any resources used by the Frame are properly freed.

3.5.4.5 JDBCExampleDisplayAll Class

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

3.5.4.6 Constructor Method

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

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

Class: JDBCExampleDisplayAll Constructor

Let's dissect the constructor:

```
this();
```

Execute the default constructor to take care of the window set up and initialization.

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

Sets the instance variables for the database Connection and PreparedStatement to reference the objects passed from JDBCExample.

```
this.setUpListBox();
```

Initializes column headings and widths for the listbox widget that is used to display the parts records.

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

Executes the database query and load the records to the list box. See the method details in the following example.

```
this.show(true);
```

Display the window.

3.5.4.7 PopulateListBox Method

This method is called from the constructor. It runs the SQL statement to select all records from the parts file.

```
public void populateListBox () {
  java.sql.ResultSet rs = null;
  String[] array = new String[5];

  try {
    rs = psAllRecord.executeQuery();
    while (rs.next()) {

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

      ivjMultiColumnListbox1.addRow(array);
    }
  } catch  (Exception e) {showException(e);}

  return;
}
```

Class: JDBCExampleDisplayAll Method: populateListBox

Let's dissect the method:

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

Executes the SQL defined by the psAllRecord PreparedStatement object, 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 **true**. If there are no records to retrieve, the method returns a false value. We will loop until next() returns a false value.

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

These lines retrieve 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.

```
ivjMultiColumnListbox1.addRow(array);
```

Adds the string values of the current parts record as a new row at the end of the list box.

3.5.5 Reusable GUI Part

For the remainder of the database examples in this chapter, a reusable class is used to handle the user interface. The advantage is that the user interface is designed, programmed, and tested once, and then 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 a part record, or adding part records to a list box.

To use the ToolboxGUI, we create a new class that is a sub-class of java.awt.Frame, and implements the PartsContainer interface. In the Visual Composition Editor of VisualAge Java, we perform an **Options ->Add Part** function and specify ToolboxGUI as the class name. The part is dropped on the empty application window and re-sized 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. Connect to the database server and return a String result.
- getRecord. Retrieve a single record from the database and place the resulting record field values in the TextFields passed.
- populateListBox. Retrieve all records from the database and add values for each record in the list box widget passed.
- updateRecord. Update the database record with the values passed; return 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 ToolboxGUI's 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 MultiColumnListbox. This class is instantiated when the **Get All Parts** button is pressed. It uses the same mechanism defined previously to call out to the parent's populateListBox method.

3.5.6 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/procedureName(?, ?, ?");
```

Defines 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);
```

Registers the third parameter (that is, the third question mark) as an output parameter for the stored procedure of SQL type integer. After the procedure is executed, the value of the parameter can be retrieved using aCS.getInt(3). Other *getters* 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);
```

Sets the value of the first parameter to an integer value of 500.

Stored procedures can be executed using the execute or executeQuery 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.

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

3.5.7 A JDBC Stored Procedure Application Example

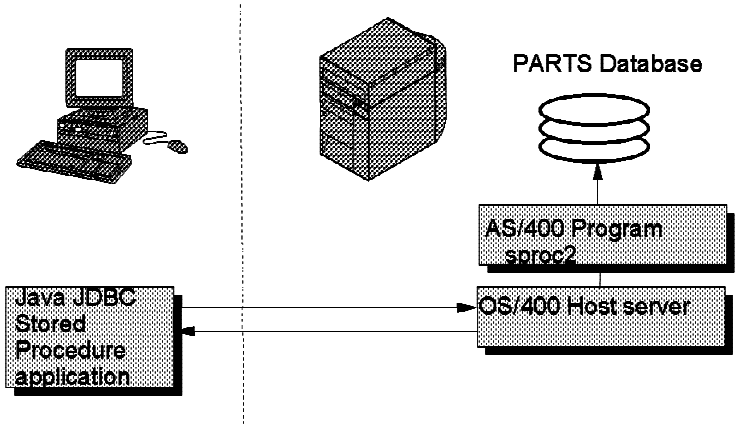
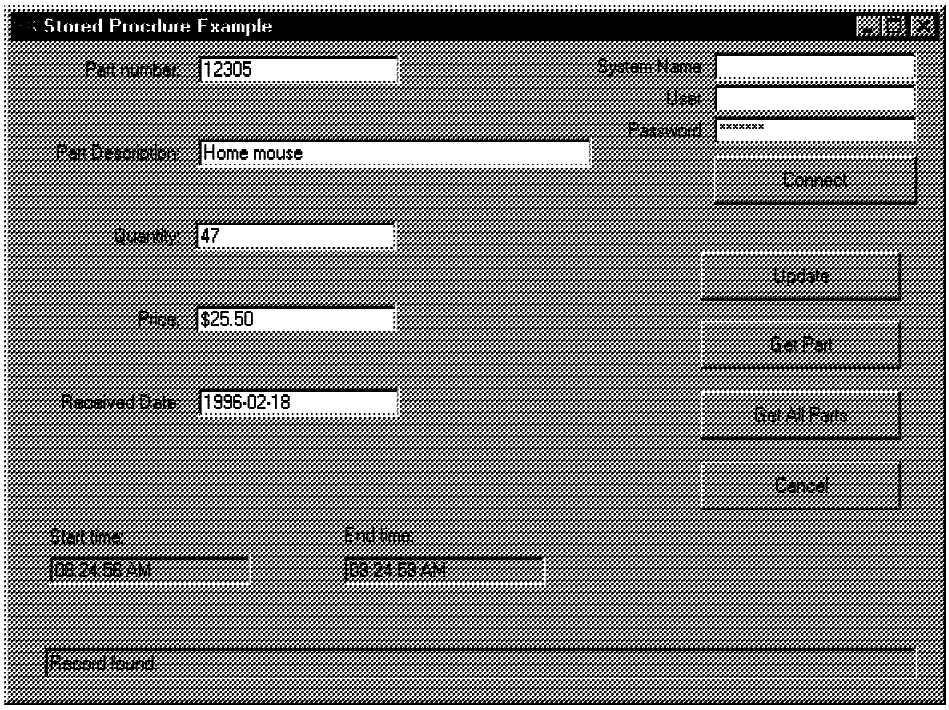
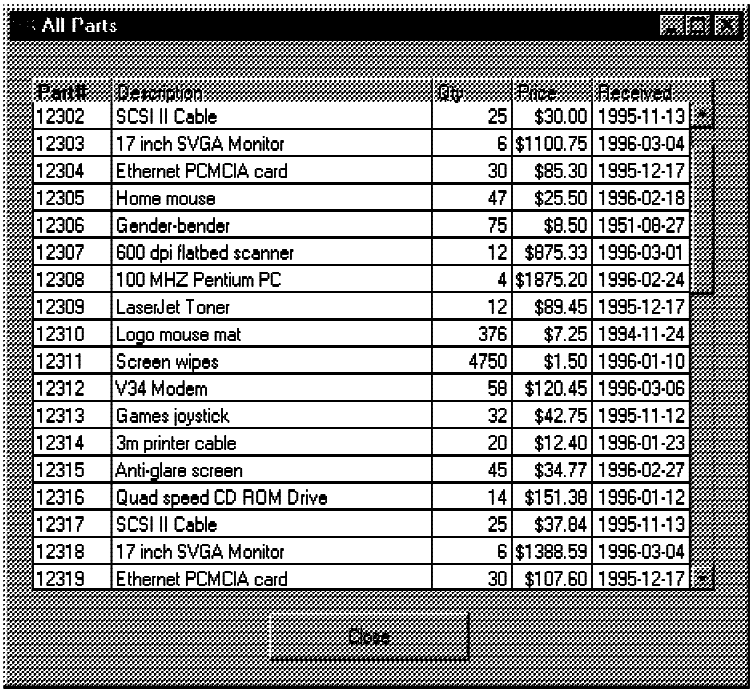


Figure 24. JDBC Application Stored Procedures

In this example, we use JDBC stored procedures to access records in an AS/400 database. 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.





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 update is not implemented in this example although it can be by using the same updateRecord method as the JDBC example or by creating a stored procedure that does the update.

The program we use as a stored procedure is written in RPG and named SPROC2 in library APILIB. The program 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.

3.5.8 StoredProcedureExample Class

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

3.5.8.1 ConnectToDB Method

This 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.

```
/**
 * Connect to the AS/400 using JDBC
 * @param systemName java.lang.String
 */
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",userid,password);
  try {dbConnect.createStatement().execute("drop procedure apilib.partqry2");
  } catch (Exception e) {};
  dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY2 (INOUT P1
    INTEGER, INOUT P2 INTEGER) EXTERNAL NAME APILIB.SPROC2 LANGUAGE RPG GENERAL");
  callableStmt = dbConnect.prepareCall("CALL APILIB.PARTQRY2(?, ?)");

  return;

}
```

Method highlights:

```
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",userid,password);
```

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

```
try {dbConnect.createStatement().execute("drop procedure apilib.partqry2");
  } catch (Exception e) {};
```

Attempts to remove the stored procedure 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
INTEGER, INOUT P2 INTEGER) EXTERNAL NAME APILIB.SPROC2 LANGUAGE RPG GENERAL");
```

Executes an SQL statement to add our procedure to the system catalog. A new statement object is created by the Connection object and the execute method is used to run an ad hoc SQL statement to declare the RPG program 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.

Important Information

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(?, ?)");
```

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.

3.5.8.2 GetRecord Method

This method is called by the ToolboxGUI class when the **Get Part** button is pressed.

```
/**
 * Retrieve a single record, using partNo as the key.
 * @param partNo java.lang.String
 * @param partDesc java.awt.TextField
 * @param partQty java.awt.TextField
 * @param partPrice java.awt.TextField
 * @param partDate java.awt.TextField
 */
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));
    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("");
    partPrice.setText("");
    partDate.setText("");
    return "Record not found.";
  }
  return "Record found.";}
```

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));
```

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 return `true:ex½g..` If there are no records to retrieve, the method returns a `false` value.

```
partDesc.setText(rs.getString(2));
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.

3.5.8.3 PopulateListBox Method

This method is called from the constructor. It runs the SQL statement to select all records from the parts file.

```
/**
 * This method was created by a SmartGuide.
 * @param aListBox com.taligent.widget.MultiColumnListbox
 */
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
  throws java.lang.Exception {

  java.sql.ResultSet rs = null;
  String[] array = new String[5];

  callableStmt.setInt(1, 2);
  callableStmt.setInt(2, 0);

  rs = callableStmt.executeQuery();
  while (rs.next()) {

    array[0] = rs.getString(1);
    array[1] = rs.getString(2);
    array[2] = Integer.toString(rs.getInt(3));
    array[3] = "$" + rs.getBigDecimal(4, 2).toString();
    array[4] = rs.getDate(5).toString();

    aListBox.addRow(array);
  }

  return;
}
```

Class: StoredProcedureExample Method: 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, 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 `true`. If there are no records to retrieve, the method returns a `false` value. We will loop until next() returns a false value.

```
array[0]  = rs.getString(1);
array[1] = rs.getString(2);
array[2] = Integer.toString(rs.getInt(3));
array[3] = "$" + rs.getBigDecimal(4, 2).toString();
array[4] = rs.getDate(5).toString();
```

These lines retrieve 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);
```

Adds the String values of the current parts record as a new row at the end of the list box.

3.5.8.4 Dispose Method

This method is called when the application window is closed.

```
public void dispose () {
  try {
    callableStmt.close();
    dbConnect.close();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  super.dispose();
  System.exit(0);
return;
}
```

Class: StoredProcedureExample Method: Dispose

Method highlights:

```
callableStmt.close();
```

Releases the CallableStatement's database and JDBC resources immediately. This also closes the current ResultSet.

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

Disconnects from the AS/400 system:

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

Call the super class dispose method to make sure any resources used by the frame are properly freed.

3.5.9 A DDM Record Level Access Application Example

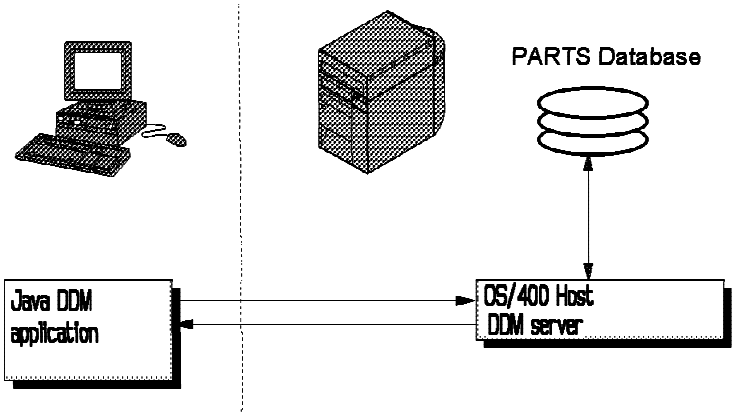
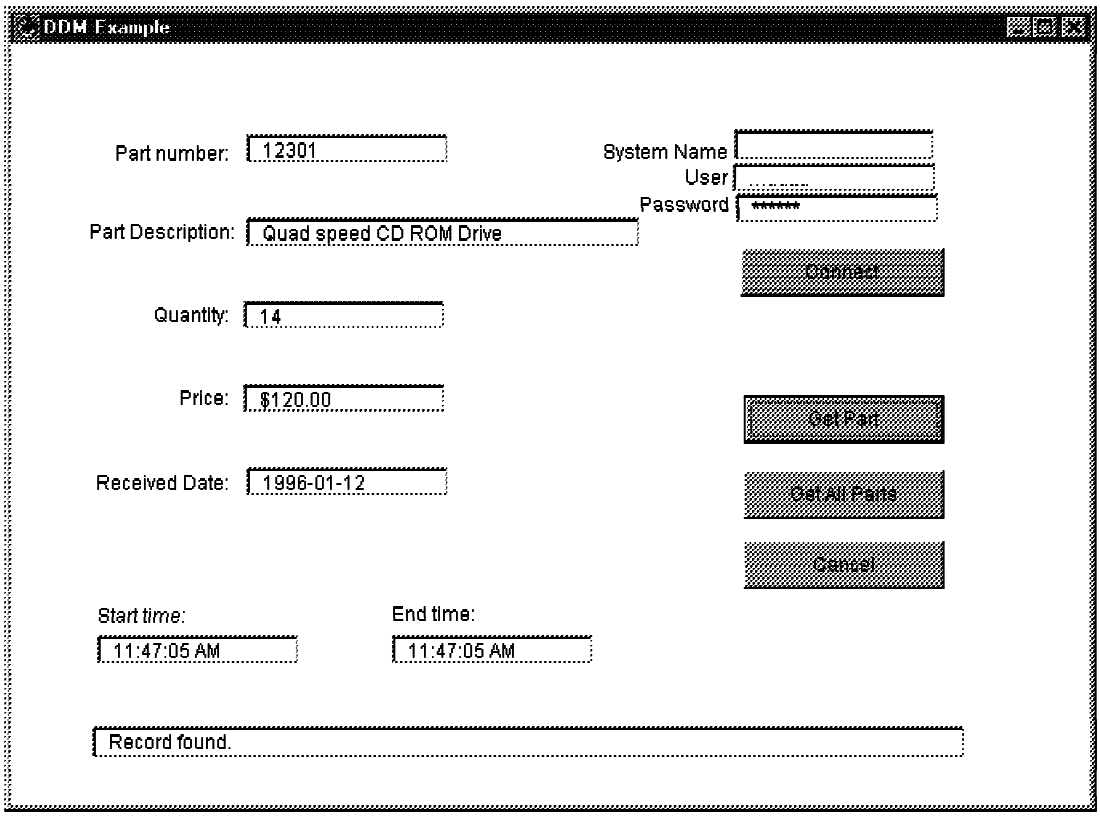
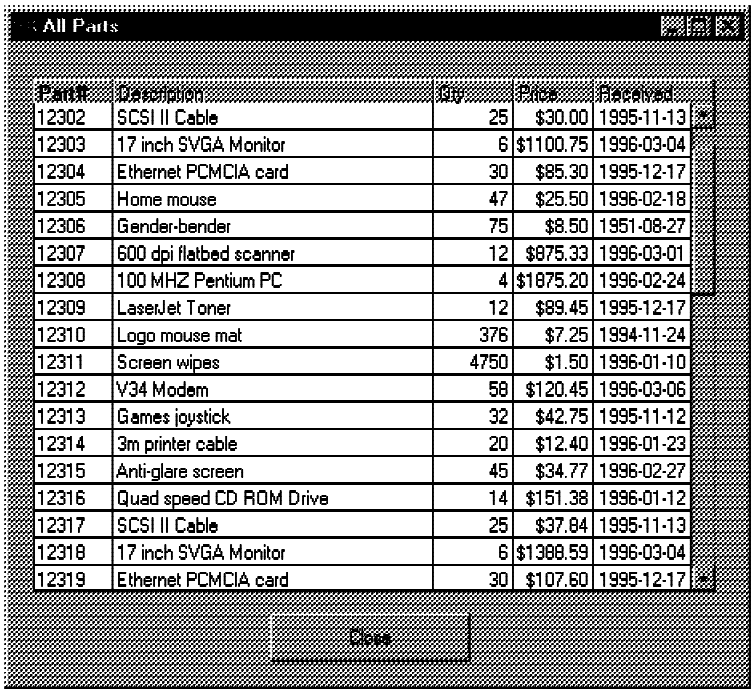


Figure 25. DDM Record Level Access

In this example, we use DDM Record Level Access to access records in an AS/400 database named Parts. 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 very easy to work with the file by field name.





Partit	Description	Qty	Price	Received
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
12315	Anti-glare screen	45	$34.77	1996-02-27
12316	Quad speed CD ROM Drive	14	$151.38	1996-01-12
12317	SCSI II Cable	25	$37.84	1995-11-13
12318	17 inch SVGA Monitor	6	$1388.59	1996-03-04
12319	Ethernet PCMCIA card	30	$107.60	1995-12-17

Class **RLExample** is the main class in this application. The ToolboxGUI class is used to handle all user interaction.

3.5.10 RLExample Class

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

3.5.10.1 Instance Variables

The following instance variables are declared for the class:

```
private AS400 as400;
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
private KeyedFile myKeyedFile;
```

3.5.10.2 ConnectToDB Method

This 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 COM.ibm.as400.access.AS400(systemName, userid, password);
 QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB",
               "PARTS",
               "*FILE",
               "MBR");
 myKeyedFile = new KeyedFile(as400, fileName.getPath());
 try{
    as400.connectService(AS400.RECORDACCESS);}
    catch(Exception e){
   System.out.println("Unable to connect");
   System.exit(0);
   }
  RecordFormat partsFormat = null;
  try
  {
   AS400FileRecordDescription recordDescription = new
   AS400FileRecordDesccription(as400, "/QSYS.LIB/APILIB.LIB/PARTS.FILE");
   partsFormat = recordDescription.retrieveRecordFormat()[0];
    // Indicate that PARTNO is a key field
    partsFormat.addKeyFieldDescription("PARTNO");
  }
  catch(Exception e)
  {
    System.out.println("Unable to retrieve record format");
    e.printStackTrace();
    System.exit(0);
  }
try{
 myKeyedFile.setRecordFormat(partsFormat);
        // Open the file.
 myKeyedFile.open(AS400File.READ_WRITE, 0,
     AS400File.COMMIT_LOCK_LEVEL_NONE);}
 catch(Exception e){
    System.out.println("Unable to open file");
    System.exit(0);
 }
  return;
  }
```

Method highlights:

```
as400 = new COM.ibm.as400.access.AS400(systemName, userid, password);
```

Creates a new AS400 connection object. System name, user ID, and password are passed through the constructor.

```
QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB",
              "PARTS",
              "*FILE",
              "MBR");
myKeyedFile = new KeyedFile(as400, fileName.getPath());
```

Creates a keyed file object that represents the file we will access on the AS/400 system. We use a QSYSObectPathName object to get the name of the file into the correct format.

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

Connect 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
 AS400FileRecordDesccription(as400, "/QSYS.LIB/APILIB.LIB/PARTS.FILE");
```

Create the Record Description object for accessing the file. The record description will be 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);
```

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);}
```

We open the file for both read and write. Since we are open for update, the blocking factor (0), doesn't matter. It is ignored and no blocking is done. If we are reading records only, we can specify a blocking factor on the open to help achieve better performance. We are not using commitment control.

3.5.10.3 GetRecord Method

This 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);

        // Read the first record matching theKey
  Record data = myKeyedFile.read(theKey);

        // If the record was not found, null is returned.
  if (data != null)
  {
  partDesc.setText((String)data.getField("PARTDS"));
  partQty.setText(((BigDecimal)data.getField("PARTQY")).toString());
  partPrice.setText("$" + ((BigDecimal)data.getField("PARTPR")).toString(
  partDate.setText((String)data.getField("PARTDT"));
  return "Record found.";

  }
else {
 partDesc.setText("");
   partQty.setText("");
   partPrice.setText("");
   partDate.setText("");
   return "Record not found.";
}
}
```

Method highlights:

```
Object[] theKey = new Object[1];
theKey[0] = new java.math.BigDecimal(partNo);
```

Create the key for reading the records. The key for a keyed file is specified as an object.

```
Record data = myKeyedFile.read(theKey);
```

Read the first record matching the key. Null is returned if the record is not found.

```
partDesc.setText((String)data.getField("PARTDS"));
partQty.setText(((BigDecimal)data.getField("PARTQY")).toString());
partPrice.setText("$" + ((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 populate the return values. This will cause the fields to be displayed on the screen.

3.5.10.4 PopulateListBox Method

This method is called from the constructor. It requests all records from the parts file.

```
public void populateListBox(com.taligent.widget.MultiColumnListbox
       aListBox) throws Exception {

String[] array = new String[5];
 try
   {

  // Display each record in the file

  Record record = myKeyedFile.readFirst();
  while (record != null)
  {
    array[0] =((BigDecimal)record.getField("PARTNO")).toString();
    array[1] =(String)record.getField("PARTDS");
    array[2] =((java.math.BigDecimal)record.getField("PARTQY")).toString();
    array[3]="$"+((BigDecimal)record.getField("PARTPR")).toString();
    array[4] =(String)record.getField("PARTDT");

    aListBox.addRow(array);
    record = myKeyedFile.readNext();

  }
 }
  catch(Exception e){
  System.out.println("unable to get all");
  System.exit(0);
 }
  return;}
```

Method highlights:

```
Record record = myKeyedFile.readFirst();
```

We use the readFirst method to read the first record.

```
array[0] =((BigDecimal)record.getField("PARTNO")).toString();
array[1] =(String)record.getField("PARTDS");
array[2] =((java.math.BigDecimal)record.getField("PARTQY")).toString();
array[3]="$"+((BigDecimal)record.getField("PARTPR")).toString();
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 listbox.

```
record = myKeyedFile.readNext();
```

We read the next record in the file.

3.5.10.5 Dispose Method

This method is called when the application window is closed.

```
public void dispose() {
 try {
 // All done with the file
   myKeyedFile.close();
 as400.disconnectAllServices();
 System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};

  super.dispose();
  System.exit(0);
return;
}
```

Class: RLExample Method: Dispose

Method highlights:

```
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();
```

Call the super class dispose method to make sure any resources used by the frame are properly freed.

3.5.11 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 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. 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/from the called program.
4. Use the Data Conversion classes to convert input parameter values from Java to 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 getOutputData method of the ProgramParameter object.
9. Convert output parameter values using the data conversion classes.

3.5.12 A Distributed Program Call (DPC) Application Example

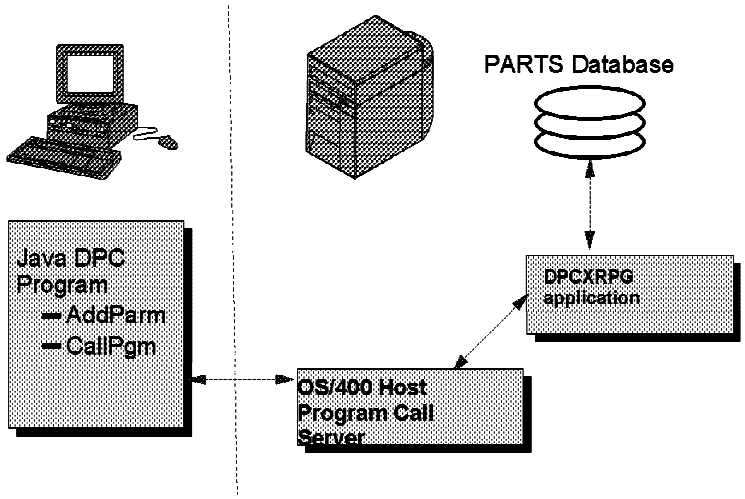
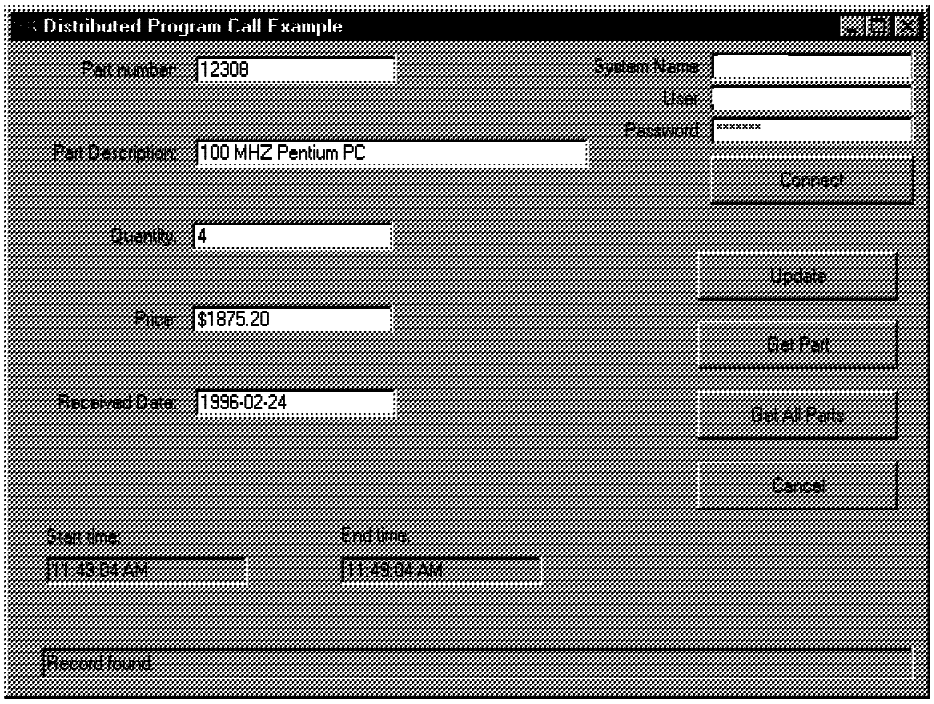
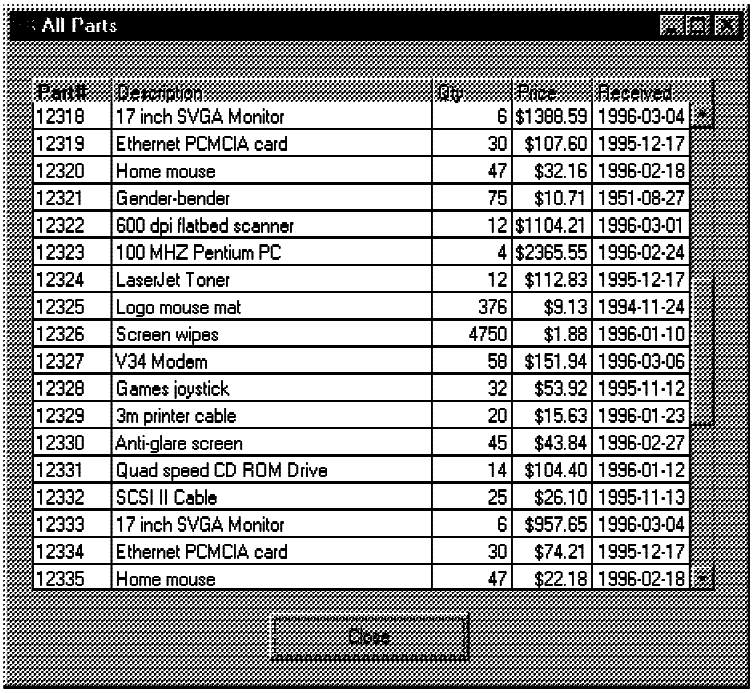


Figure 26. Distributed Program Call Example

In this example, we use the Distributed Program Call (DPC) interface to allow a client program to call an AS/400 program. The client program requests data from the AS/400 database by calling a AS/400 program. Information is passed between the programs using parameters. It is up to the application implementer to handle data conversions.





The client program requests data from the server program by calling it and passing it parameters. The input parameters are a flag and a part number. 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 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 class is used to handle all user interaction. A native RPG program is called on the AS/400 system to access and return data. Record update is not implemented in this example, although it can be by changing the RPG program to accept a changed record on the parameter list and updating the record in the database.

3.5.12.1 RPG Program Background

Library: APILIB

Program Name: DPCXRPG

Parameters:

Sequence	Description	Length Type	Input/Output	Values
1	Action flag	1 character	I/O	**Input** S - Retrieve single record A - Position to start of file F - Fetch next record **Output** Y - operation succeeded X - operation failed or EOF found
2	Part Number	5.0 packed	I/O	Part number to retrieve (input) or retrieved (output)
3	Description	25 character	O	
4	Quantity	5.0 packed	O	
5	Price	6.2 packed	O	
6	Ship Date	10 date	O	

3.5.13 DPCExample Class

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

3.5.13.1 Instance Variables

The following instance variables are declared for the class:

```
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
private AS400 as400;
private ProgramCall pgm;
private String progName = "/QSYS.LIB/APILIB.LIB/DPCXRPG.PGM";
```

3.5.13.2 ConnectToDB Method

This 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 COM.ibm.as400.access.AS400(systemName, userid, password);
  as400.connectService(AS400.COMMAND);
  pgm = new ProgramCall(as400);
  return;

}
```

Class: DPCExample Method: connectToDB

Method highlights:

```
as400 = new COM.ibm.as400.access.AS400(systemName, userid, password);
```

Creates a new AS400 connection object. System name, user ID, and password are passed through the constructor.

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

Connect 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 defined in the as400 object. The program and parameter information are supplied later.

3.5.13.3 GetRecord Method

This 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 {

// Setup the parameters
  ProgramParameter[] parmlist = new ProgramParameter[ .6];
// First parameter is to input action
  AS400Text asFlag = new AS400Text(1);
  parmlist[0] = new ProgramParameter( asFlag.toBytes("S") , 1 );
// Second parameter is to input PartNo
  AS400PackedDecimal asPartNo = new AS400PackedDecimal(5,0);
  parmlist[1] = new ProgramParameter( asPartNo.toBytes(new ja va.math.BigDecimal(partNo)) ,3);
// Third parm is output description
  parmlist[2] = new ProgramParameter(25);
// Fourth parm is quantity
  parmlist[3] = new ProgramParameter(3);
// Fifth parm is price
  parmlist[4] = new ProgramParameter(4);
// Sixth parm is date
  parmlist[5] = new ProgramParameter(10);

// Set the program name and parameter list
  pgm.setProgram( progName, parmlist );

// Run the program
  if (pgm.run() != true) {

    // Note that there was an error
    System.out.println( "program failed:" + progName );
    // Show the messages
    AS400Message[] messagelist = pgm.getMessageList();
    for (int i=0; i  messagelist.length; i++)  {
      // show each message
      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));
      partQty.setText(((java.math.BigDecimal)(
            new AS400PackedDecimal(5,0)).toObject(parmlist[3]
            .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));
    }
    else {
      partDesc.setText("");
      partQty.setText("");
      partPrice.setText("");
      partDate.setText("");
      return "Record not found.";
    }
  }

  return "Record found.";
}
```

Class: DPCExample Method: getRecord

Method highlights:

```
ProgramParameter[] parmlist = new ProgramParameter&l bracket.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.

```
// Second parameter is to input PartNo
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5,0);
parmlist[1]=new ProgramParameter(asPartNo.toBytes(
                new java.math.BigDecimal(partNo)),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.

```
// Third parm is output description
parmlist[2] = new ProgramParameter(25);
// Fourth parm is quantity
parmlist[3] = new ProgramParameter(3);
// Fifth parm is price
parmlist[4] = new ProgramParameter(4);
// Sixth parm is date
parmlist[5] = new ProgramParameter(10);
```

The next four paramters are output only and require that the number of output bytes be declared.

```
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. The method returns true if successful and false if a problem occurred.

```
AS400Message[] messagelist = pgm.getMessageList();
for (int i=0; i  messagelist.length; i++)  {
  // show each message
  System.out.println( messagelist[i] );
}
```

If an error ocurred 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 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));
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 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.

3.5.13.4 PopulateListBox Method

This method is called from the constructor. It runs the RPG program multiple times to retrieve all records from the PARTS file.

```
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
         throws java.lang.Exception {
  String[] array = new String[5];

// Setup the parameters
  ProgramParameter[] parmlist = new ProgramParameter[ .6];
// First parameter is to input action
  AS400Text asFlag = new AS400Text(1);
  parmlist[0] = new ProgramParameter( asFlag.toBytes("A") , 1 );
// Second parameter is to output PartNo
  parmlist[1] = new ProgramParameter(3);
// Third parm is output description
  parmlist[2] = new ProgramParameter(25);
// Fourth parm is quantity
  parmlist[3] = new ProgramParameter(3);
// Fifth parm is price
  parmlist[4] = new ProgramParameter(4);
// Sixth parm is date
  parmlist[5] = new ProgramParameter(10);

// Set the program name and parameter list
  pgm.setProgram( progName, parmlist );

  String flag = null;
```

Class: DPCExample Method: populateListBox

```
  if (pgm.run() != true) {
    // Note that there was an error
    System.out.println( "program failed:" + progName );
    // Show the messages
    AS400Message[] messagelist = pgm.getMessageList();
    for (int i=0; i  messagelist.length; i++)  {
      // show each message
      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) {

          // Note that there was an error
          System.out.println( "program failed:" + progName );
          // Show the messages
          AS400Message[] messagelist = pgm.getMessageList();
          for (int i=0; i  messagelist.length; i++)  {
          // show each message
          System.out.println( messagelist[i] );
          }
        return;
        }
        else {
          flag = (String)(asFlag.toObject(parmlist[0].getOutputData(),0));

          if (flag.equals("Y")) {
            array[0] =(((java.math.BigDecimal)(new
AS400PackedDecimal(5,0)).toObject(parmlist[1].
getOutputData(),0)).toString());
            array[1] =(String)(new AS400Text(25)).toObject(pa rmlist[2].getOutputData(),0);
            array[2] =((java.math.BigDecimal)(new
AS400PackedDecimal(5,0)).toObject(parmlist[3].g
etOutputData(),0)).toString();
            array[3] = "$" + ((java.math.BigDecimal)(new AS40 0PackedDecimal(6,2))
.toObject(parmlist[4]getOutputData(),0)).toString();
            array[4] =(String)(new AS400Text(10)).toObject(pa rmlist[5].getOutputData(),0);

            aListBox.addRow(array);
          }
        }
      } while (flag.equals("Y"));
    }
  }

  return;
}
```

Class: DPCExample Method: populateListBox

Method highlights:

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.

```
pgm.setProgram( progName, parmlist );
```

Associate the program name and parameter list with the ProgramCall object.

```
if (pgm.run() != true) {
```

Call 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)) {
```

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 fetch 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., meaning 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 and place their Java String converted value into a string array for addition to the list box.

3.5.13.5 Dispose Method

This method is called when the application window is closed.

```
public void dispose () {
  try {
    as400.disconnectAllServices();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  super.dispose();
  System.exit(0);
return;
}
```

Class: DPCExample Method: dispose

Method highlights:

```
as400.disconnectAllServices();
```

Releases all connections to the AS/400 system and releases resources associated with server jobs processing requests for the client.

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

Call the super class dispose method to make sure any resources used by the Frame are properly freed.

3.5.14 Data Queues

The Data Queue classes allow a Java program to create, delete, write, and read data queues on the AS/400 system.

The 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. For 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.

3.5.14.1 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 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.

3.5.14.2 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.

3.5.15 A Data Queue Application Example

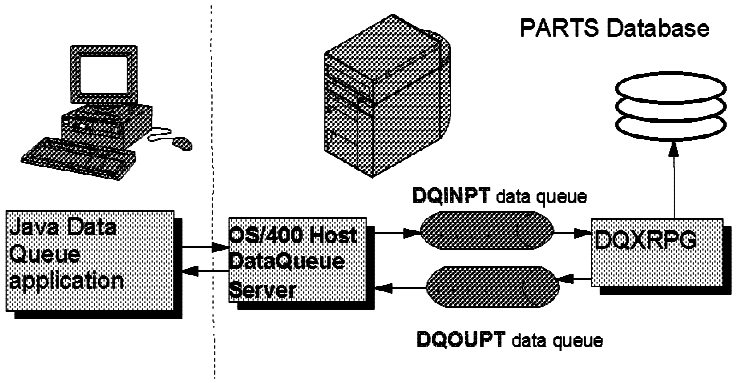
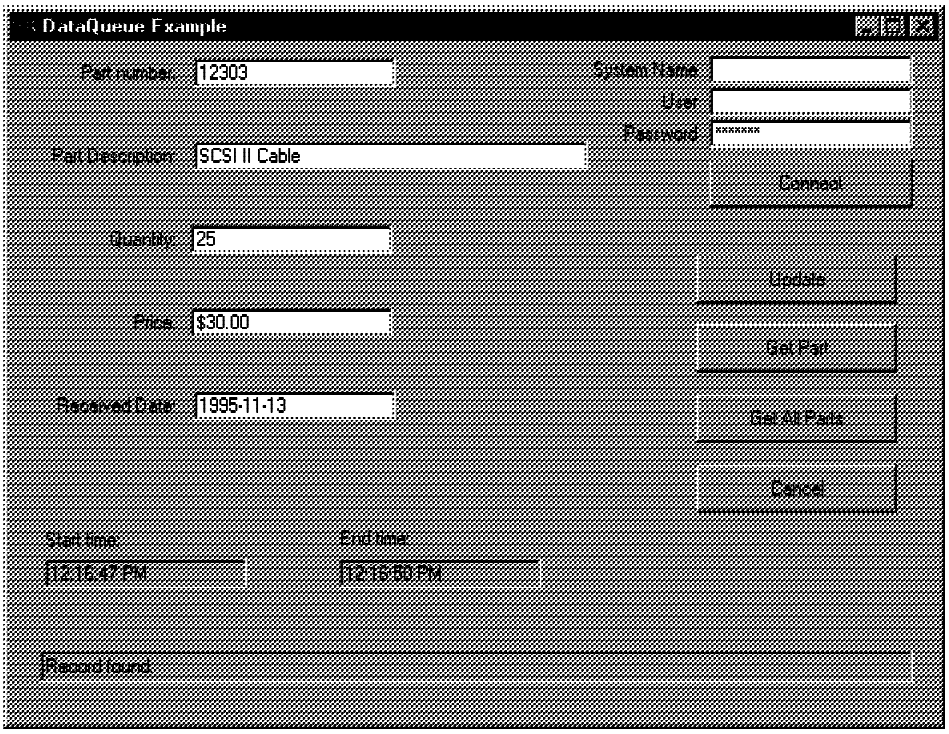
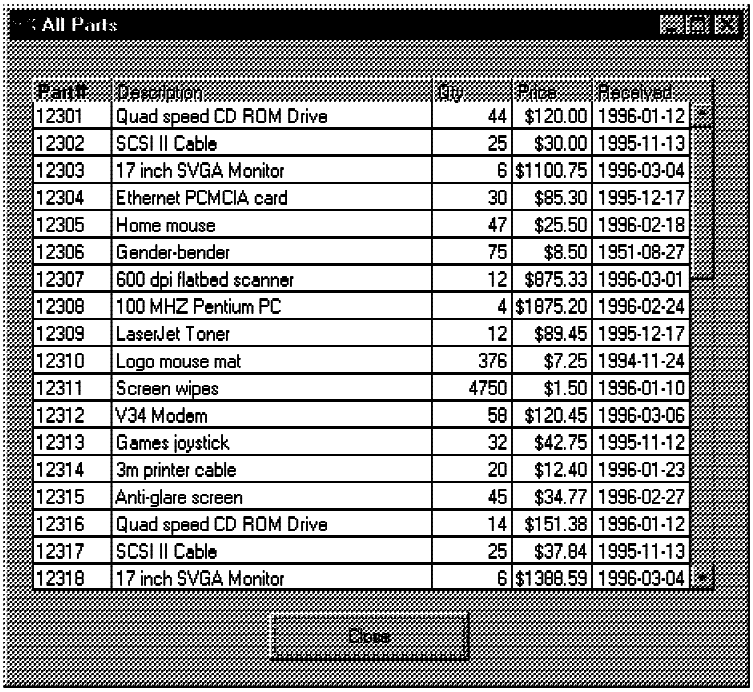


Figure 27. Data Queue Application

In this example we use, the Data Queue interface to allow a client program to interface with an AS/400 program. 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#	Description	Qty	Price	Received
12301	Quad speed CD ROM Drive	44	$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
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
12315	Anti-glare screen	45	$34.77	1996-02-27
12316	Quad speed CD ROM Drive	14	$151.38	1996-01-12
12317	SCSI II Cable	25	$37.84	1995-11-13
12318	17 inch SVGA Monitor	6	$1398.59	1996-03-04

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 class is 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 update is not implemented in this example although it can be by

changing the RPG program to accept a changed record on the input data queue and update the record in the database.

3.5.15.1 Data Queue Server Program Background

An input queue and output queue were created with the commands:

```
CRTDTAQ DTAQ(APILIB/DQINPT) MAXLEN(6) TEXT('Data Queue for Parts Input')
CRTDTAQ DTAQ(APILIB/DQOUPT) MAXLEN(48) TEXT('Data Queue for Parts Output')
```

Program Name: APILIB/DQXRPG

Data queue, DQINPT layout:

Queue Position	Description	Length Type	Values
1 - 1	Action flag	1 character	S - Retrieve single record A - Position to start of file
2 - 6	Part Number	5.0 zoned	Part number to retrieve

Data queue, DQOUPT layout:

Queue Position	Description	Length Type	Values
1 - 1	Action flag	1 character	Y - operation succeeded X - record not found or EOF
2 - 6	Part Number	5.0 zoned	Part number to retrieved
7 - 31	Description	25 character	
32 - 34	Quantity	5.0 packed	
35 - 38	Price	6.2 packed	
39 - 48	Ship Date	10 character	

3.5.16 DataQueueExample Class

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

3.5.16.1 Instance Variables

The following instance variables are declared for the class:

```
private  COM.ibm.as400.access.AS400 as400;
private  COM.ibm.as400.access.DataQueue dqInput;
private  COM.ibm.as400.access.DataQueue dqOutput;
private  COM.ibm.as400.access.RecordFormat rfInput;
private  COM.ibm.as400.access.RecordFormat rfOutput;
```

3.5.16.2 ConnectToDB Method

This 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.

```
/**
 * Connect to the AS/400 using JDBC
 * @param systemName java.lang.String
 */
public void connectToDB(String systemName, String userid, String password) throws Exception {
  as400 = new COM.ibm.as400.access.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");

  return;
}
```

Class: DataQueueExample Method: connectToDB

Method highlights:

```
as400 = new COM.ibm.as400.access.AS400(systemName, userid, password);
```

Creates a new AS400 connection object. System name, user ID, and password are passed through the constructor.

Note: If you import the COM.ibm.as400.access classes, you don't have to fully qualify the class name. You could write it as: **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");
```

Creates new DataQueue objects for the input and output queues. The fully-qualified IFS name of the data queues are passed in the constructor.

3.5.16.3 GetRecord Method

This 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();
// set values of the input record ...
  Record rInput = rfInput.getNewRecord();
  rInput.setField("flag","S");
  rInput.setField("partno",new java.math.BigDecimal(partNo));

  dqInput.write(rInput.getContents());
  DataQueueEntry dqe = dqOutput.read();

  Record rOutput = rfOutput.getNewRecord(dqe.getByteData());

  if (((String)rOutput.getField("flag")).equals("Y")) {
    partDesc.setText((String)rOutput.getField("partds"));
    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("");
    partPrice.setText("");
    partDate.setText("");
    return "Record not found.";
  }
  return "Record found.";
}
```

Class: DataQueueExample Method: getRecord

Method highlights:

```
if (rfInput == null) intRecordFormat();
```

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));
```

Set 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.

```
dqInput.write(rInput.getContents());
```

Write the input record to the input data queue. The getContents method returns a byte array of the value of the record in AS/400 format.

```
DataQueueEntry dqe = dqOutput.read();
```

Read the next entry off the output data queue. This returns a data queue entry object.

```
Record rOutput = rfOutput.getNewRecord(dqe.getByteData());
```

Creates a new output record, using the output record format and setting field values using the array of bytes returned by the getByte 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"));
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.

3.5.16.4 InitRecordFormat Method

This 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");

// set up the input record format....
  rfInput = new RecordFormat();
  rfInput.addFieldDescription(asFlag);
  rfInput.addFieldDescription(asPartNo);
// set up the output record format....
  rfOutput = new RecordFormat();
  rfOutput.addFieldDescription(asFlag);
  rfOutput.addFieldDescription(asPartNo);
  rfOutput.addFieldDescription(asPartDS);
  rfOutput.addFieldDescription(asPartQy);
  rfOutput.addFieldDescription(asPartPR);
  rfOutput.addFieldDescription(asPartDt);

  return;
}
```

Class: DataQueueExample Method: initRecordFormat

Method highlights:

```
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);
```

Create 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);
```

Create the output record format by adding field descriptions to a new RecordFormat object.

3.5.16.5 PopulateListBox Method

This 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 the parts records have been returned by the server program.

```
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
                                  throws java.lang.Exception {
  String[] array = new String[5];

  if (rfInput == null) initRecordFormat();

// set values of the input record ...
  Record rInput = rfInput.getNewRecord();
  rInput.setField("flag","A");

  dqInput.write(rInput.getContents());
  String flag = null;
  do {

    Record rOutput = rfOutput.getNewRecord(dqOutput.read().getByteData());
    flag = (String)rOutput.getField("flag");

    if (flag.equals("Y")) {
      array[0] =((java.math.BigDecimal)rOutput.getField("part no")).toString();
      array[1] =(String)rOutput.getField("partds");
      array[2] =((java.math.BigDecimal)rOutput.getField("part qy")).toString();
      array[3] = "$" + ((java.math.BigDecimal)rOutput.getFiel d("partpr")).toString();
      array[4] =(String)rOutput.getField("partdt");

      aListBox.addRow(array);
    }
  } while (flag.equals("Y"));

  return;
}
```

Class: DataQueueExample Method: populateListBox

Method highlights:

```
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());
```

Put the current value of the input record format on the input data queue.

```
Record rOutput = rfOutput.getNewRecord(dqOutput.read().getByteData());
```

Read the next entry from the output data queue and use the array of bytes returned 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", meaning 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.

3.5.16.6 Dispose Method

This method is called when the application window is closed.

```
public void dispose () {
  try {
    as400.disconnectAllServices();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  super.dispose();
  System.exit(0);
return;
}
```

Class: DataQueueExample Method: dispose

Method highlights:

```
as400.disconnectAllServices();
```

Releases all connections to the AS/400 system and releases resources associated with server jobs processing requests for the client.

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

Call the super class dispose method to make sure any resources used by the frame are properly freed.

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:

- Establish a connection.
- Create a spooled file list.
- Set the user filter.
- Open the spooled file list.
- Retrieve entries.
- Close the spooled file list.
- Close the connection.

3.6.1 A Print Example

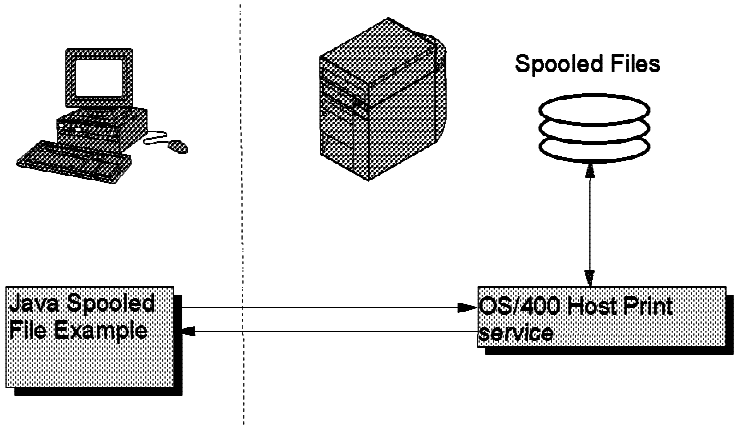


Figure 28. AS/400 Toolbox for Java Print

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. Spooled files can be created, deleted, held, and released. Spooled files can also be filtered by user name.

Toolbox classes used:

- AS400(String, String, String):

  Constructor for class COM.ibm.as400.access.AS400. Constructs an AS/400 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.

SpooledFileList methods used:

- openSynchronously():

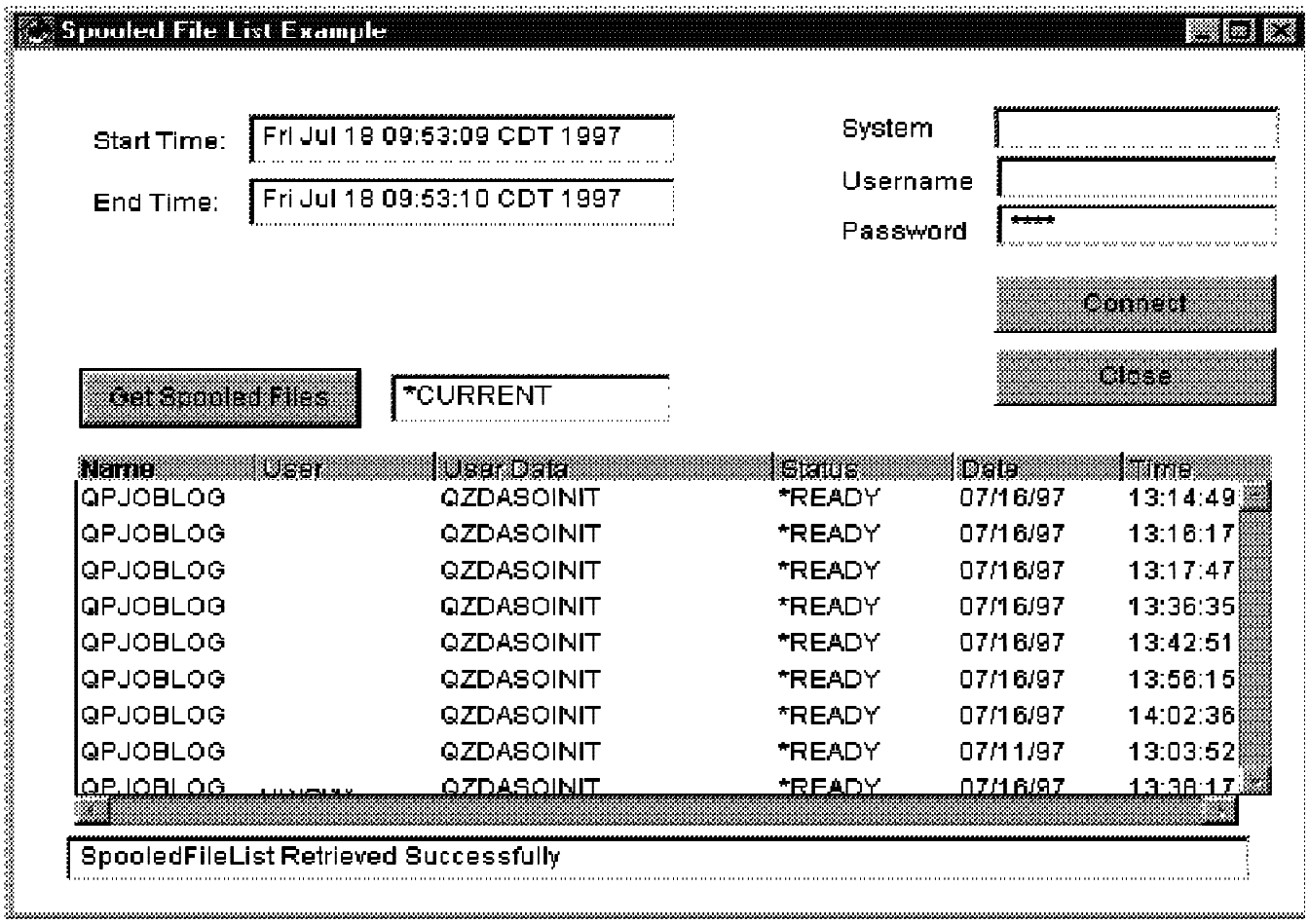
  Builds the list synchronously. This method does not return until the list has been built completely.

- getObject(int index):

  Returns one object from the list.

- close():

  Closes the spooled file list.



This application was built 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. We use the Taligent multi-column list box to display the spooled files. The Taligent MultiColumnListbox is downloadable from their Web site. This window 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 Connect Method

This 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)
  {  try{
   as400 = new AS400(systemName, userid, password);


   as400.connectService(AS400.PRINT);


   } catch   (Exception e) { return "Exception " + e; }
   return "Connected";
}
}
```

Method highlights:

`as400 = new AS400(systemName, userid, password);`

Creates a new AS400 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 the user requests a spooled file.

3.6.2.2 FormatSpooledFile Method

```
public String []
        formatSpooledFile(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;
 }
}
}
```

Method highlights:

This 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.

3.6.2.3 GetSpooledFilesForUser (String User)

```
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);
              getMultiColumnListbox1().addRow( formatSpooledFile(currentFile),
                   currentFile);
            }
     } catch (Exception e) {return "An exception occurred" + e;}
    list.close();
    getMultiColumnListbox1().repaint();
    return "SpooledFileList Retrieved Successfully";
 }
}
```

Method highlights:

- 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 the spooled file must have in order 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 has been 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.

3.7 Integrated File Systems Access

The Integrated File System support allows access to files in the AS/400 system's Integrated File Systems as a stream of bytes. It provides 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 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 were running on a Windows 95 or Windows NT personal computer, the network drives function of AS/400 Client Acces 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 An IFS Example

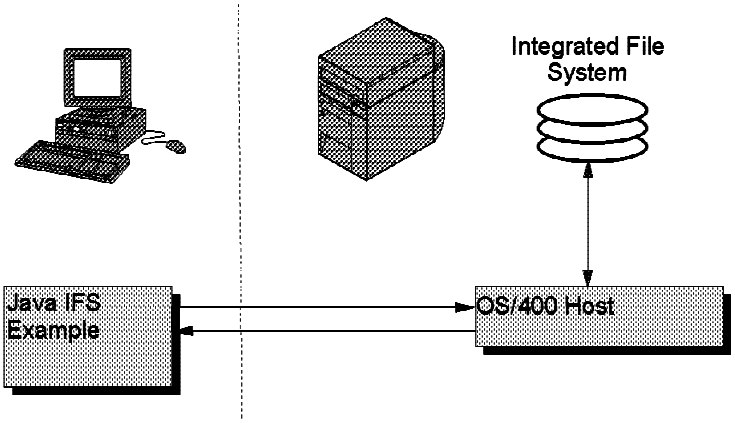


Figure 29. AS/400 Toolbox for Java IFS

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.

Toolbox classes used:

- AS400(String, String, String):

  Constructor for class COM.ibm.as400.access.AS400. Constructs an AS/400 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 AS400 system.

- IFSFileInputStream(AS400, IFSFile, int):

  Constructor for class COM.ibm.as400.access.IFSFileInputStream. Constructs an input stream to read contents of the file.

IFS methods used:

- 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 this IFSFile was created from.

- 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 the file was modified (as a long).

- available():

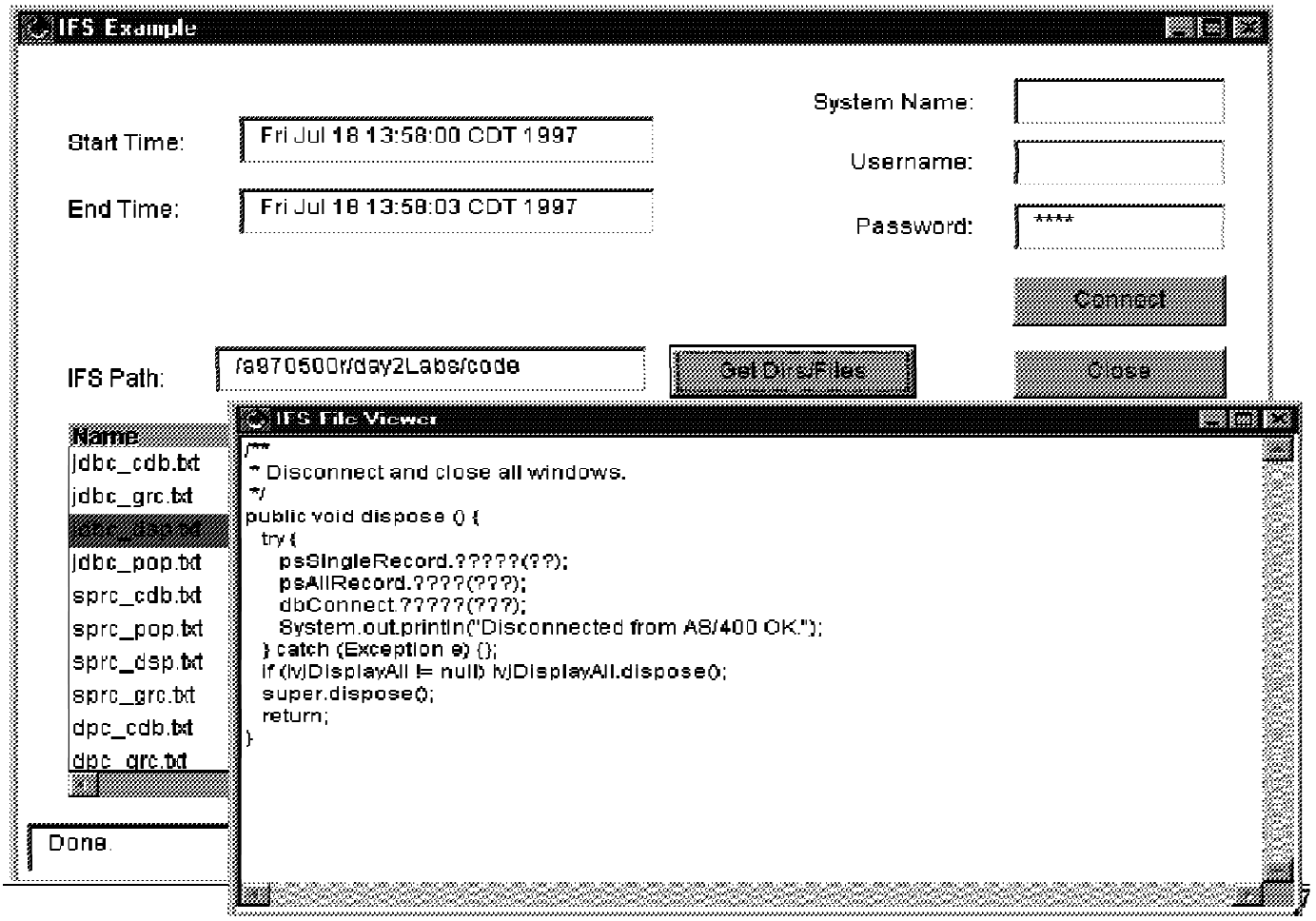
  Method in class COM.ibm.as400.access.IFSFileInputStream. Returns the number of available bytes in the file.

- 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 IFS classes of the AS/400 Toolbox to allow a Java program to retrieve a list of files from the AS/400 Integrated File System. If a text file is selected from the list of files, its contents are displayed. We use the Taligent multi-column list box to display the list of files. This window shows the files stored in the IFS for the path entered in the window's text box.

Application flow:

- Establish a connection.
- Set IFS Path to view.
- Create an IFSFile Object.
- Retrieve list.
- Open an IFSFileInputStream.
- Read file contents.
- Close the connection.

3.7.2 IFSExample Class

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

3.7.2.1 Connect Method

This 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;

}
```

Method highlights:

`as400 = new COM.ibm.as400.access.AS400(systemName, userid, password);`

Creates a new AS400 connection object. System name, user ID, and password are passed through the constructor.

`as400.connectService(AS400.COM.ibm.as400.access.AS400.FILE);`

Connect 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 PopulateList Method

This 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[] rowData = new Object[4];
 Object rowKey;
 .
 .
 try {
  // Create the IFSFile object given the system and path.
  aFile = new COM.ibm.as400.access.IFSFile(as400, IFSPath);
  // Get a String array list of files and directories.
  files = aFile.list();
  // Loop for each item in the list to get name, size, type, and modified data
  for (int i=0; i<files.length; i++) {
   aFile = new COM.ibm.as400.access.IFSFile(as400,IFSPath, files[i]);
   // The IFSFile object is added to the MultiColumnListbox as a key
   rowKey = aFile;
   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());
   getMultiColumnListbox1().addRow(rowData, rowKey);
  }
 } catch (java.io.IOException ex) {
  System.out.println("Error Receiving Files: "+ex);
 }
 ....
}

}
```

Method highlights:

```
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 Readfile() Method

This 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();
.
.
.
 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, 0);
  getFileContents().setText(t);
}
}
```

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. Overview of the CPW Application

In this chapter, we cover the CPW (Commercial Processing Workload) Benchmark order entry application. The CPW benchmark includes interactive and batch work and is used by the Rochester Development Lab to determine performance ratings of the various AS/400 processor features. The CPW benchmark itself is a modified implementation of the TPC-C (Transaction Processing Commercial-benchmark C) workload. Since CPW is not the actual TCP-C benchmark, performance metrics based upon CPW applications are **not** representative of IBM's or other vendor's optimized implementations.

However, CPW is representative of sophisticated commercial applications and industry standard benchmarks and is more complex than the original AS/400 performance rating RAMP-C benchmark, which was used in AS/400 releases prior to November 1996.

This section introduces the application and specifies the database layout. In Chapter 6, “Developing AS/400 Java Applets” on page 155, we build the CPW order entry application as an Internet-based applet that uses the JDBC interface to access the AS/400 database.

4.1 Overview of the Application

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

4.1.1 The Company

The Company is a wholesale supplier with one warehouse and 10 sales districts. Each district serves 3000 customers (30 000 total customers for the company). The warehouse maintains stock for the 100 000 items sold by the Company.

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

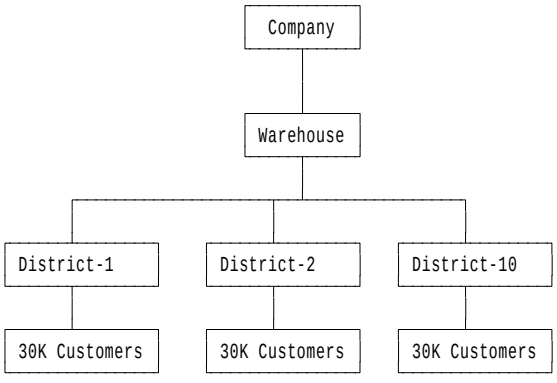


Figure 30. Company Structure

4.1.1.1 The Company Database

The company runs its business with a database. This database is used in a mission critical, OLTP (online transaction processing) environment. The database includes tables with the following data:

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

4.1.1.2 A Customer Transaction

1. Customers telephone one of the 10 district centers to place an order.
2. The district customer service representative answers the telephone, gets the following information, and enters it into the application:
   a. Customer number
   b. Item Numbers of the items the customer wants to order
   c. The quantity required for each item
3. The customer service representative enters the district number into the application.
4. The application then:
   a. From the Customer Table, reads the customer last name, customer discount rate, and customer credit status.
   b. From the Item Table, reads the item names, item prices, and item data for each item ordered by the customer.
   c. Reads the District Table for the district tax and the next available district order number. The next available district order number is incremented by one and updated.
   d. Inserts a new row into both the New Order Table and the Order Table to reflect the creation of the new order.
   e. Checks if the quantity of ordered items is in stock by reading the quantity in the Stock Table. The quantity is reduced by the quantity ordered and the new quantity is written into Quantity.
   f. A new row is inserted into the Order Line Table to reflect each item in the order.
   g. Writes a shipping record of the order (used to ship order).

4.1.1.3 Database Table Structure

The CSDB database has nine tables:

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

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

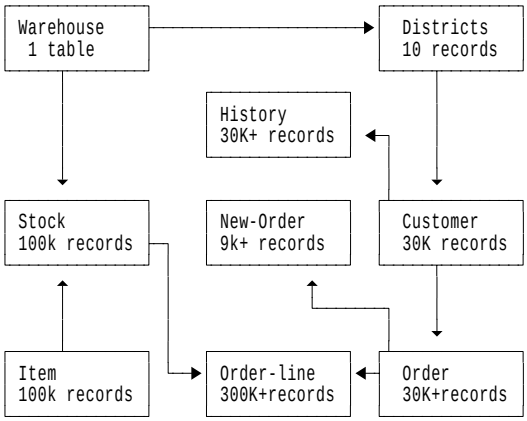


Figure 31. CSDB Database Table Relationships

4.2 CPW Benchmark Database Layout

The sample application uses the following tables of the CPW benchmark database:

- District
- Customer
- Order
- Order line
- Stock
- Item (catalog)

The following sections describe in detail the layout of the database tables.

4.2.1 District

Table 1. District Table Layout (DSTRCT)

Field Name	Real Name	Type	Length
DID	District ID	Decimal	3
DWID	Warehouse ID	Character	4
DNAME	District Name	Character	10
DADDR1	Address Line 1	Character	20
DADDR2	Address Line 2	Character	20
DCITY	City	Character	20
DSTATE	State	Character	2
DZIP	Zip Code	Character	10
DTAX	Tax	Decimal	5
DYTD	Year to Date Balance	Decimal	13
DNXTOR	Next Order Number	Decimal	9
Note: Primary Key: DID, DWID			

4.2.2 Customer

Table 2. Customer Table Layout (CSTMR)

Field Name	Real Name	Type	Length
CID	Customer ID	Character	4
CDID	District ID	Decimal	3
CWID	Warehouse ID	Character	4
CFIRST	First Name	Character	16
CINIT	Middle Initials	Character	2
CLAST	Last Name	Character	16
CLDATE	Date of Last Order	Numeric	8
CADDR1	Address Line 1	Character	20
CCREDT	Credit Status	Character	2
CADDR2	Address Line 2	Character	20
CDCT	Discount	Decimal	5
CCITY	City	Character	20
CSTATE	State	Character	2
CZIP	Zip Code	Character	10
CPHONE	Phone Number	Character	16
CBAL	Balance	Decimal	7
CCRDLM	Credit Limit	Decimal	7
CYTD	Year To Date	Decimal	13
CPAYCNT	Quantity	Decimal	5
CDELCNT	Quantity	Decimal	5
CLTIME	Time of Last Order	Numeric	6
CDATA	Customer Information	Character	500
Note: Primary Key: CID, CDID, CWID			

4.2.3 Order

Table 3 (Page 1 of 2). Orders Table Layout (ORDERS)

Field Name	Real Name	Type	Length
OWID	Warehouse ID	Character	4
ODID	District ID	Decimal	3
OCID	Customer ID	Character	4
OID	Order ID	Decimal	9
OENTDT	Order Date	Numeric	8
OENTTM	Order Time	Numeric	6
OCARID	Carrier Number	Character	2

Table 3 (Page 2 of 2). Orders Table Layout (ORDERS)

Field Name	Real Name	Type	Length
OLINES	Number of Order Lines	Decimal	3
OLOCAL	Local	Decimal	1
Note: Primary Key: OWID, ODID, OID			

4.2.4 Order Line

Table 4. Order Line Table Layout (ORDLIN)

Field Name	Real Name	Type	Length
OLOID	Order ID	Decimal	9
OLDID	District ID	Decimal	3
OLWID	Warehouse ID	Character	4
OLNBR	Order Line Number	Decimal	3
OLSPWH	Supply Warehouse	Character	4
OLIID	Item ID	Character	6
OLQTY	Quantity Ordered	Numeric	3
OLAMNT	Amount	Numeric	7
OLDLVD	Delivery Date	Numeric	8
OLDLVT	Delivery Time	Numeric	6
OLDSTI	District Information	Character	24
Note: Primary Key: OLWID, OLDID, OLOID, OLNBR			

4.2.5 Item (Catalog)

Table 5. Item Table Layout (ITEM)

Field Name	Real Name	Type	Length
IID	Item ID	Character	6
INAME	Item Name	Character	24
IPRICE	Price	Decimal	5
IDATA	Item Information	Character	50
Note: Primary Key: IID			

4.2.6 Stock

Table 6 (Page 1 of 2). Stock Table Layout (STOCK)

Field Name	Real Name	Type	Length
STWID	Warehouse ID	Character	4
STIID	Item ID	Character	6
STQTY	Quantity in Stock	Decimal	5

Table 6 (Page 2 of 2). Stock Table Layout (STOCK)

Field Name	Real Name	Type	Length
STDI01	District Information	Character	24
STDI02	District Information	Character	24
STDI03	District Information	Character	24
STDI04	District Information	Character	24
STDI05	District Information	Character	24
STDI06	District Information	Character	24
STDI07	District Information	Character	24
STDI08	District Information	Character	24
STDI09	District Information	Character	24
STDI010	District Information	Character	24
STYTD	Year To Date	Decimal	9
STORDRS	Quantity	Decimal	5
STREMORD	Quantity	Decimal	5
STDATA	Item Information	Character	50
Note: Primary Key: STWID, STIID			

4.3 Database Terminology

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

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

Table 7. Database Terminology

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

Chapter 5. Enterprise Access Builder For Data (DAX)

The VisualAge for Java Enterprise edition includes the following components:

1. Database access:

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

2. CICS access:

   This allows CICS transactions to be wrapper'd and used within a Java application.

3. C + + Server access:

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

4. Remote Method Invocation (RMI) access:

   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.

This chapter focuses entirely on the Database Access Builder (DAX) component of the VisualAge Java Enterprise Edition.

5.1 Enterprise Access Builder for Data (DAX)

Enterprise Access Builder for Data (referred to as Data Access Builder or DAX) is part of the VisualAge Java Enterprise edition that 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 yet 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, 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 provides better performance than JDBC data access.

- **Commitment control and connection**

  Services are provided for connecting to your databases. In addition, commit and rollback methods are also generated for transactions.

5.2 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.

5.2.1 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 then entering the quantity of the part being ordered.

The application uses three DB2/400 database tables that are described in the following table.

Table 8. 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 the following tables.

.

Table 9. 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 10. 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 11. 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

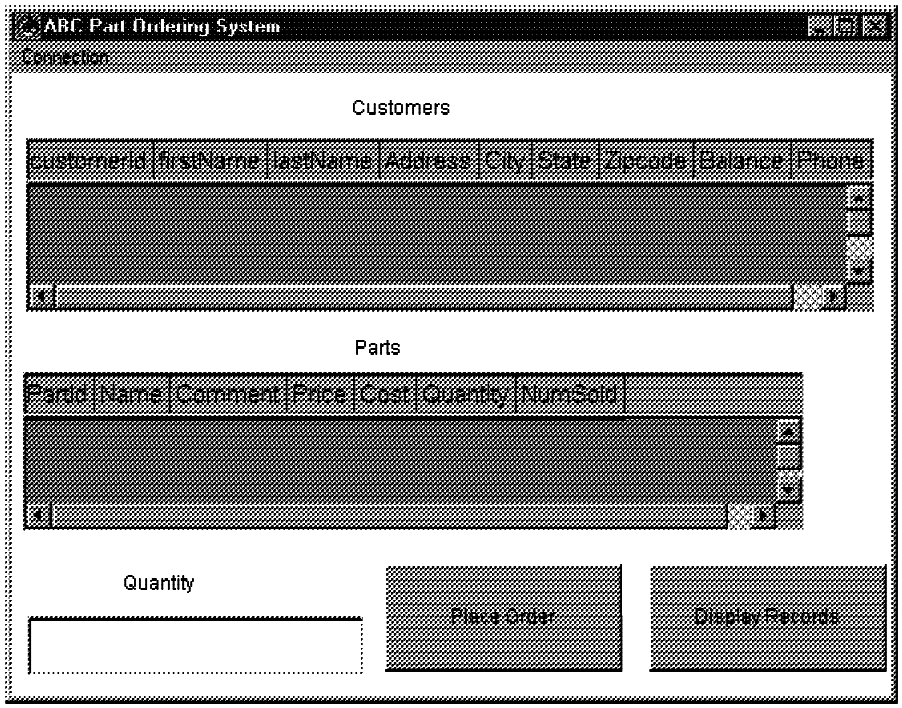


Figure 32. The Parts Order Management Window

The application we create is shown in the preceding figure. 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.

The following processing occurs when the "Display Records" push button is clicked.

1. All of the customer records are read from the database and placed in the Customer multi-column list box.
2. All the parts records are read from the database and placed in the Parts multi-column list box.

The following processing occurs when the "Place Order" push button is clicked.

1. The parts record field IQTY is reduced by the quantity ordered and 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 database.

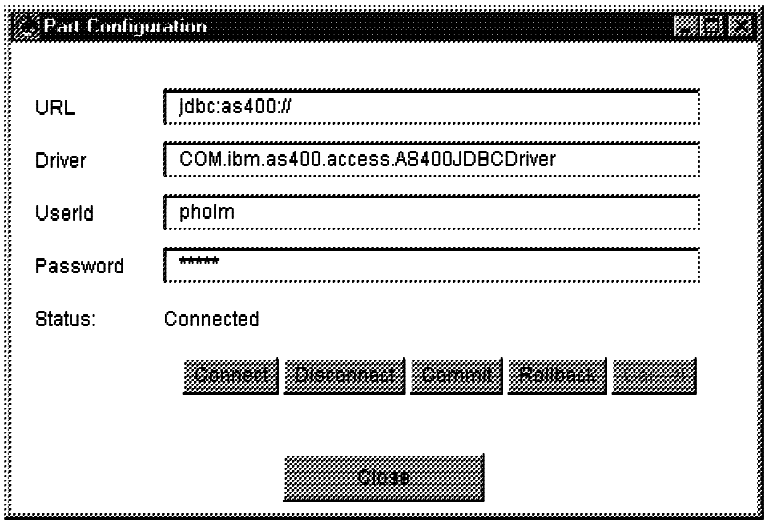


Figure 33. The Parts Configuration Window

The Parts Ordering Application "Parts Configuration" frame is shown in the preceding figure. It allows specifying a URL, JDBC driver, User Id, password, and commitment control option. The user uses this window to connect to the AS/400 system.

5.3 Generating the Application Using DAX

In this section, we build the complete ABC Part Ordering System application using DAX.

5.3.1 Understanding Our 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. The following diagram illustrates the use of Unified Methodology Language (UML) to describe our object model. UML is basically a diagramming language to describe object data properties, actions, and relationships with other objects. Refer to *www.rational.com* for more information on UML. 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:

1. Models the real world:

   OOP allows us to create objects and classes that are the same as their real world counterparts. This makes software simpler and more understandable.

2. Promotes re-usability:

   Objects can be created in an abstract way and then sub-classed or extended using inheritance. This allows object properties and operations to be reused.

The following UML object model illustrates the software design we used 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 while the bottom compartment contains the operations or methods that this class can perform.

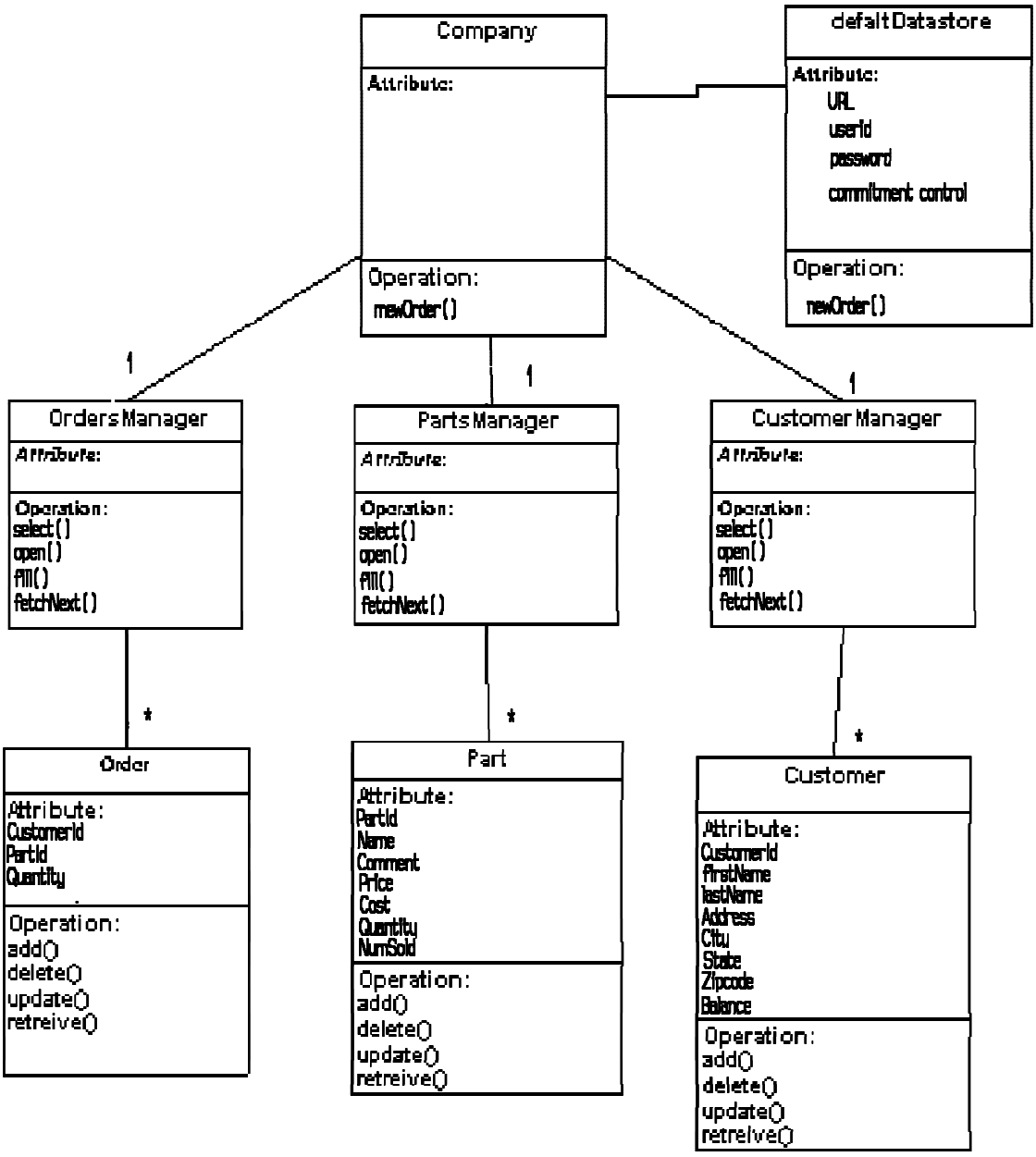


Figure 34. UML Object Model

The following table describes all of the classes previously illustrated.

Table 12. 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.

5.3.2 Building the Application

Without the use of the Data Access Builder, we must create all of the classes manually as well as coding all the logic to select, update, and insert records into the database. With the use of 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 we are about to create. We can start the Data Access Builder by choosing the **Selected** menu option and then selecting **Tools-Data Access-Create Data Access Beans** from VisualAge for Java Workbench menu. This brings up a Data Access Builder session window.

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 following figure:

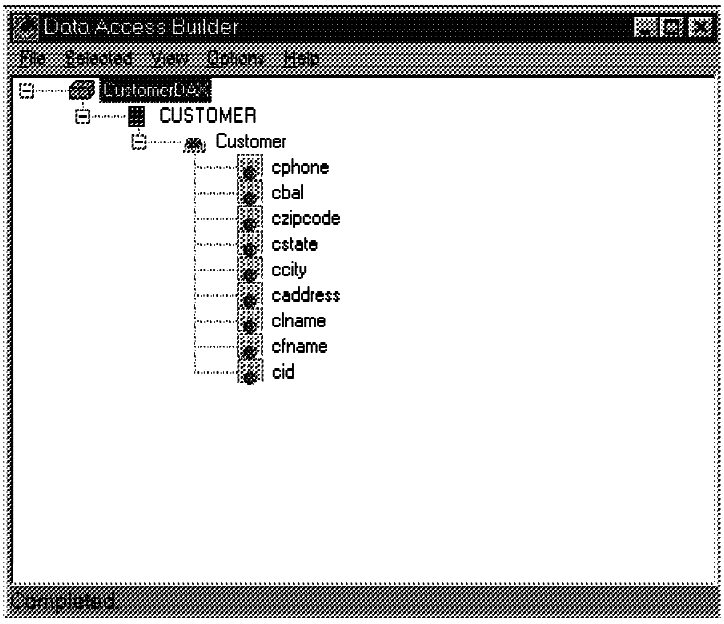


Figure 35. The DAX Generation Window

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 and 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 attributes settings of the Customer class shown in the following table. The data identifier, which is the field or fields that identify a record, can also be specified in this window.

We make the following changes by selecting **Attributes** from the Customer pop-up menu:

Table 13 (Page 1 of 2). 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

Table 13 (Page 2 of 2). Customer Table			
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 instaniating 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 following classes are generated by DAX and placed in the Java package:

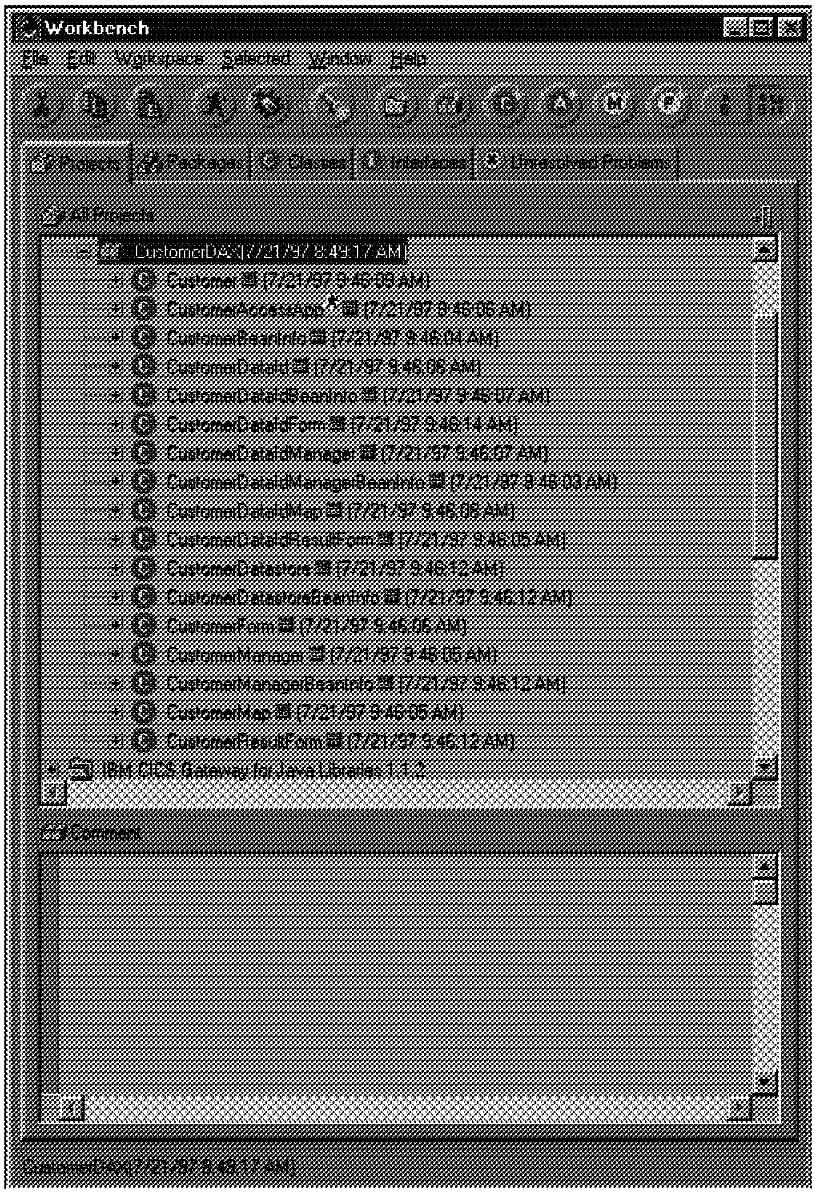


Figure 36. The Dax Generated Customer Window

The following table describes each generated class (bean).

Table 14 (Page 1 of 2). 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.

Table 14 (Page 2 of 2). Generated Classes	
CustomerManager	This class is responsible for selecting, updating, deleting, and creating new Customer objects. This class "manages" Customer objects.
CustomerManagerBeanInfo	This is the JavaBean information for the CustomerManager class.
CustomerMap	This is an internal DAX object that is used to automatically retrieve fields from the SQL/JDBC cursor and put them into a Customer object.
CustomerResultForm	This is a GUI container that contains a multi-column list box for displaying a list of customers.

Note: 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 then 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 these classes in the following figure.

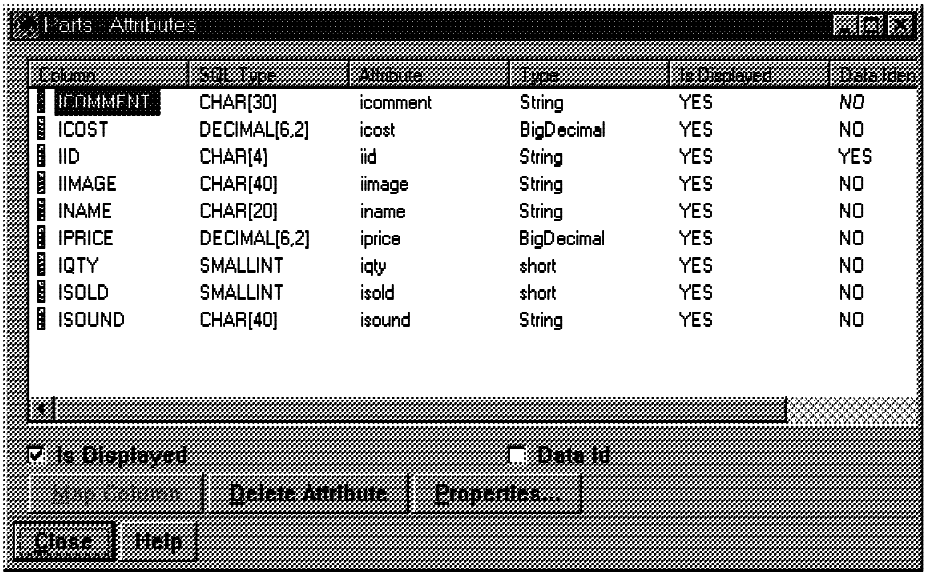


Figure 37. The Parts Attributes Window

Table 15. 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 16. 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

5.4 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, but 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.

The following figure 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 methods 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.

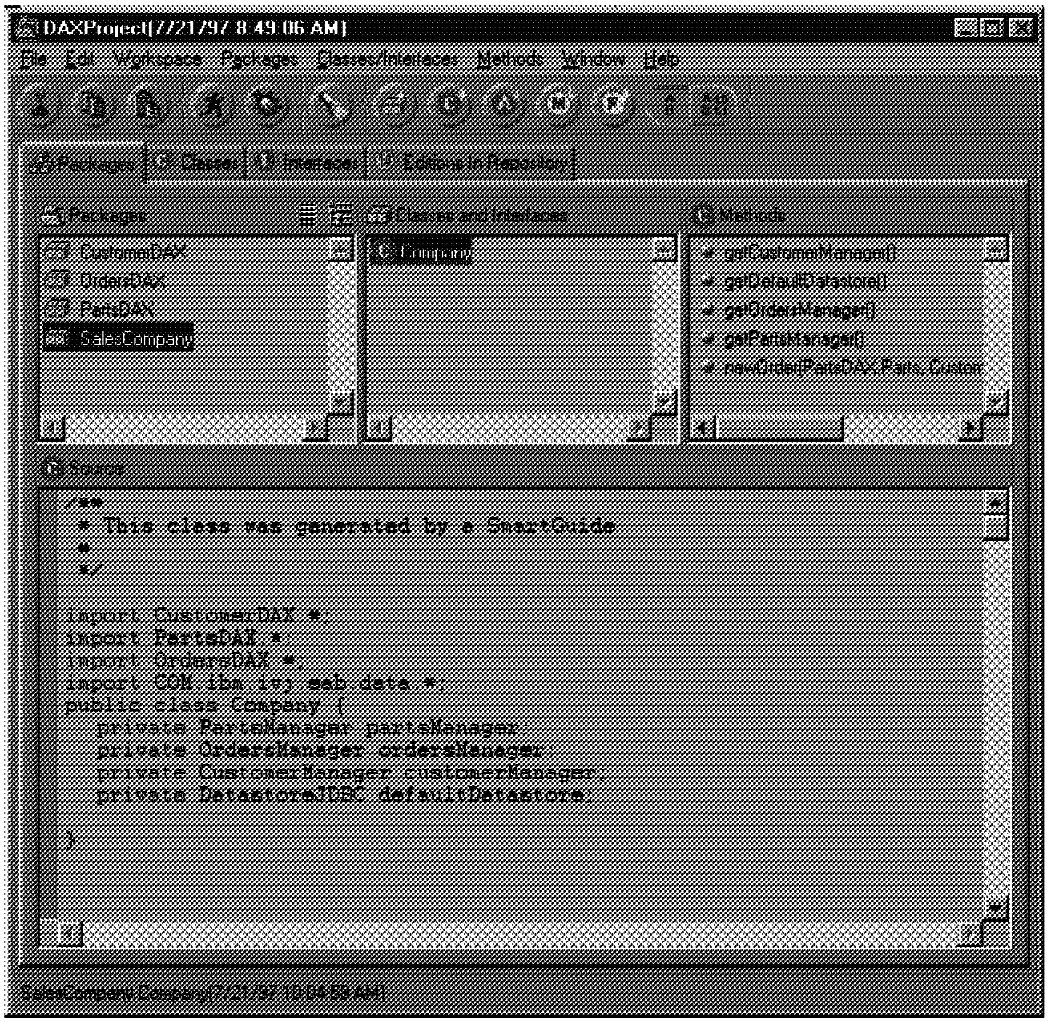


Figure 38. The DAXProject Window

The following methhod is used to get the customerManager object. This sample 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.

```
public CustomerManager getCustomerManager() {
          if (customerManager = =  null) {
          customerManager =  (new
         CustomerManager(getDefaultDatastore()));
          }
          return customerManager;
         }
```

The getDefaultDatastore() method that is shown in the following figure 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;
         }
```

The *newOrder (Parts aPart, Customer aCustomer, String aQuantity)* method shown in the following figure 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. This method accepts the three objects in the parameter list. The method then:

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.

```
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.setOrdertmsp(new
java.sql.Timestamp(System.currentTimeMillis())); // sets the timestamp to the current clock timestamp
/* Update the part inventory quantity and amount
          sold. */
 short newQuantity =
         (short)(aPart.getQuantity().shortValue() -
         quantitySold);
aPart.setQuantity(new
          Short(newQuantity));
short newNumSold =
          (short)(aPart.getNumSold().shortValue() +
          quantitySold);
aPart.setNumSold(new Short(newNumSold));
/** There is a bug in the EAB that prevents the update() methods from running. Instead of using update()
methods we will delete the part record and re-add it with the updated inventory quantity */
 Parts aNewPart =  (Parts) aPart.clone(); //Create a copy of the part record.
/* Delete and re-add the part */
 try {
aPart.delete();
aNewPart.add();
} catch(Exception e) {
System.out.println("Error updating part "+e);
}
/** Add the new order to the database */
try {
newOrder.add();
} catch(Exception problem) {
System.out.println("error
adding order " +  problem);
}
}
```

5.5 Building a Custom GUI Using DAX Objects

The last task is to create a user interface for our ABC parts ordering system. Basically, we just assemble and connect the classes that DAX created for us along with our custom *Company* class.

We first create a new class named OrderMainFrame that extends from java.awt.Frame that we use to compose our GUI windows.

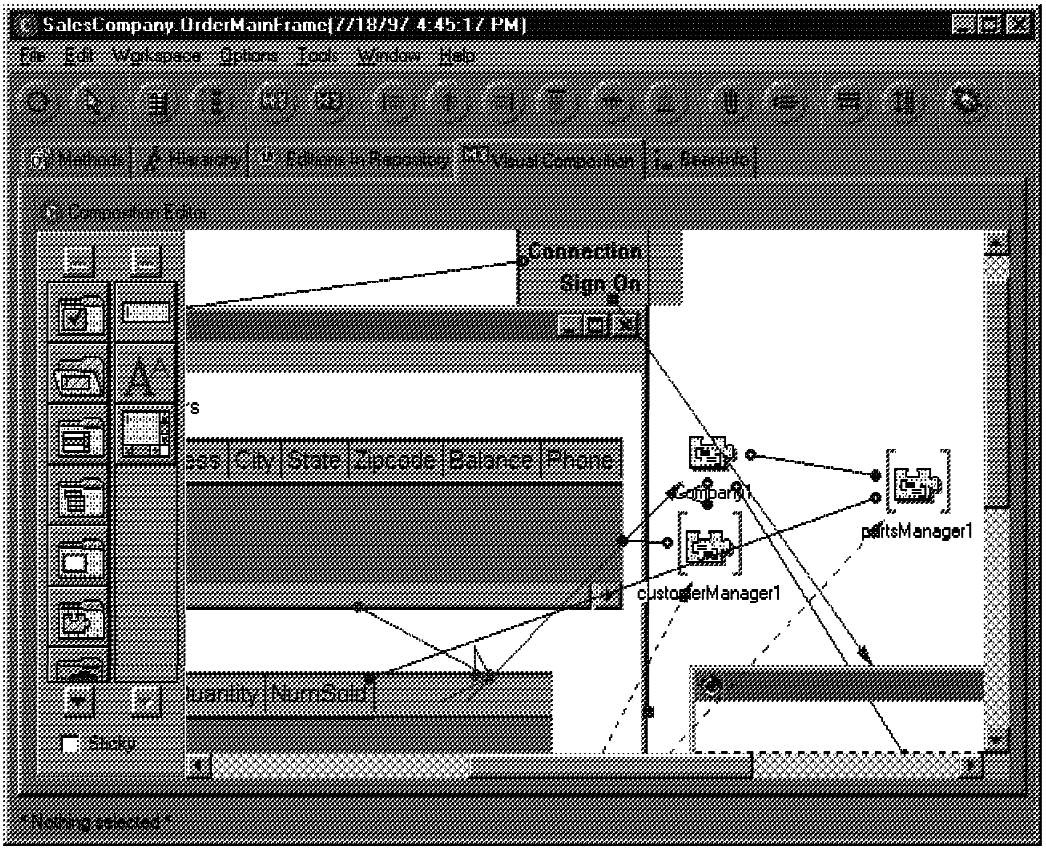


Figure 39. The OrderMainApp Composition Editor Window

We then use the Visual Composition Editor to:

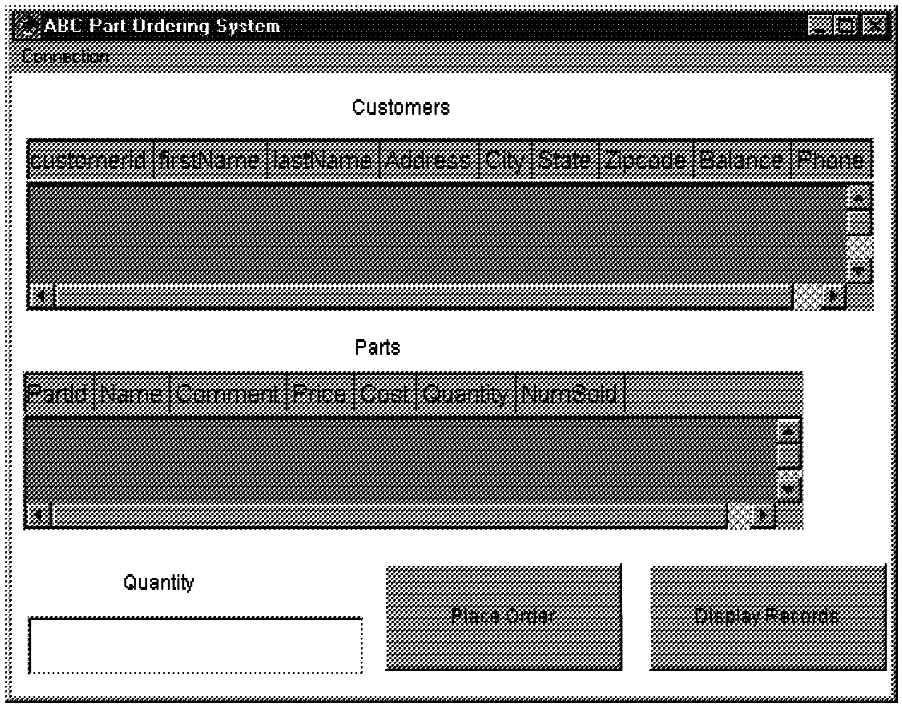
- Add the visual parts.
- Add the non-visual parts.
- Add the connections.

The following parts or classes are used:

Table 17. 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 looks similar to the following figure:



The following figure shows how your "Order Configuration" window will look.

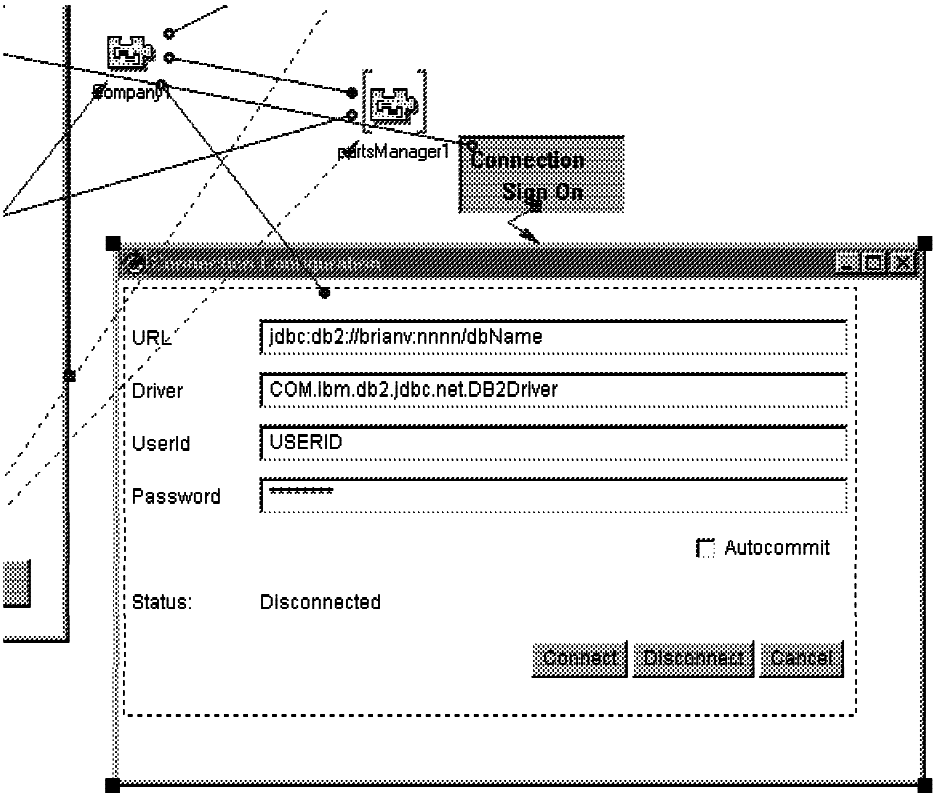


Figure 40. Order Configuration Composition Editor

Since the Company class contains the PartsManager and CustomerManager attributes, we use the "Tear off Property" popup 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 following connections complete the application.

Table 18 (Page 1 of 2). 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()

Table 18 (Page 2 of 2). Application connections			
Order pushbutton	actionPerformed	Company	newOrder(), use the quantity, SelectedObjects from the PartsResultForm and CustomerResultForm as the parameters.

5.6 The Completed Application

The following window shows the completed application. Refer to the VisualAge for Java online documentation for further information about DAX.

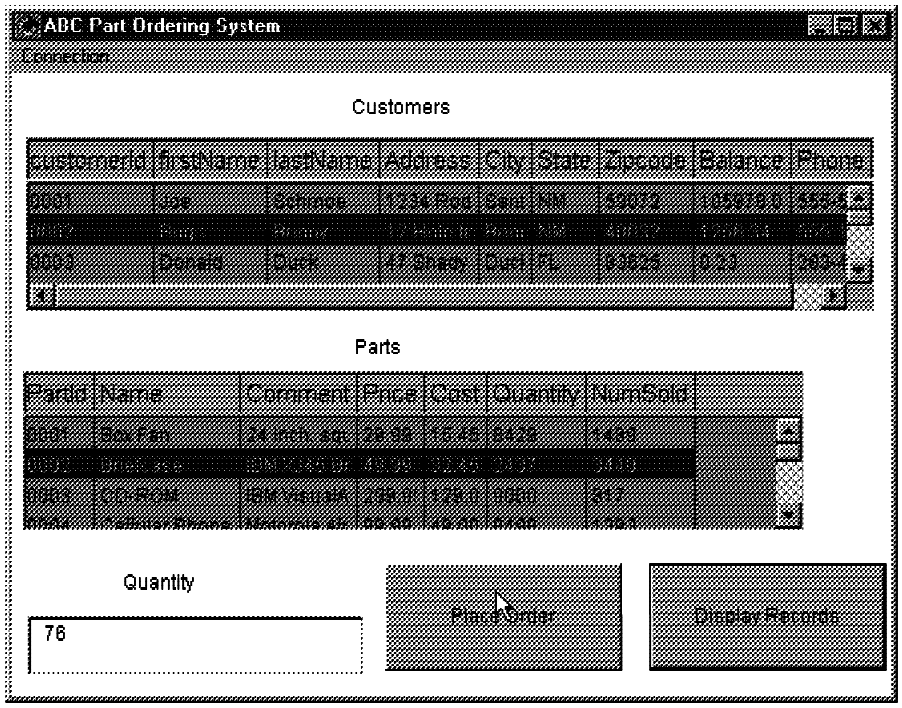


Figure 41. The Completed Application

5.7 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 duplicate 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. DAX should be considered for any serious programming efforts. The DAX support is only available with the Enterprise Edition of VisualAge for Java.

Chapter 6. Developing AS/400 Java Applets

This chapter discusses developing AS/400 client/server applications that run as Java applets under the control of a Web browser.

The topics covered here include:

- Java Applets
- HTML for Applets
- Limitations of Applets
- Building the AS/400 "Shopping Cart" applet
- JAR files

We investigate an AS/400 Internet-based shopping application as an example throughout this chapter. Before we go into the details of the Internet shopping application, we provide an overview of Java applets.

6.1 Applet Class Structure

The Java Development Toolkit, which is available from Sun Microsystems, Inc., supplies a number of packages that provide the base support for Java. We refer to it as the JDK. The latest release of the JDK is JDK 1.1 with version 1.1.1 being the latest version available. The earlier release of the JDK is called JDK 1.0, with JDK 1.0.2 being its latest version. In this chapter, we use JDK 1.0 to refer to the latest version of JDK 1.0 and JDK 1.1 to refer to the latest version of JDK 1.1.

In the JDK 1.1, the Applet class definition is:

```
public class MyAppletClass extends java.applet.Applet
         implements java.awt.event.ActionListener
```

Applet extends (sub-classes) the class java.applet.Applet and implements the interface java.awt.event.ActionListener. Being a sub-class of java.applet.Applet, it inherits from Panel, Container, and Component. It inherits all the methods and interfaces of its ancestors.

For example, Methods inherited include:

- getCodebase() method, which returns the URL from which the Applet is loaded.
- add() method, which allows other GUI components to be added to your Applet.

We go through some of the essential methods. More can be found by reading the ancestors' classes.

A Java interface is the same as an abstract class. It allows us to define methods without actually implementing them. We can then implement the interface in a class. Since Java allows a class to have only one super class, we do not have the concept of multiple inheritance. Implementing interfaces allows us to get around this restriction.

An interface is a collection of the method's structure definition. Interfaces:

- Define the names of methods supported.

- Provide the input and output parameters definitions (type).

Internet browsers are designed based on the fact that because all applets are extended from java.applet.Applet, they can use the methods inherited from the Applet class to control an Applet. They mainly use init(), start(), stop(), and destroy(). Some other methods you normally encounter are update(), paint(), and repaint().

The ActionListener interface is the way that the JDK 1.1 implements the Event model; it allows the browsers to notify the Applet of any events (such as Mouse Move, Clicked, Enter). An example of how ActionListener works is when the user clicks a button, an Action Event is sent to the actionPerformed() method. Then it is up to the actionPerformed() method to determine what to do with this event.

The Action Event is an object that provides further information such as who generates the event. It can also sub-divide the Event into different types (left-clicked, double-clicked). Java programmers can define their own type of event and listener interface. VisualAge for Java provides some common events and listeners that the programmer can implement.

A Java applet goes through a life cycle of being loaded by a user in a browser, having users leave and return to the applet's page, and finally having the user end the browser. When the browser loads the applet, an instance of the applet sub-class is created. The applet gets initialized and the browser starts running it. The browser does so by calling the constructor of the applet class, where the constructor of the applet is the name of the class. For example, if the applet is named "MyApplet", the constructor is the method "MyApplet(...)". Next, the init() method is called. The init() method is where you can initialize your own variables and objects. You can also initialize your variables in the first declaration of the variables. Finally the start() method is called to start running the applet. Sometimes, especially in simple applets, it is not necessary to write code for all these methods.

When users leave the applet's page (for example, when the user minimizes the window) the applet stops running. When the user reopens the window, it starts running again. The browsers do this by calling Start() whenever the user maximizes or reopens the Applet window or browser window. It calls the Stop() method whenever the user minimizes the applet or browser window. The Start() and Stop() methods are also used to control threads.

Finally, when the user exits the browser, the applet stops and it does some final cleanup before the browser exits. The browser calls the stop() and then the destroy() methods. Destroy() is used to deallocate resources such as AS/400 connections, server programs, and for releasing non-Java resources. This ends the life of the applet.

6.1.1 Applet Limitations

Because applets are run inside a Web browser and the user may load an applet from an external Web site, restrictions must be placed on the capabilities of an applet. Without restrictions, an applet can damage the user's system by doing things such as loading viruses or deleting files. The following restrictions apply to applets:

1. Applets cannot make network connections except to the host that it was loaded from.

2. An applet cannot load libraries or define native methods.
3. An applet cannot ordinarily read or write files on the host that is executing it.
4. Applets cannot start any program on the host that is executing it.
5. Applets cannot read certain system properties.

Each browser has a SecurityManager object that implements its security policies. When a SecurityManager detects a violation, it throws a SecurityException. An applet can catch this SecurityException and react appropriately.

Some of the preceding limitations can be relaxed by using Signed Applet support, which provides key encryption, or by running a trusted applet that is downloaded from a trusted host.

6.1.2 Applet Capabilities

1. Applets running within a Web browser can easily cause HTML documents to be loaded.
2. Applets can invoke public methods of other applets on the same page.
3. Applets that are loaded from the local file system (from a directory in the user's CLASSPATH) have none of the restrictions that applets loaded over the network do.
4. Although most applets stop running once you leave their page, they do not have to.

6.1.3 HTML Tags for Applets

JAR files are used to compress and package applets. A JAR file can contain all the classes required by an applet. It can also contain any supporting files (for example, image files). Because JAR files are compressed, the number of bytes that must be transferred when loading an applet in a browser is significantly reduced. JAR support is only available with the JDK 1.1.

```
<HTML>
<applet code="ToolboxApplet.ToolboxApplet"
width=500 height=400 archive="applets.jar">
<hr>
This Applet is only seen
on JDK1.1 compatible Browser
<hr>
</applet><p>
```

This is the HTML tag for an applet in a JAR file. We discuss how to compress files using JDK 1.1's JAR utility later. If you do not use JAR files, you can omit the "archive=" parameter. The browser then tries to load the class in the Package directory where the Homepage is loaded from. The HTML tag is the same for the JDK 1.1 or 1.0. The only difference is that only a JDK 1.1 enabled-browser understands the "archive=" tag.

In this example, the browser loads the file "ToolboxApplet\ToolboxApplet.class" and sets its size to 500 x 400 pixels. Using layouts allows the applet to re-size itself according to the HTML tag. Otherwise, if the height and width tag in the HTML tag are not set properly, part of the applet may be hidden. We do not cover layout definitions in this redbook, but you can reference it in other Java Text books.

Remember that a Package name is case sensitive. Although the Windows 95 directory name is not case sensitive, directory names may be case sensitive on other platforms. In the preceding example, the class is loaded from **Home Page directory\ToolboxApplet**.

6.1.4 Browser Versioning

One of the problems that you can encounter when writing applets is that not all browsers support the latest version of the JDK. In this section, we discuss which browsers support JDK 1.1 and can run JDK 1.1 applets. We also discuss the key differences between JDK 1.0 and JDK 1.1.

Here is a list of some of the more popular browsers and what level of applets they support:

- HotJava 1.0:
  - Supports the JDK 1.1.
  - Can run JDK 1.1 applets.
- Netscape Communicator Preview Release 4.0:
  - Does not fully support the JDK 1.1:
    .
    - Cannot run any of the JDK 1.1 GUI applets.
    - Some important Java Packages are missing.
- Netscape Communicator Preview Release 5.0:
  .
  - Does not fully support the JDK 1.1:
    - Can run some JDK 1.1 GUI applets.
- Microsoft Internet Explorer 4.0:
  - Does not support the JDK 1.1.
  - Cannot run any the JDK 1.1 GUI applets.

During the development of this redbook, the JDK 1.1 had just become available. Therefore, we found that most JDK 1.1 products were available as test versions only. We found that all the applet work we did and have documented here can be run in Sun's JDK 1.1, the HotJava Browser, as well as VisualAge for Java. Other browsers such as Netscape Communicator are starting to implement support for the JDK 1.1, so in the future, all the examples shown here should be able to run with many other browsers.

6.1.5 JDK 1.1 Applets versus JDK 1.0 Applets

In this section, we discuss developing applets with the JDK 1.1 and JDK 1.0 and the compatibility issues involved when using a mixture of JDK 1.1 and JDK 1.0 applets. If you are developing new applets, the JDK 1.1 is the best choice. The JDK 1.1 provides many bug fixes and enhancements over version 1.0. It also uses the new Event model. This is the main reason for its incompatibility with version 1.0. Some of the older browsers do not support the new event models and, thus, cannot run version 1.1 applets. Even when every browser available supports the JDK 1.1, users will take time to switch to the new browsers. Until everyone has switched, you probably want to supply 1.0-compliant applets. As long as you take care to provide backward compatibility, you can use 1.1 features and still provide applets that most browsers can display.

When designing Applets, you have to consider supporting browsers with limited capabilities. The following ways are common in dealing with the compatibility problem.

- One applet, mixed code:
  - Write a single program that encloses the JDK 1.1 API calls in try/catch statements.
  - The code is not elegant.
  - This technique does not let you take advantage of every new feature, since you cannot put class and method definitions in try/catch statements.
- Switcher applet, dynamically loaded code:
  - Specify a "switcher" applet in your APPLET tag.
  - The switcher applet determines whether the 1.1 applet or 1.0 applet is loaded.
  - It displays the 1.1 or 1.0 applet, treating it the same as a panel.
  - Write a switcher, 1.0, and 1.1 applet.
- Two applets, two HTML pages:
  - If neither of the preceding techniques suit you, you can always provide two pages, one for each version of the applet.

More details on applet compatibility can be found on the following Web page:

`http://java.sun.com/products/jdk/1.1/compatible/index.html`

This finishes the applet overview part of the chapter. We now show developing an Internet-based shopping applet that uses the AS/400 system as a database server.

6.2 Internet Shopping Application Example Introduction

In this section of the chapter, we investigate the shopping applet. It is actually a set of three applets. These applets use the CPW databases that are described in Chapter 4, “Overview of the CPW Application” on page 129. The databases are AS/400 databases and are accessed using JDBC. This suite of applets allow customers to select items from the Items database, place and confirm orders, and to check on the status of orders. Chapter 3, “AS/400 Toolbox for Java” on page 61 shows how Java can used to build traditional AS/400 client/server applications, while here we show how the application can be implemented as an Internet-based applet. The first two applets are the ToolboxApplet and CartApplet.

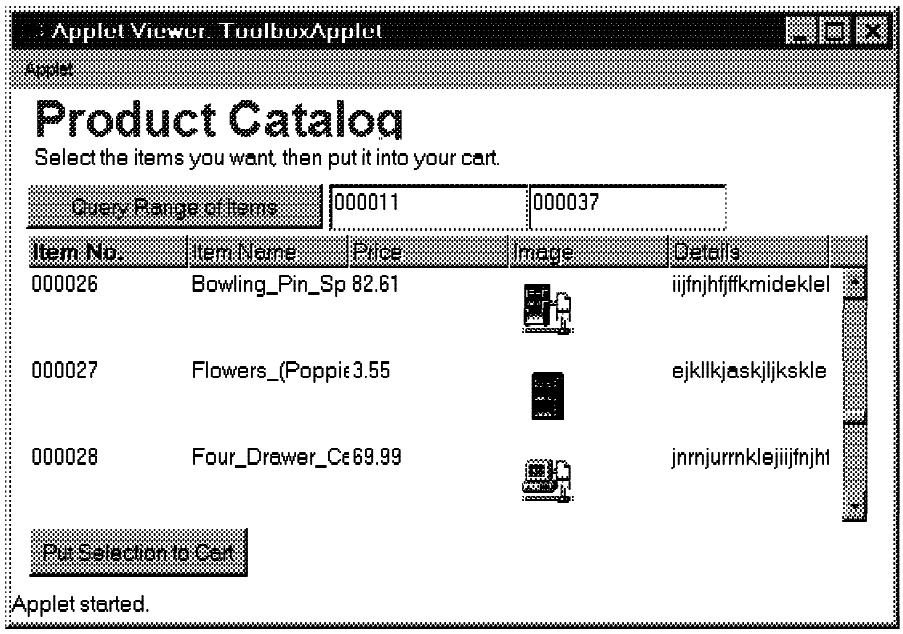
- ToolboxApplet
  - An Applet for querying the items database and selecting items to be ordered
- CartApplet
  - For checking on what items have been selected
  - For placing and confirming an order for those items

Normally the ToolboxApplet works in conjunction with the CartApplet. The third applet is independent of the preceding two applets.

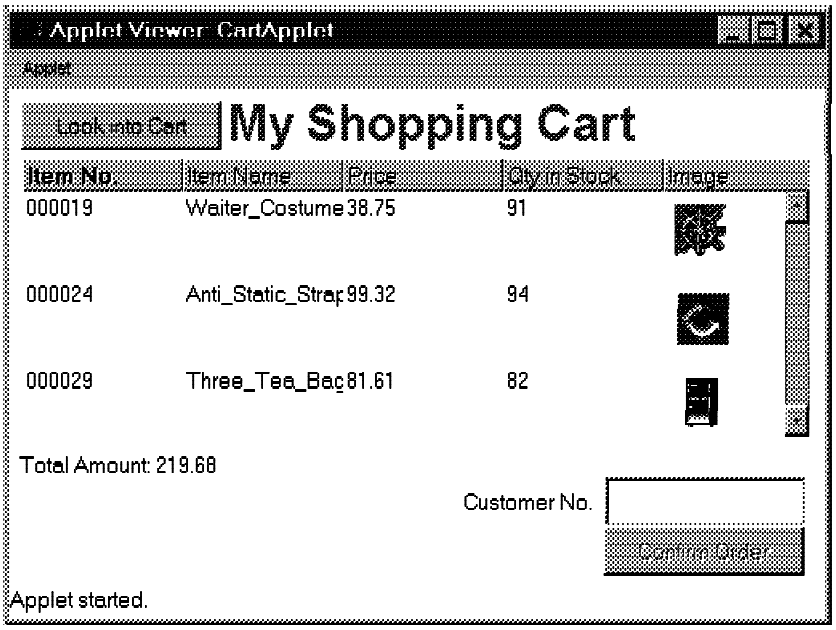
- StatusApplet
  - For checking the status of an order

6.2.1 Shopping Application User Interface

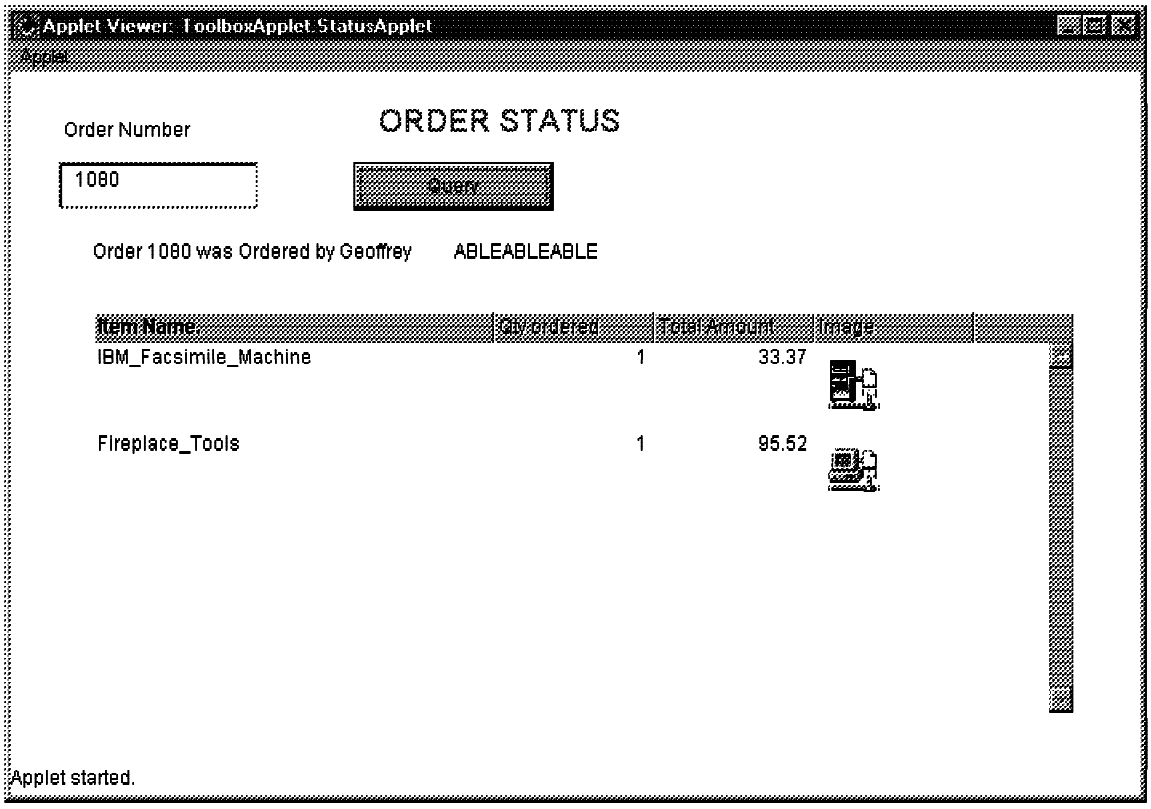
- We investigate three different applets as shown in the following figures.
- (The images shown were captured when running in the Applet Viewer from Sun's JDK 1.1.)



"ToolboxApplet" -- the Product Catalog Searching Engine: The preceding figure is the "ToolboxApplet", which is the first applet we investigate. It queries the ITEMS database for product information and puts the results in the list box shown. Customers can then select the items they want and put them into a "Shopping Cart". Later, they can use the "CartApplet" to check on their selected items and confirm them for ordering. They can also use the "StatusApplet" to check on the status of their orders.



"CartApplet" -- the Shopping Cart Where You Place an Order: The preceding shopping cart applet is expected to be in a frame that always shows in the browser until the customer leaves the shop. Items in the cart are shown together with the total amount of the items selected. Customers confirm their order by typing in a valid customer number. If the customer number is valid, the "Confirm Order" button is enabled so that the customer can confirm the order. On confirmation of the order, the system returns an "Order Number" that can be used to keep track of the order status.

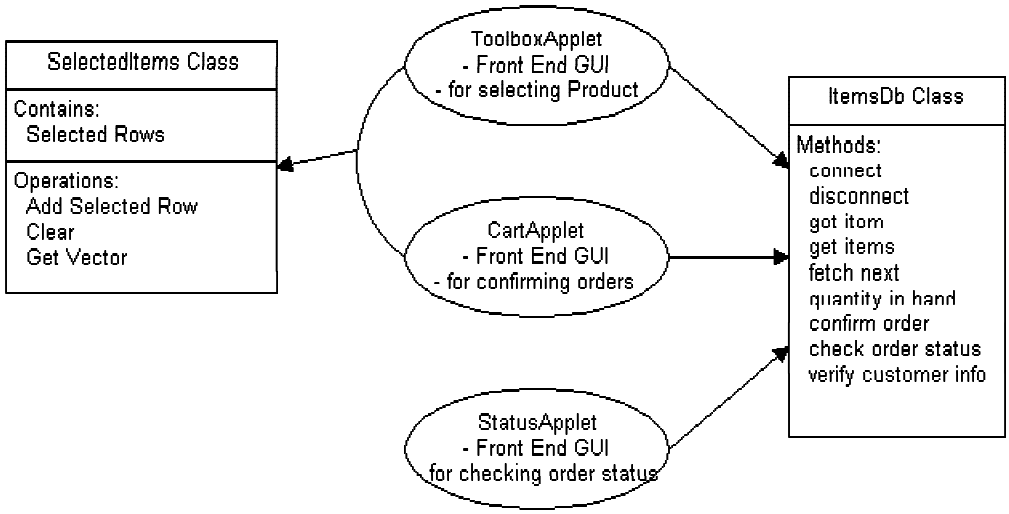


"StatusApplet" -- an Applet Where You Can Check for the Order Status: The preceding applet allows customers to check their order status by typing in the "Order Number" they received when they confirmed the order.

The Shopping Application has the following limitations. These limitations can easily be eliminated by adding additional function to the applets.

- The Toolbox applet only allows you to order a quantity of one.
- The CartApplet does not allow you to delete items in the cart.

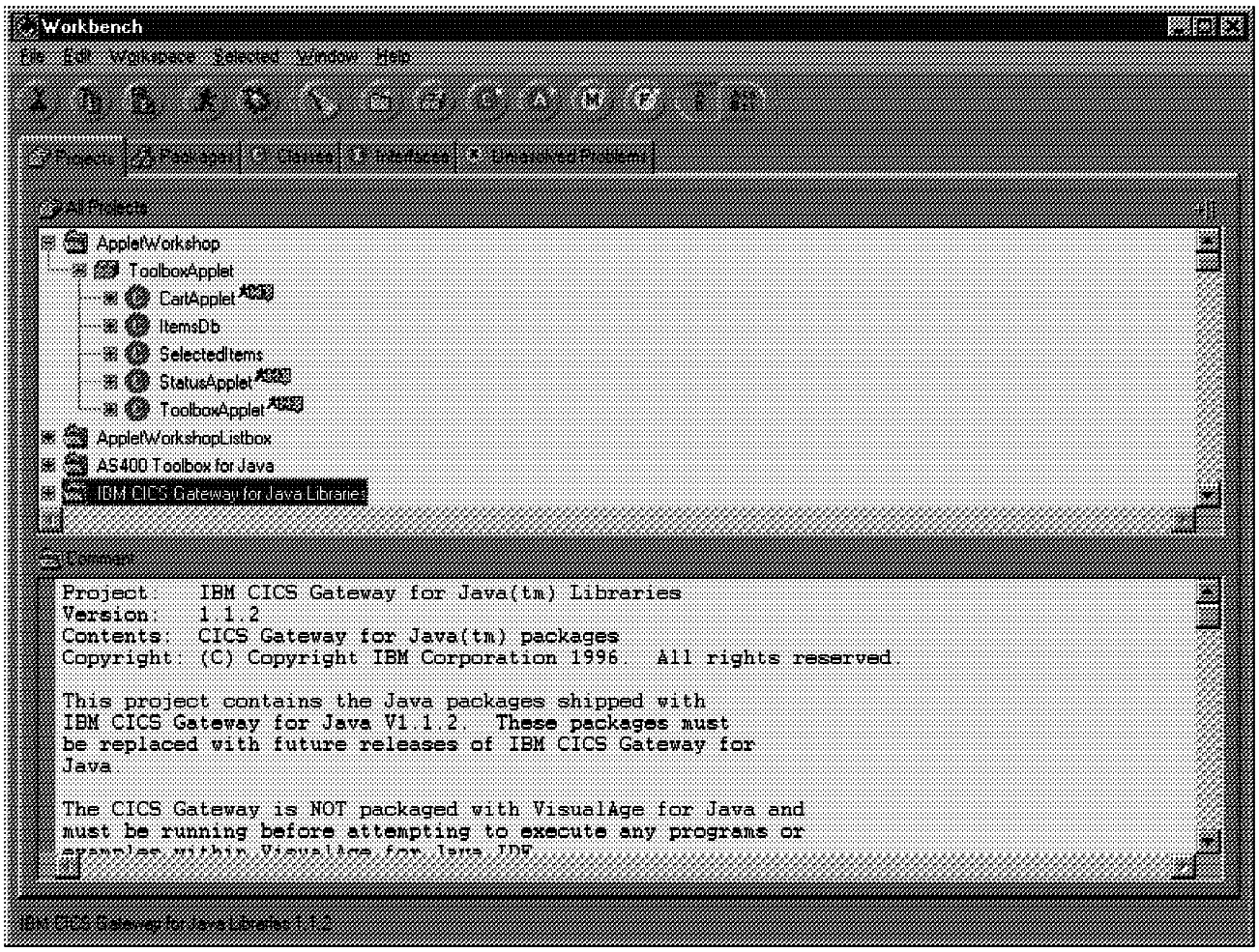
6.3 Shopping Application Objects and Classes



We have a project named "AppletWorkshop" that holds our package "ToolboxApplet" and our classes, which include:

- "ToolboxApplet" -- the product catalog searching engine.
- "CartApplet" -- the shopping cart where you place an order.
- "StatusApplet" -- an applet where you can check for the order status.
- "ItemsDb" -- a class for accessing the AS/400 database for this application; it is used by all the preceding applet classes.
- "SelectedItems" -- a class for storing items you selected; the applets can put things into the cart or see what is in the cart.

We built the "ItemsDb" and "SelectedItems" classes for others to use instead of writing the same code for every application or program. When you do your own design, you may decide to implement more generic classes that can be reused more readily.



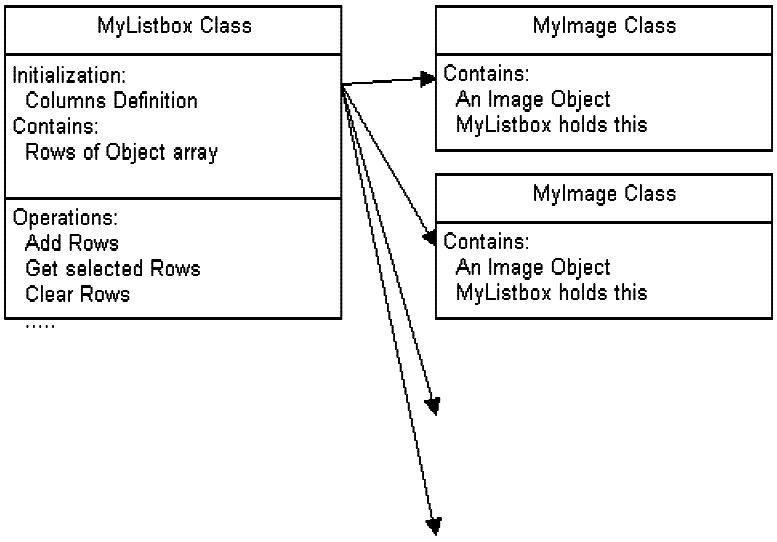
The preceding screen shows the VisualAge "WorkBench". The "runner" superscript on the top right hand corner of a class means that it is a runnable Java application or applet.

We also use several other classes:

- A simple MultiColumnListbox created by Taligent that is used to display the selected items.
- The AS/400 Toolbox classes that allow access to the AS/400 system.
- The JAVA classes in Java Classes Library.

For some of the background of the MultiColumnListbox from Taligent:

- It is available at www.taligent.com.
- We used a beta version of it and created our own sub-classes: MyListbox and MyImage.
- We placed it in the itsc.taligent.widget package.



MyListbox class:

- Is a sub-class of the MultiColumnListbox.
- It over rides the reshape() method to allow forced repaint without passing any graphics object parameters. This allows you to refresh the list box anytime you want to.

MyImage class:

- A simple support class containing the Image object and also the list box object.
- When the image in the list box finishes loading in its own thread, the list box is repainted.

We do not cover how to modify it in the redbook, but we provide all the source in the appendix.

One other class that may be a little bit complicated is java.util.Vector:

- Vector is a collection of different objects.
- The difference between a vector and an array is:
  - An array is a collection of **the same type** of objects.
  - A vector is a collection of different or the same type of objects.
- VectorEnumeration is a support class:
  - It allows you to scroll through the entire list of objects inside a Vector.
  - It provides two main methods:
    - getNextElement()
    - hasmoreElements()
- Elements can be of different types in a vector; you have to type cast the element to the proper type before using:
  - For example, ImageExample = (Image) getNextElement()
  - Provided that you know the next element is an image.

6.4 Testing the Applets

You can run the ToolboxApplet and the StatusApplet inside the VisualAge for Java IDE. The CartApplet must be exported and run outside the IDE.

We used the Sun JDK 1.1 applet viewer to test outside the IDE.

To run the applets outside the VisualAge IDE, you have to export the applet classes to the proper directory. We use **C:\mytest** as the directory in this example. Export the following classes to the package directory (C:\mytest\ToolboxApplet):

- ToolboxApplet
- CartApplet
- StatusApplet
- ItemsDb
- SelectedItems

You need to put all the homepage(*.htm files) in "C:\mytest". The homepage (*.htm files) refers to the applet class files in the package directory with the APPLET tag. Any applet viewer or browser tries to find them in "ToolboxApplet" under the current directory where you load your homepage(*.htm). The AS/400 Toolbox classes and any third party classes should be placed under "\MYTEST" followed by the package name as the sub-directory.

The preceding procedures are necessary because Java applications look into the CLASSPATH directory to find the appropriate classes. But when you write applets, assume that the browsers or AppletViewers do not know about the classes that do not come with the JDK. Therefore, you must put them under the directory tree in this way so they are found.

Have the Path set up so that Sun JDK's Appletviewer can be called from anywhere.

When you want to test your applet with Sun JDK's Applet Viewer, type:

```
AppletViewer TestHomePage
```

This should open your TestHomePage.htm in the current directory (C:\MYTEST). Three HTM files are required to run the applets:

"AppletViewer Order.htm"	-- which runs both "ToolboxApplet" and "CartApplet"
"AppletViewer Order1.htm"	-- which only runs "ToolboxApplet"
"AppletViewer status.htm"	-- which runs "StatusApplet"

All the HTM files point to the class files in the Package directory.

This is the HTM tag for Order.htm:

```
<HTML>
<applet code="ToolboxApplet.CartApplet" width=700 height=400>
  <hr>
  This Applet would only be seen on JDK1.1 compatible Browser
  <hr>
  </applet><p>
<applet code="ToolboxApplet.ToolboxApplet" width=700 height=400>
```

```
  <hr>
  This Applet would only be seen on JDK1.1 compatible Browser
  <hr>
    </applet><p>
```

This is the HTM tag for Order1.htm:

```
<HTML>
<applet code="ToolboxApplet.ToolboxApplet" width=500 height=400>
  <hr>
  This Applet would only be seen on JDK1.1 compatible Browser
  <hr>
  </applet><p>
```

This is the HTM tag for Status.htm:

```
<HTML>
<applet code="ToolboxApplet.StatusApplet" width=600 height=400>
  <hr>
  This applet is only seen on JDK1.1 compatible Browser
  <hr>
  </applet><p>
```

6.5 The "SelectedItems" Class

We investigate the *"SelectedItems"* class for use with the *"ToolboxApplet"* and the *"CartApplet"*.

"SelectedItems" acts as a buffer class that stores items selected by the customer. As the customer views items that are available, they can select items and place them in their "Shopping Cart". Later, they can view the items they selected. The selected items are stored in the "SelectedItems" class.

"SelectedItems" contains a **STATIC** vector named *"wanted"* that contains all the items selected. This is the "shopping cart" where we keep the selected items. It also contains a **STATIC** BigDecimal *"totalAmount"* that stores the total dollar amount of items selected. We use **STATIC** because we want other classes to share these variables.

6.5.1 Writing the Class

The class definition is as follows:

```
import java.util.Vector;
import java.math.BigDecimal;
/**
* This Class was generated by a SmartGuide.
*
*/
public class SelectedItems extends java.lang.Object {
private static Vector wanted;
static BigDecimal totalAmount;
}
```

The two import statements tell the compiler which package is used. The BigDecimal class is in the java.math package, and Vector is in the java.util package.

In some cases, generic import statements are used where instead of writing "import java.math.BigDecimal;", we can write "import java.math.*;" so that all classes in the java.math package are known to the compiler.

We recommend using the previous specific import statement rather than the generic one, as specific imports can also let others know exactly what other classes are used in the code. This facilitates modifying code and packaging the application using JAR files.

6.5.2 Writing the Methods

First we write the *getVector()* Method to let others access the Vector **"wanted"**, as we intentionally made it private. Making the variables private lets the class owner have more flexibility for future changes because other class users can only access them through the public method and not access the variables directly. Also, if no one instantiates the Vector, we do so here. This is called lazy initialization so that later if the Vector is disposed, it is regenerated when needed.

```
public Vector getVector () {
if(wanted==null) wanted=new Vector();
return wanted;
}
```

Second, we write the *clear()* method so the other applets can clear the cart and remove selections from the cart.

```
public void clear () {
wanted=null;
}
```

Finally, we write the *addSelectedRows()* method to allow items to be added to the Vector "wanted" that we previously created. An object array is used as the input parameter.

We must add the object array to our vector and then add the price of the item to the *totalAmount*. We have put the price of the items in the third element of the input Object Array; that is, Object[2].(remember all arrays start with 0). You can also check out the BigDecimal Class we used to see what other calculation methods are available. Because we may not have initialized the *"totalAmount"*, we set it to zero if it is empty.

```
public void addSelectedRow (Object[] row) {
getVector().addElement(row);
if(totalAmount==null) totalAmount=new BigDecimal("0");
totalAmount=totalAmount.add(new
        BigDecimal((String)row[2]));
}
```

6.6 The "ItemsDb" Class

ItemsDb class is the database access class we use. All the GUI front-end applets use this class to access the AS/400 databases.

The class structure is as follows:

```
import java.math.*;
import java.util.*;
public class ItemsDb extends java.lang.Object {
private java.sql.Connection dbConnect;
private java.sql.PreparedStatement psItem;
private java.sql.PreparedStatement psItemRange;
private java.sql.PreparedStatement psCustomerDb;
private java.sql.PreparedStatement psQuantityInHand;
private java.sql.Statement sGetInetOrderNo;
private String systemName =  new String("TCPASM02");
private String userid =  new String("UUUUUUUU");
private String password =  new String("PPPP");
private java.sql.ResultSet rs =  null;
public String itemId;
public String itemName;
public BigDecimal itemPriceBigDecimal;
public String itemPrice;
public String itemInfo;
public String validCustomerId =  null;
}
```

Basically, the class definition declares all the variables that we use in this class.

Important Information

To run the shopping cart applets, you must change the variables used for system name, user ID and password to values that work with your AS/400 system.

A basic explanation of the variables follows:

Variables	Description
java.sql.Connection dbConnect	The connection to AS/400
java.sql.PreparedStatement psItem	Optimized Query - see *connect()* for def (Prepared Statements).
java.sql.PreparedStatement psItemRange	Optimized Query - see *connect()* for def.
java.sql.PreparedStatement psCustomerDb	Optimized Query - see *connect()* for def.
java.sql.PreparedStatement psQuantityInHand	Optimized Query - see *connect()* for def.

java.sql.Statement sGetInetOrderNo	Dynamic Query - used to get Internet Order Number. (slower than Prepared Statement, but has more flexibility where SQL can be changed on the fly.)
String systemName	Stores the AS/400 system name to connect to (Internet server name - for example, www.as400.com).
String user ID	Stores the Default User Id for logon from Internet shopping applications.
String password	Stores the password for the Default User Id.
java.sql.ResultSet rs	As a temporary variable for the Query Result Set returned.
String itemId	Store the item Id value of the current record. - **IID** of **ITEM** database in **CSDB** library
String itemName	Store the itemName **INAME** value of the current record.
BigDecimal itemPriceBigDecimal	Store the itemPrice **IPRICE** of the current record in BigDecimal format for caculation.
String itemPrice	itemPrice in String format
String itemInfo	itemInfo - **IDATA** field of **ITEM** database
String validCustomerId	The Valid Customer Id for ordering through Internet.

6.6.1 Common Methods All Applets Use

ItemsDb has some methods that are used by all the GUI applets.

6.6.1.1 Connect()

Connect() is the method that connects to the AS/400 system with the system name, user id, and password defined in the class variables. It also prepares the JDBC statements that are used to access the AS/400 databases.

```
public String connect () {
try{
Class.forName("COM.ibm.as400.access.AS400JDBCDriver");
dbConnect =
      java.sql.DriverManager.getConnection("jdbc:as400://" +
      systemName +
"/csdb;naming=system;errors=full;date
       format=iso",userid,password);
psItem =  dbConnect.prepareStatement("SELECT * FROM
       CSDB/ITEM WHERE IID =  ?");
psItemRange =  dbConnect.prepareStatement("SELECT * FROM
       CSDB/ITEM WHERE IID =  ? AND IID =  ?");
psCustomerDb =  dbConnect.prepareStatement("SELECT CID
       FROM CSDB/CSTMR WHERE CID =  ? AND CDID=001 AND
       CWID='0001'");
psQuantityInHand =  dbConnect.prepareStatement("SELECT
       STQTY FROM CSDB/STOCK WHERE STWID= '0001' AND STIID=?");
} catch (Exception e) {
System.out.println("connect(): "+e);
e.printStackTrace();
return "Connect: "+e;}
return "Connect Successfully";
}
```

6.6.1.2 Disconnect()

Disconnect() is the method for closing all AS/400 connections.

.

```
public void disconnect () throws Exception {
dbConnect.close();
psItem.close();
psItemRange.close();
psCustomerDb.close();
psQuantityInHand.close();
return;
}
```

6.6.1.3 Finalize()

Finalize() automates the *disconnect()*, so that every time the *ItemsDb* class is disposed, it disconnects from the AS/400 system and release all resources allocated.

.

```
protected void finalize() {
try { disconnect();super.finalize(); } catch(Throwable t)
       {System.out.println(t);}
}
```

6.6.2 Methods Used by ToolboxApplet

ItemsDb has some methods that are used only by the ToolboxApplet.

6.6.2.1 FetchNextItem()

FetchNextItem() fetches the next record from the current result set and puts the corresponding field values to the public class variables. It returns itself (*this*), for cascading of methods.

.

```
public ItemsDb fetchNextItem () {
try {
if (rs.next()) {
itemId=rs.getString("IID");
itemName=rs.getString("INAME");
itemPriceBigDecimal=rs.getBigDecimal("IPRICE",2);
itemPrice=itemPriceBigDecimal.toString();
itemInfo=rs.getString("IDATA");
}
else {
itemId=null;
itemName=null;
itemPriceBigDecimal=null;
itemPrice=null;
itemInfo=null;
}
} catch (Exception e) {System.out.println("fetchnext
        fail: "+e);}
return this;
}
```

6.6.2.2 GetItem()

GetItem() querys item information from the *ITEM* database with the *itemno*, and uses *fetchNextItem()* to load the data into the class variables.

```
public ItemsDb getItem (String itemno) {
try {
psItem.setString(1, itemno);
rs =  psItem.executeQuery();
fetchNextItem();
} catch (Exception e) {System.out.println("getItem fail:
        "+e);}
return this;
}
```

6.6.2.3 GetItems()

GetItems() querys item information from the *ITEM* database with the range of *itemno*, and users use *fetchNextItem()* to load the data into the class variables. If there are no more records, the class variables are set to null.

```
public ItemsDb getItems (String itemnoMin, String
        itemnoMax) {
if(itemnoMax.length()==0) {
getItem(itemnoMin);
}
else {
try {
psItemRange.setString(1, itemnoMin);
psItemRange.setString(2, itemnoMax);
rs =  psItemRange.executeQuery();
} catch (Exception e) {System.out.println("getItemS fail:
        "+e);}
}
return this;
}
```

6.6.3 Methods Used by CartApplet

ItemsDb provides some methods that are only used by the CartApplet.

6.6.3.1 QuantityInHand()

QuantityInHand() returns the Quantity in Stock of a particular item with item number *itemNo*.

```
public BIgDecimal quantityInHand (String itemNo){
try{
// Get next Order No. and the Inet YTD Balance
psQuantityInHand.setString(1,itemNo);
rs =  psQuantityInHand.executeQuery();
rs.next();
return rs.getBigDecimal("STQTY",0);
} catch(Exception e) {System.out.printIn(e)}
return null;
}
```

6.6.3.2 VerifyCustomer()

VerifyCustomer() checks whether the customerId provided is a valid one, and returns true or false depending on the result of the check. If the customerId is valid, it saves it in the class variable *validCustomerId*.

```
public boolean verifyCustomer (String customerId) {
boolean isvalid=false;
try {
psCustomerDb.setString(1, customerId);
rs =  psCustomerDb.executeQuery();
if(rs.next()) {isvalid=true; validCustomerId=customerId;}
} catch (Exception e) { validCustomerId=null;}
return isvalid;
}
```

6.6.3.3 ConfirmOrder()

The *confirmOrder()* method creates an order from the items inside the *SelectedItems* class. The customer Id should be validated by the *verifyCustomer()* method before hand.

When confirming the order, it gets the next order number **DNXTOR** from the district database **DSTRCT**, increments it by one, and writes it back to the database. In a real life application, locking records, optimistic record locking, or stored procedures should be considered to provide database integrity. See Appendix D, “Internet Shopping Applet Code Listings” on page 245 for a complete listing of the **confirmOrder** method.

The **confirmOrder** method uses JDBC to access the AS/400 databases. It contains the following logic:

- Checks to see is there are items in the cart.
- Retrieves the next order number and district YTD balance from the AS/400 district file.
- Updates the district YTD balance with the new order total.
- Increments the district next order number by one.
- Inserts an order record to the AS/400 order file.
- Updates the stock balance in the AS/400 stock file.
- Inserts an order line record in the AS/400 order line file for each item ordered.
- Returns the order number used.
- Clears out the items from the cart.

6.6.4 Methods Used by the StatusApplet

ItemsDb provides some methods that are used only by the StatusApplet.

6.6.4.1 CheckOrderStatus()

The **checkOrderStatus()** method takes in an order ID *orderIdString* and then does all the queries and finally returns a Vector *orderStatus* containing all the details about the order. Please refer to Appendix D, “Internet Shopping Applet Code Listings” on page 245 for a complete listing of this method.

```
public Vector checkOrderStatus (String orderIdString) {
Vector orderStatus=new Vector();
if(orderIdString.length()>9 || orderIdString.length()==0) return null
try{
  sGetInetOrderNo =  dbConnect.createStatement();
  rs=sGetInetOrderNo.executeQuery("SELECT OCID,OLINES FROM CSDB/ORDER
   rs.next();
   String customerId=rs.getString("OCID");
   BigDecimal orderLines=rs.getBigDecimal("OLINES",0);
   rs=sGetInetOrderNo.executeQuery("SELECT CFIRST,CLAST FROM CSDB/CSTMR
   rs.next();
String lastName=rs.getString("CLAST");
String firstName=rs.getString("CFIRST");
orderStatus.addElement(lastName);
orderStatus.addElement(firstName);.
.
.
return orderStatus;
```

6.7 The "ToolboxApplet" Applet

This applet connects to the AS/400 system when it is being initialized. Customers can press the *"Query Range of Items"* button to get the items where the range is specified by TextField 1 and 2. The item information is displayed in the list box. It uses the *ItemsDb* class that does all the AS/400 Database accesses. It also uses the *MyListbox* class from Taligent for displaying the query result.

6.7.1.1 Basic Class Definition

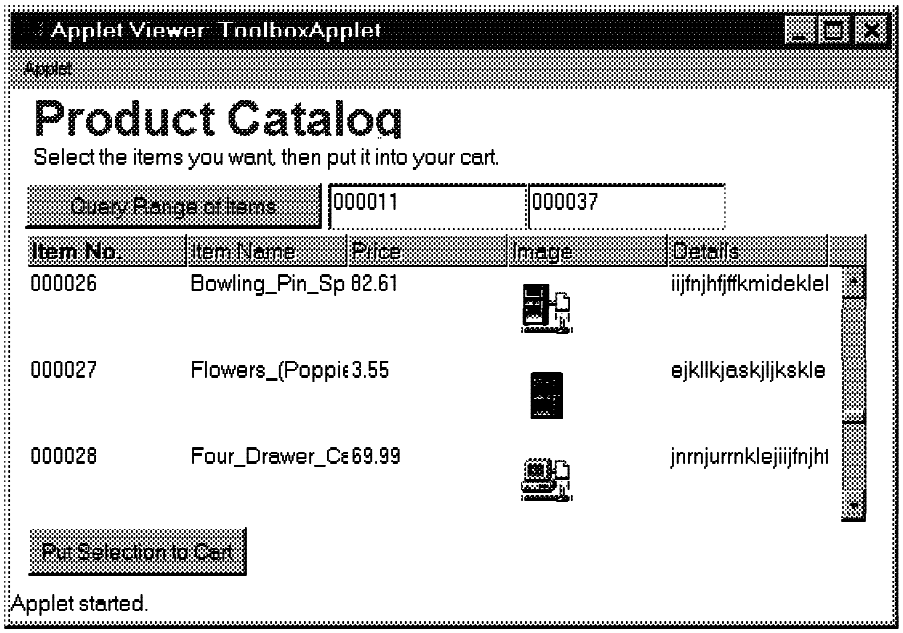
We show the basic class structure, class variables, and the GUI part. The class structure was generated by using the Applet Wizard in the toolbar of the Workbench.

```
import com.taligent.util.*;
import com.taligent.widget.*;
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.net.URL;
import java.util.*;
public class ToolboxApplet extends java.applet.Applet
        implements java.awt.event.ActionListener {
static ToolboxApplet.SelectedItems selected =  new
   SelectedItems();
private String imagePath =  "file:/C:/mytest/solution/";
//... ...( the above image path is where you find the
     product image, 000001.GIF, ...)
//... ...( which may change according to setup. It also
  accept http://www....)
```

We define the SelectedItems class and instantiate it. Because it uses static variables, they are, therefore, shared by all classes that instantiated it.

We also define the imagePath for loading the product GIF images.

6.7.1.2 Building the Visual Parts:



Labels - "Product Catalog", "Select the items...", and "Message:" are text labels. You can also change their font and color setting by double-clicking them.

Buttons - the "Query Range of Items" and "Put Selection to Cart" buttons; you can change the button text by ALT-Clicking on them.

Text Fields - the two entry fields allow the customer to query a range of items to be displayed in the list box.

List box - used to display the items returned by the query.

6.7.2 MyInit()

The MyInit method initializes the list box by adding column names, setting column widths, and setting row heights.

```
/**
* This method was created by a SmartGuide.
*/
public void MyInit () {
String columns[] =  {"Item No.","Item
        Name","Price","Image","Details"};
getListbox().addColumns(columns);
getListbox().getColumnInfo(0).setWidth(100);
getListbox().getColumnInfo(1).setWidth(250);
getListbox().getColumnInfo(2).setWidth(100);
getListbox().getColumnInfo(3).setWidth(100);
getListbox().getColumnInfo(4).setWidth(100);
getListbox().setMultipleSelections(true);
getListbox().setPreferedRowHeight(50);
getListbox().reshape();
}
```

For this method, getListbox() returns the object we built in the Visual Editor named "Listbox". We use getColumnInfo() to get the Column references so we can set the column's width. We also set other attributes of the list box such as MultipleSelections and RowHeight. Reshape() refreshes the list box.

6.7.3 AddAllRows()

The **addListboxRow()** method is used by the **addAllRows()** method to populate the list box.

```
/**
public void addListboxRow () {
Object myObject[]=new Object[5];
myObject[0]=getItemsDb().itemId;
myObject[1]=getItemsDb().itemName;
myObject[2]=getItemsDb().itemPrice;
// URL baseURL=getCodeBase();
String imageName=getItemsDb().itemId+".GIF";
try {URL baseURL=new URL(imagePath);
myObject[3]=new
      MyImage(getImage(baseURL,imageName),getListbox());}
catch(Exception e){ myObject[3]="Not Loaded";
myObject[4]=getItemsDb().itemInfo;
getListbox().addRow(myObject);
return; }
myObject[4]=getItemsDb().itemInfo;
getListbox().addRow(myObject);
}
```

We get the "Item id", "Item Name", "Item Price" from ItemsDb by using *getItemsDb().PublicVariables*. Look back into the ItemsDb Class Definition to see what public variables are available.

We also get an image from the *imagePath* defined in the class definition. *MyImage(.Image,Listbox)* loads the image and puts it into the list box automatically. You can look into the coding of MyImage Class in the Appendix to learn how to load an image URL. (It is inside the Package itsc.taligent.widget.)

6.7.4 AddAllRows()

```
/**
* This method was created by a SmartGuide.
*/
public void addAllRows () {
while(getItemsDb().fetchNextItem().itemId!=null) {
addListboxRow();
getListbox().repaint();
}
return;
}
```

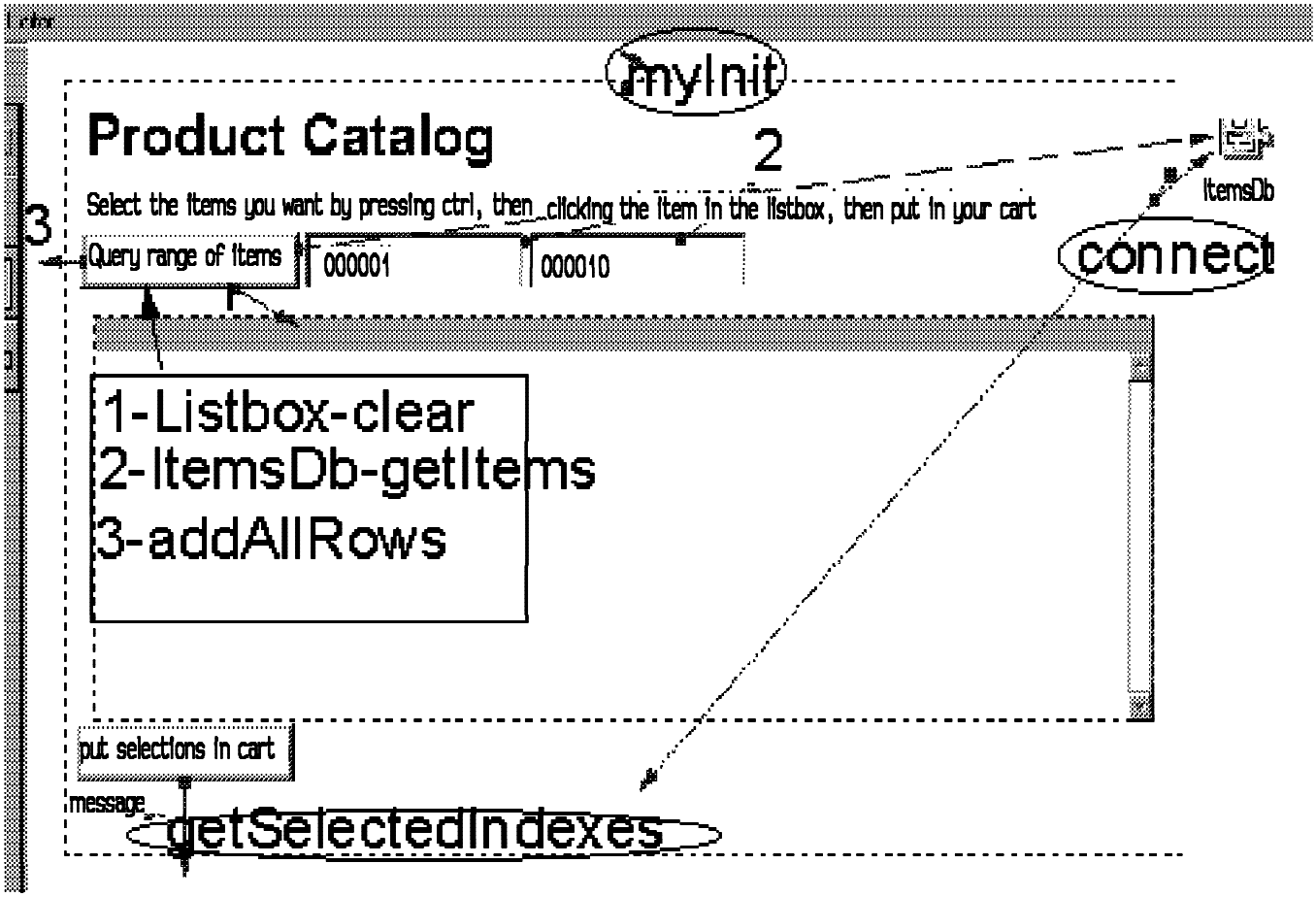
The addAllRows() method loops and gets all the items from the resultset. It calls **addListboxRow** to build the list box and finally it repaints the list box.

6.7.5 GetSelectedIndexes()

The **getSelectedIndexes()** method determines which list box indexes are selected. It then adds the selected rows of the list box to the SelectedItems class named **selected**. In other words, it puts the selected items in the "cart".

```
public void getSelectedIndexes() {
int[] indexes=getListbox().getSelectedIndexes();
for(int i=0;iindexes.length;i++) {
selected.addSelectedRow(getListbox().getRow(indexes[i]));
}
}
```

6.7.6 Checking the Connections



If you are viewing the applet in the Visual Builder, you can check how we link the user interface to our class and methods.

We use the following connections in this applet:

- Initialization event of the ToolboxApplet:
  - MyInit - to initialize the list box

- Start event of the ToolboxApplet:
  - ItemsDb - connect method
  - Connect the normal result to the message label.
- Query range of items button:
  - Clear the list box.
  - ItemsDb - getItems
  - AddAllRows
- Put selections in cart button:
  - GetSelectedIndexes

6.8 The "CartApplet" Applet

Next, we view the *"CartApplet"* that shows the selected items. Also, we have to allow the customer to confirm the selections and write the order to the order database.

In the design of the *"CartApplet"*, we implement a button "Look into Cart" to refresh the shopping cart. We could implement an Event for the automatic refresh of the cart whenever an item is put into the cart. But for simplicity, we choose to implement the "Look into Cart" button.

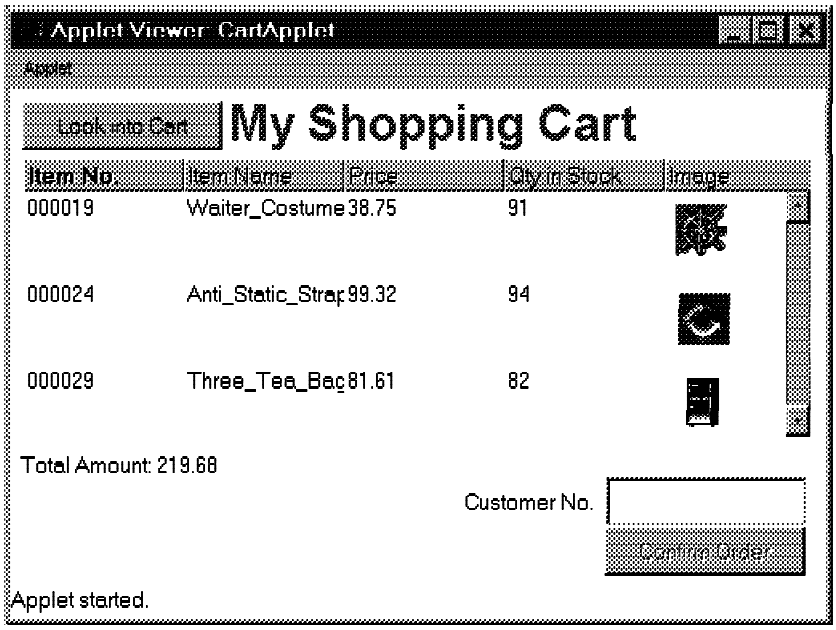
6.8.1 Writing the Class

Basic Class Definition:

```
import java.applet.*;

import java.awt.*;

import java.util.*;

 public class CartApplet extends java.applet.Applet
   implements java.awt.event.ActionListener,
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
ToolboxApplet.SelectedItems cart =  new SelectedItems();
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
}
```

In the class definition, we create a *SelectedItems* object and named it *cart*. Actually, this is shared with *the ToolboxApplet. The ToolboxApplet* puts items in the cart and here we make the order from the items in the cart.

The visual parts:



Labels - "My Shopping Cart", "Total Amount:", and "Customer No." are text labels. You can change their font and color setting by double-clicking them.

Buttons - the "Look into Cart" and "Confirm Order" buttons. You can change the button text by ALT-Clicking them.

Text Fields - the Entry Field for entering customer number.

List box - for displaying selected items.

ItemsDb visual part - for accessing the databases.

MessageBox visual part - for displaying messages.

6.8.2 Viewing the Methods

First, we view the MyInit() for the list box.

And this time, we want the following columns in it:

"Item No.", "Item Name", "Price", "Qty in Stock", "Image", and "Details".

```
public void MyInit () {
// To initialize the MultiColumnListBox
String columns[] =  {"Item No.","Item Name","Price","Qty
        in Stock","Image","Details"};
getListbox().addColumns(columns);
getListbox().getColumnInfo(0).setWidth(100);
getListbox().getColumnInfo(1).setWidth(250);
getListbox().getColumnInfo(2).setWidth(100);
getListbox().getColumnInfo(3).setWidth(100);
getListbox().getColumnInfo(4).setWidth(100);
getListbox().getColumnInfo(5).setWidth(100);
getListbox().setPreferedRowHeight(50);
getListbox().reshape();
}
```

Next, we view the *showCart()* method.

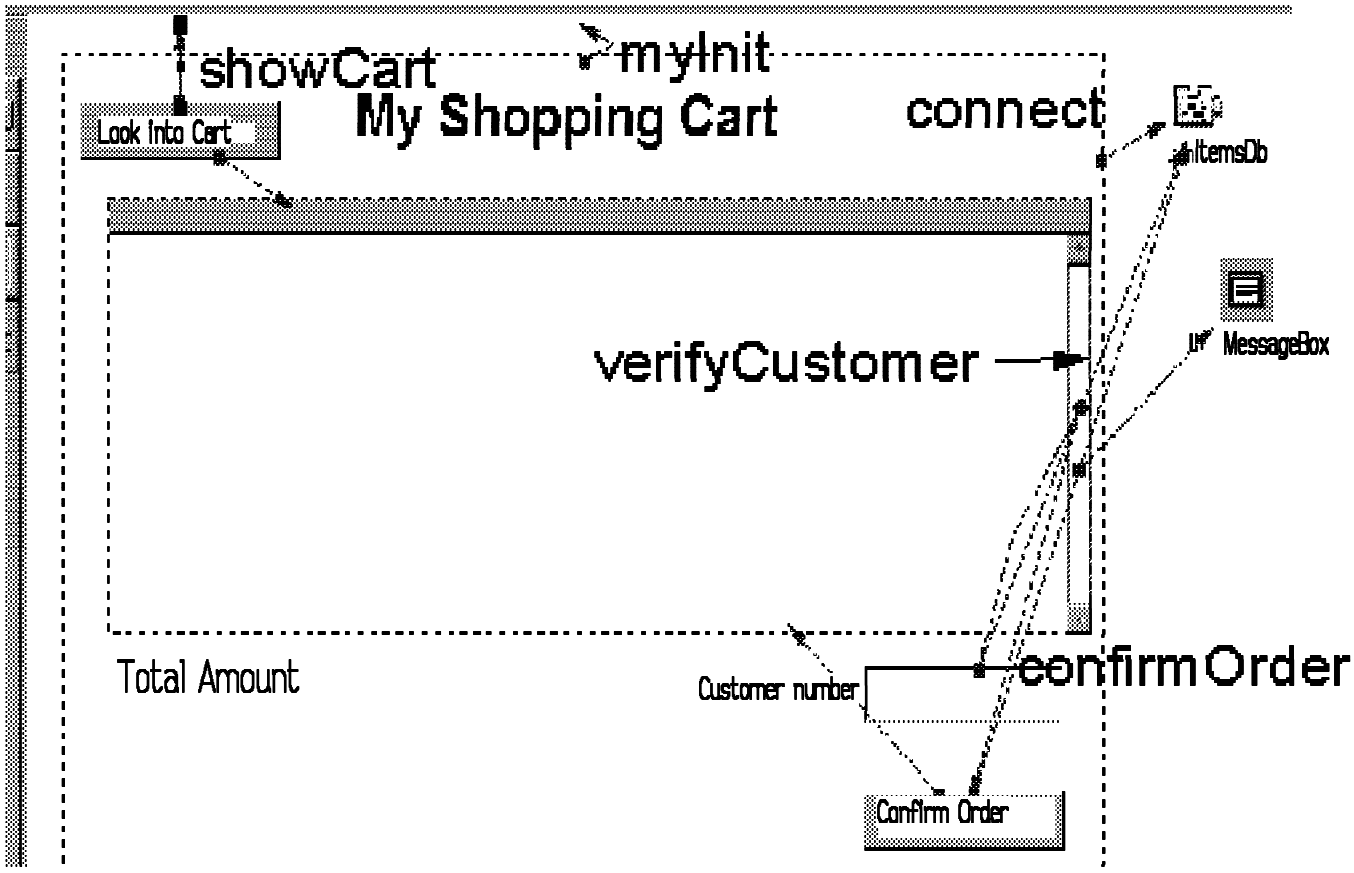
We use an object array of 6 (that is, Object[6]) to store the elements from the vector where there are five object (columns) in it. We insert the quantity in stock of the item to the 4th element (that is, Object[3]) where the quantity in stock can be obtained by calling a method in ItemsDb. We change it to String for display.

We then add the fourth and fifth element of the vector to the fifth and sixth object of our array.

Finally, we add our object array as a row to the list box, display the total amount in our label where we assume it is Label3, and then have the list box repaint itself.

```
public void showCart () {
Object myObject[]=new Object[6];
try {
if(cart.getVector()!=null) {
Enumeration enum=cart.getVector().elements();
while(enum.hasMoreElements()) { Object[] element=((Object[])enum.nextElement());
myObject[0]=element[0]; //ItemId
myObject[1]=element[1]; //ItemNam
myObject[2]=element[2]; //Price
//[4] is qty in stock.
myObject[3]=(getItemsDb().quantityInHand(((String)eleent[0]))).toString();
myObject[4]=element[3]; // Image
myObject[5]=element[4]; //Details
getListbox().addRow(myObject);
};
getLabel3().setText("Total Amount:
    "+cart.totalAmount.toString());
getListbox().repaint();
return;
}
} catch(Exception e)
   {e.printStackTrace();System.out.println(e); }
return;
}
```

6.8.2.1 Viewing the Connections



- *Init Event of the Applet*:

  The Init Event of the Applet is connected to *"*Event to Script*"* *MyInit()* method. The Init Event of the Applet is also connected to the *ItemsDb* visual part's *connect()* method.

- *Look into Cart button*:

  The *"Look into Cart"* button is first connected to the *list box* visual part's *clear()* method. It is then connected to the "Event to Script" *showCart()* method.

- *Validate Customer Number*:

  We validate the Customer number field so that the *"confirm Order"* button is only enabled if the customer number is entered correctly. We have built a *verifyCustomer()* method in *ItemsDb* to use. We connect it with the *KeyReleased event* of the *Customer Number Entry Field* and the *Confirm Order button*. It enables the *Confirm Order button* if the *Customer Number* in Entry Field is found in Database.

- *Confrim Order button*:

  We link the *"confirm Order"* button to the *ItemsDb* Part *confirmOrder()* method, with *"cart"* as the input parameters (Hint: set it using **Open Setting** and then **Set Parameters**). And the output should be sent to the MessageBox's *show()* method.

Finally, we connect the *"confirm Order"* button to *list box* visual part's *clear()* method, so that after confirming the order, items in the list box are cleared.

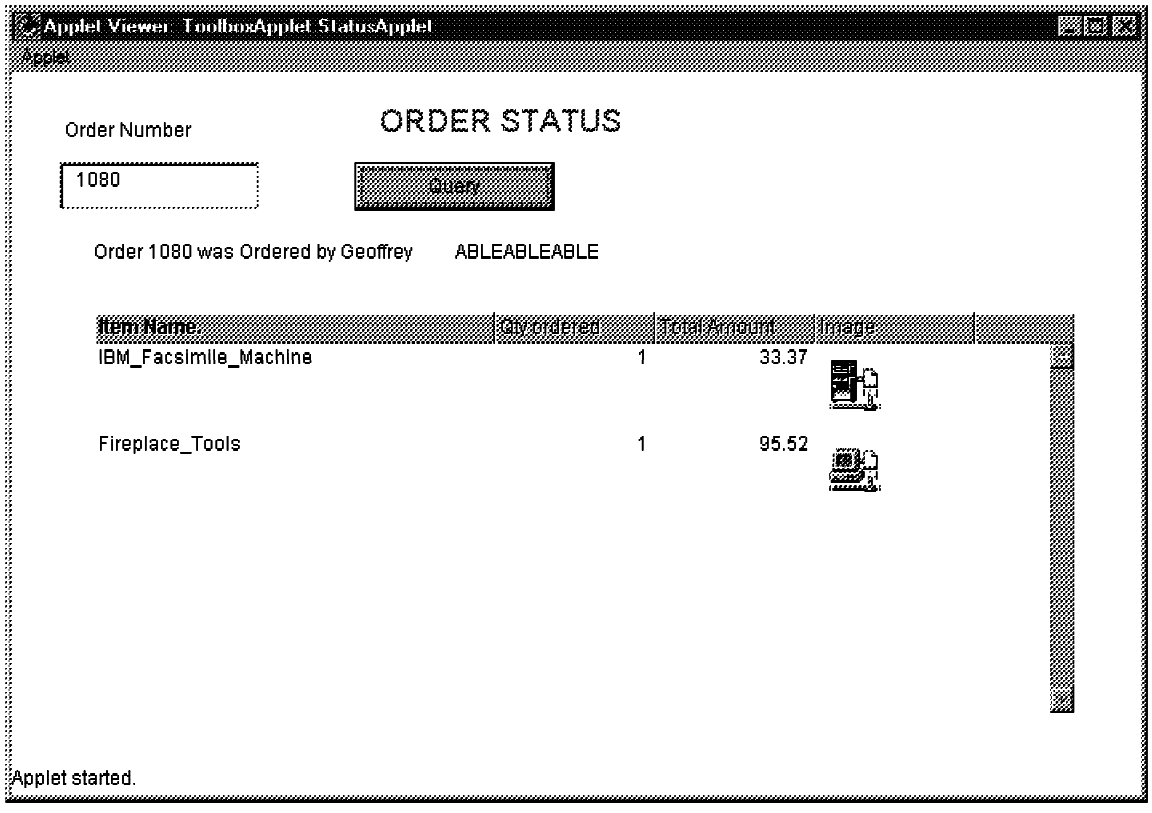
6.9 The Check Order Status Applet

In the Class definition, we need:

```
import itsc.taligent.widget.*;
import java.applet.*;
import java.awt.*;
import java.net.URL;
import java.util.*;
public class StatusApplet extends java.applet.Applet {
private java.lang.String imagePath =
        "file:/C:/mytest/solution/";
//... ...( the above image path is where you find the
        product image, 000001.GIF, ...)
//... ...( which may change according to setup. It also
        accept http://www.... )
```

The Visual parts:

.



Labels - "ORDER STATUS", and "Order Number" text labels; you can change their font and color setting by double-clicking them..

Button - the "Query" button; you can change the button text by ALT-Clicking on it.

Text Fields - the Entry Field for entering Order number.

List box - for displaying the items.

ItemsDb visual part - for database access.

To implement the Check Order Status Applet, two user-written methods are used.

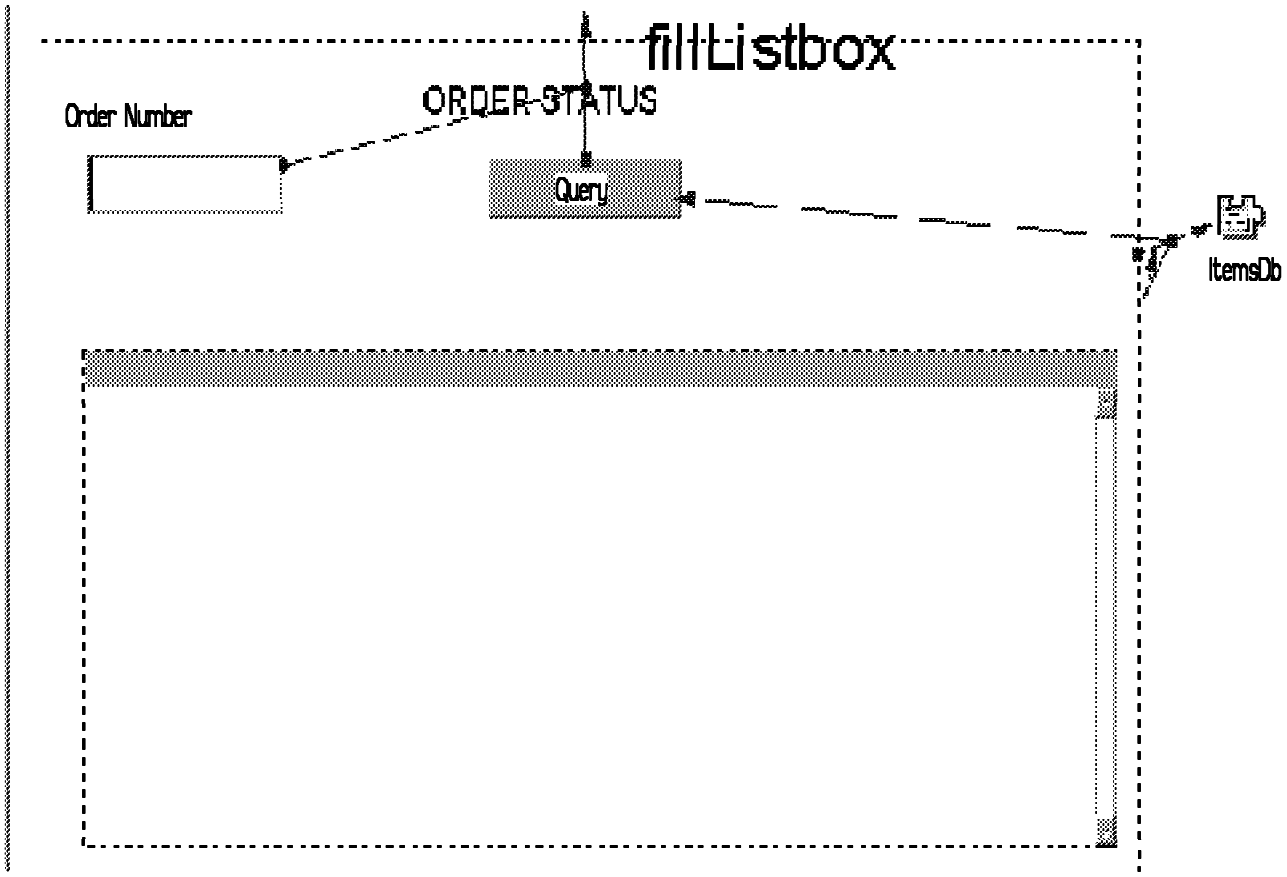
- *fillListbox(OrderId String)* that calls the getItemsDb().checkOrderStatus() method with OrderId as the parameter. If the order is found, it puts the items ordered in the list box to display. The checkOrderStatus method returns a vector that contains lastname, firstname, and an array of order detail. See Appendix D, “Internet Shopping Applet Code Listings” on page 245 for a complete listing of this method.

```
public void fillListbox (String orderId) {
Vector orderStatus=getItemsDb().checkOrderStatus(orderId);
if(orderStatus==null) {
getLabel2().setText("Order No. "+orderId+" Not Found !!!");
return; }
Enumeration  detailLine=orderStatus.elements();
String lastName=((String)detailLine.nextElement());
String firstName=((String)detailLine.nextElement());
getLabel2().setText("Order "+orderId+" was Ordered by "+firstName+" "+la

while(detailLine.hasMoreElements()) {
  Object[] detail= ((Object[])detailLine.nextElement());

  String imageString=((String)detail[3]);
try {
  URL baseURL=new URL(imagePath);
  detail[3]=new MyImage(getImage(baseURL,imageString),getListbox());
} catch (Exception e) {detail[3]="Not Loaded";}
.
.
.
```

- *MyInit()*, which initializes the MyListbox Class with proper Columns, headings, and sizes.

```
public void MyInit () {
String columns[] =  {"Item Name","Qty ordered","Total
          Amount","Image"};
getListbox().addColumns(columns);
getListbox().getColumnInfo(0).setWidth(250);
getListbox().getColumnInfo(1).setWidth(100).setAlignment(ListboxColumn.RIGHT);
getListbox().getColumnInfo(2).setWidth(100).setAlignment(ListboxColumn.RIGHT);
getListbox().getColumnInfo(3).setWidth(100);
getListbox().setPreferedRowHeight(50);
getListbox().reshape();
return;
}
```

6.9.1.1 Viewing the Connections



- *Init Event of the Applet*:

  The Init Event of the Applet is connected to *"*Event to Script*"* *MyInit()* method. The

  Init Event of the applet is also connected to *the ItemsDb* visual part's *connect()* method.

- *Query button*:

The *"Query"* button is connected to the *list box* visual part's *clear()* method. It is also connected to the "Event to Script" *fillListbox()* method. Finally, the Text Entry Field's *text* property is connected to the previous connection's *orderId* parameter.

Note: The most important thing here is the format of the Vector returned from *checkOrderStatus()*, and how you organize it and put it into the Mylistbox class. You need to check out and understand how to use Vector, Vector enumeration, elements(), and hasMoreElements() of the Vector classes in the java.util package.

6.9.2 Using JAR Files

In this section, we discuss using the JAR support of JDK 1.1 to compress and package Applets.

You can simply run "JAR" without any parameter to see the documentation.

One thing to remember is that Java is a case-sensitive language, so when typing in a package name such as "ToolboxApplet", "itsc", and "COM", capitalization should be followed strictly. Or else browsers that try to open classes in the jar file throw a ClassNotFound exception.

jar cvf applets.jar ToolboxApplet itsc COM	This compresses all files under the directory "ToolboxApplet", "itsc", and "COM".
jar tvf applets.jar	This displays what is inside "applets.jar".

Looking at what is being put into our "applets.jar", you learn that some of the classes are actually unnecessary they are there because they are in the directory. JAR works the same as a ZIP or other compression utilities in that it compresses everything; it does not determine what is needed and what is not. So it is your responsibility to JAR only those items needed (ONLY Class files and GIF files you have used).

jar xvf applets.jar	This extracts all files according to the original directory information. In addition, it also extracts the manifest file and appends the content to the "*META-INF\MANIFEST.MF*" file

MANIFEST.MF file is for other JDK 1.1 compliance tools or browsers to find out what is inside of a JAR file.

If needed, you can provide your own *MANIFEST.MF* file during compression as shown in the following example.

```
jar cvfm applets.jar meta-inf/manifest.mf   ToolboxApplet COM Applets in JAR
```

When an Applet is in a JAR file, the HTML code is shown in the following example:

```
<HTML>
<applet code="ToolboxApplet.ToolboxApplet"
width=500 height=400 archive="applets.jar">
<hr>
This Applet would only be seen
on JDK1.1 compatible Browser
<hr>
</applet><p>
```

You can try to open the preceding homepage with the HotJava 1.0 browser, which supports JDK1.1. You have to set the HotJava Preference to medium security for unsigned applet so that you can connect from a PC to the AS/400 system.

Chapter 7. JavaBeans

This chapter is designed to help you understand what a JavaBean (also referred to as a bean throughout) is and how they are created. We first discuss what a JavaBean is made of and go on to discuss why JavaBeans are so appealing. You then learn what makes a good JavaBean and when to use them. Finally, I'll guide you step-by-step to create a simple JavaBean.

Java code snippets are scattered throughout followed by several complete working examples that you can compile and try on your own.

What Do I Need and 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 a 1.1 Java Development Kit and a text editor.

You do not need to be a Java expert to understand this chapter. Although there are some portions that require some basic Java and object oriented skills, there is still a lot that can be learned even for someone with no Java knowledge.

This chapter is not meant to teach every last detail about JavaBeans, but provides enough information for you to understand and create simple-to-intermediate JavaBeans. We also tell you what JavaBeans are capable of and what additional information to gather for your specific needs.

7.1 What Do JavaBeans Offer?

Before you spend days reading documentation and tutorials, we discuss some of the benefits of using JavaBeans.

Reuse, Reuse, Reuse

Probably the largest benefit gained from using beans is the ability to use a bean over and over because a bean 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 used in so many applications that the time saved by programmers who can use the standard Java button rather than create their own is unimaginable. But this does not only apply to graphical components. A bean can be something as complex as a grammar and spelling checker, but can also be reused because of the large number of word processing applications.

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 begin with, but many beans can be bought and reused and many builder tools actually allow 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 than to write everything yourself and wind up with several syntax errors.

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 thing to remember is that JavaBeans are 100% Java and simply an extension of Java, so JavaBeans offers everything Java offers and more. The data processing industry has seen incredible growth and support for Java in the past couple of years and there seems to be no stopping point yet, so as with Java, JavaBeans is not expected to die any time 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/intranet ready and are perfect for distributed applications. They also inherit all of the built-in security Java offers.

Easy Packaging and Distributing

Java JAR files make it easy to package up 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 to use. Because of introspection and the beans 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 a code to determine a method's parameters.

7.2 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 leaving 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, we first discuss some basic JavaBeans concepts and terminology used. We then learn what is done to create a simple JavaBean without any complex BeanInfo knowledge (found in 7.3, “Creating a Simple JavaBean” on page 192.

What Does it Actually Mean 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 the few simple rules and naming conventions. There is no class that a bean must extend or interfaces that must be implemented although some are offered to help with complex beans.

Another important thing to remember is 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, but 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 that is general enough to be used in many applications, but not all are graphical. Most builder tools allow the user to have an icon that represents the bean while designing the application, which is invisible at run time.

JavaBean Terminology

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 way to change it. Another property is hair color. This is a readable/writable property because we can change their hair color.

The JavaBean specification suggests a naming convention to be used when creating properties. First, a property should be private or protected and 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. The following Java code sample is for an age property.

```
private int age;
public int getAge() {
 return age;
}
public void setAge(int newAge) {
 age = newAge;
}
```

Method:
A method for a bean is nothing different from any regular Java method. It is simply an interface by which beans can communicate by passing parameters and getting values or objects back.

There are a couple things 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 (Advanced JavaBeans concepts), all public methods are displayed using Introspection, so 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 in to 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 the appropriate actions.

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 analyses the bean first to determine what properties, methods, and events the bean has.

Customization:
Customization is exactly as it is 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 so new development does not have to take place. The JavaBeans specification gives us two ways to do this. First, property editors can be created

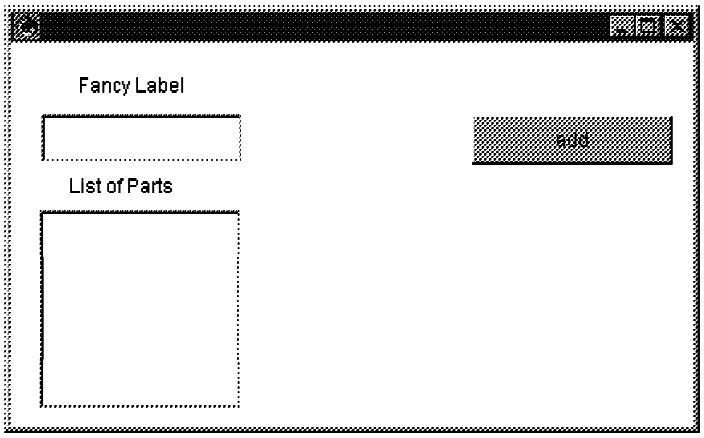
to make changing a bean property easier and more robust by checking to make sure a property's value is being set within a valid range, and so on. Second, a Customizer class can be created, which can be a wizard to take a developer step-by-step through using the bean, so say good-bye to manuals because documentation can be shipped right within 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.

7.3 Creating a Simple JavaBean

In this section we create a simple bean named **FancyLabel**. It demonstrates how we can externalize methods and properties. The function of this bean is to allow 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 Label** is shown in red when the mouse passes over it and green when the mouse is not over it.



We have three properties that we make available for others to use:

- mouseInsideColor
- mouseOutsideColor
- mouseInside

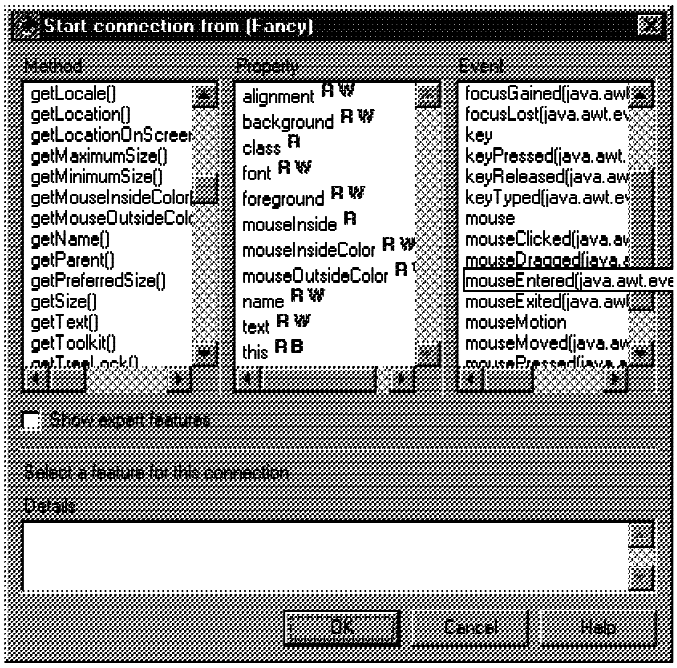
We only implement two methods:

- mouseEntered
- mouseExited

```
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;
public FancyLabel() {
 super();}
public FancyLabel(String text) {
 super(text);}
public FancyLabel(String text, int alignment) {
super(text, alignment);}
 // Property Get and Set methods
public java.awt.Color getMouseInsideColor() {
  return mouseInsideColor;}
public java.awt.Color getMouseOutsideColor() {
  return mouseOutsideColor;}
public boolean isMouseInside() {
  return mouseInside;}
public void setMouseInsideColor(java.awt.Color newInsideColor) {
 mouseInsideColor =  newInsideColor;
 repaint();}
public void setMouseOutsideColor(java.awt.Color newOutsideColor) {
 mouseOutsideColor =  newOutsideColor;
 repaint();}}
```

```
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);}
```

Because our label is a bean, we can use a tool such as VisualAge for Java to display its methods, properties and events.



To make the bean work with our application, we make two connections:

- Event **mouseEntered** to the mouseEntered method
- Event **mouseExited** to the mouseExited method

7.4 Making ItemsDb a JavaBean

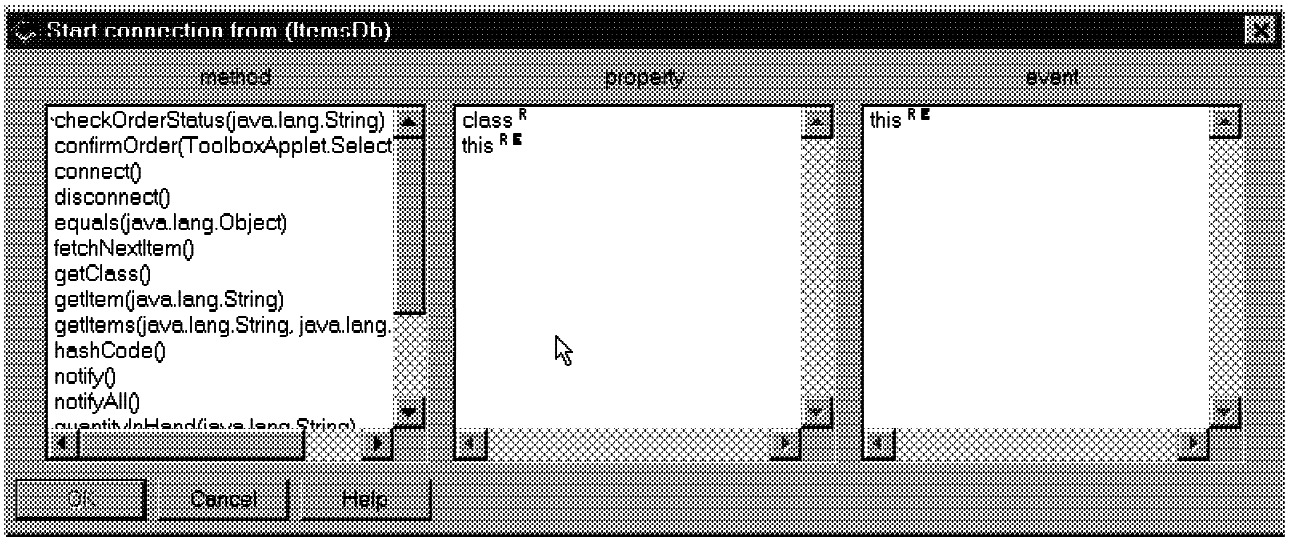
In this section of the chapter, we show a more practical use of JavaBeans. We change the ItemsDb class that we used in the shopping cart applet in Chapter 6, “Developing AS/400 Java Applets” on page 155 into a bean.

To make "ItemsDb" a simple bean, we publicize the methods, properties, and Events of "ItemsDb". Actually, we do not need to do anything with all the public methods; they are automatically seen.

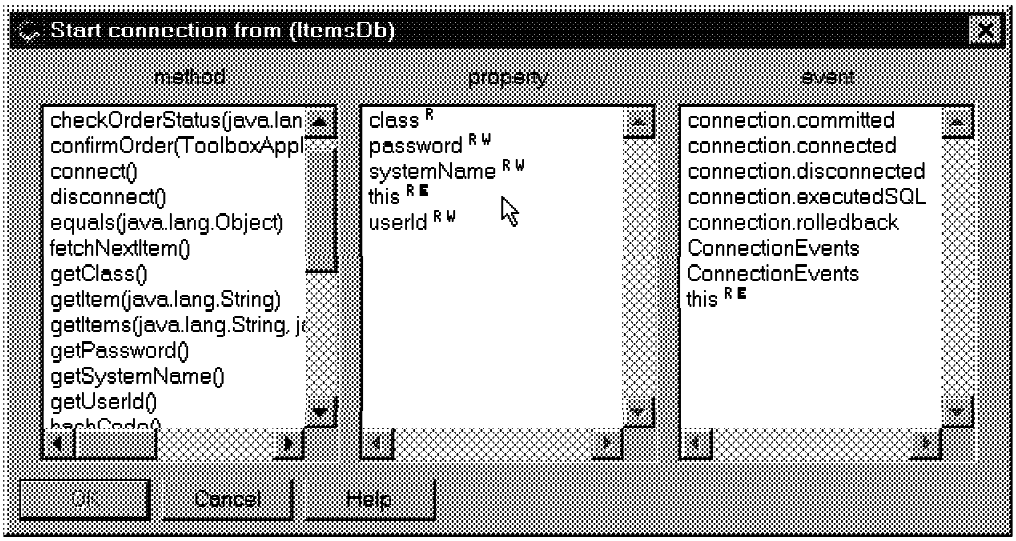
We only need to set up the properties that are the values of "systemName", "userid", and "password", and the Event of "ItemsDb", which is the new "Connected" event.

7.4.1 Review of Current "ItemsDb"

We first review the current "ItemsDb" class. We open the "ToolboxApplet" Applet and find the "ItemsDb" in the Visual Builder. We choose "connect" and then "All features" by right-clicking it.



In the preceding window, notice that ItemsDb has no properties or events.



When we finish this section, you can see the preceding window with the new properties and events.

7.4.1.1 Adding Properties

We add the properties for "systemName", "userId", and "password" to "ItemsDb".

We open the "ItemsDb" class and create the getter and setter methods for the properties that we want to externalize.

We add the new method getSystemName() to it as follows:

```
public String getSystemName () {
return systemName;
}
```

Add the new method setSystemName() as follows:

```
public void setSystemName(String tSysname) {
systemName=tSysname;
return;
}
```

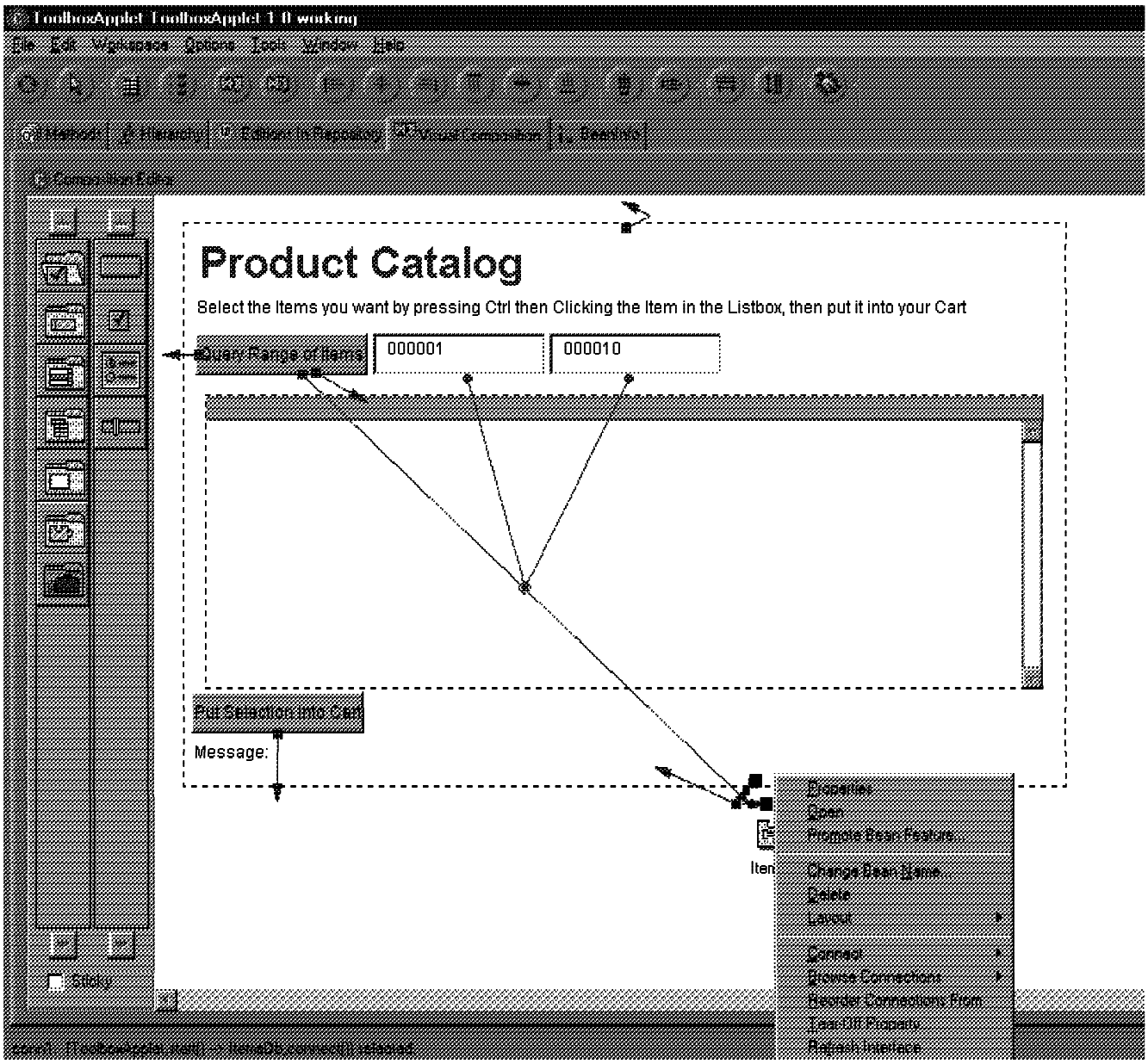
Notice that "systemName" is the variable where we store the AS/400 server name.

We next add getter and setter methods for userId and password.

- getPassword()
- setPassword()
- getUserId()
- setUserId()

Note: For the bean to recognize a property, the only thing you need to do is add two new public methods, the get*Property*() and set*Property*(). The *property* can be any name you want, not necessarily equal to the variable name.

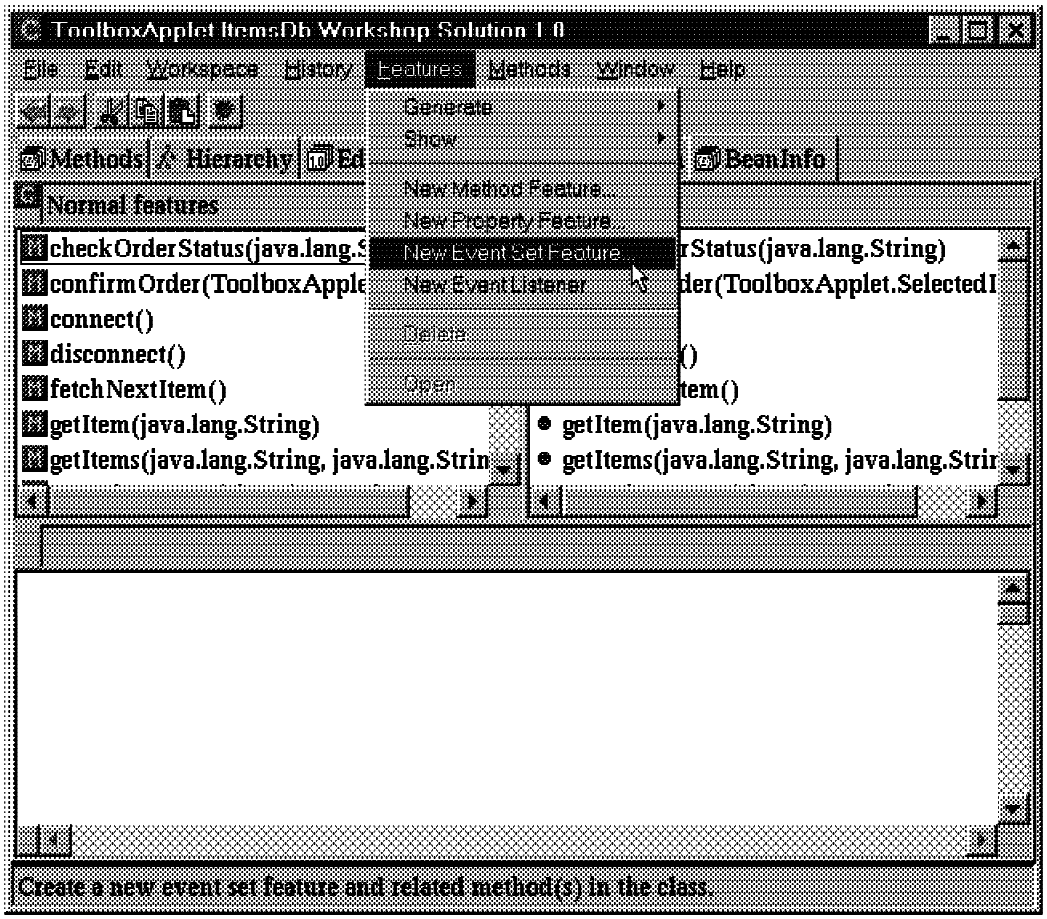
After saving all of the methods, we open the "ToolboxApplet" visual builder.



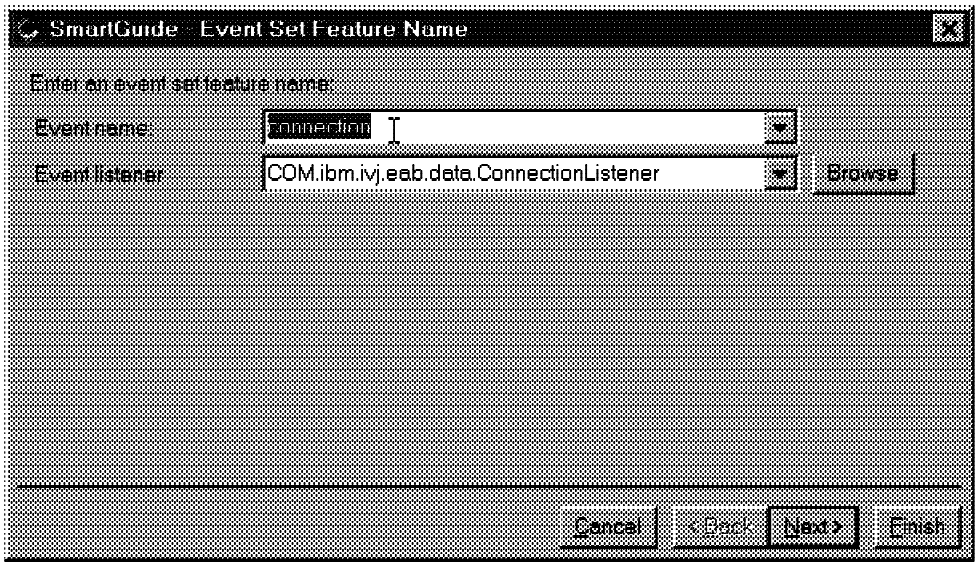
We right-click on "ItemsDb" and select "Refresh Interface" to get the new properties into the Visual Builder. You can check the properties by right-clicking the "ItemsDb", select "Connect", and then "All Features". Also, you can now set or change the property by double-clicking "ItemsDb" ("properties").

7.4.1.2 Adding Events

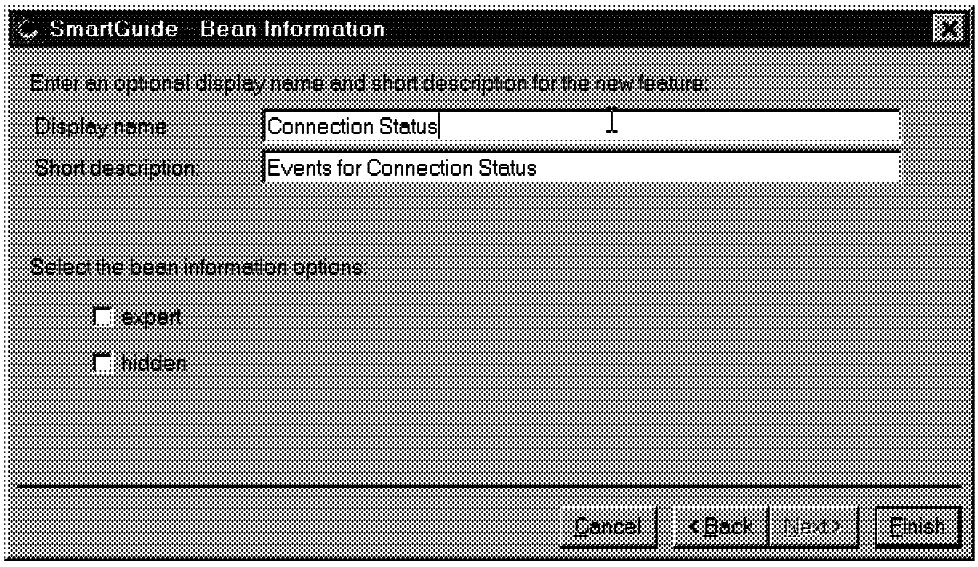
To finish this bean, we add all of the events to the "ItemsDb"class. We open the "ItemsDb" class again and click the "BeanInfo" tab.



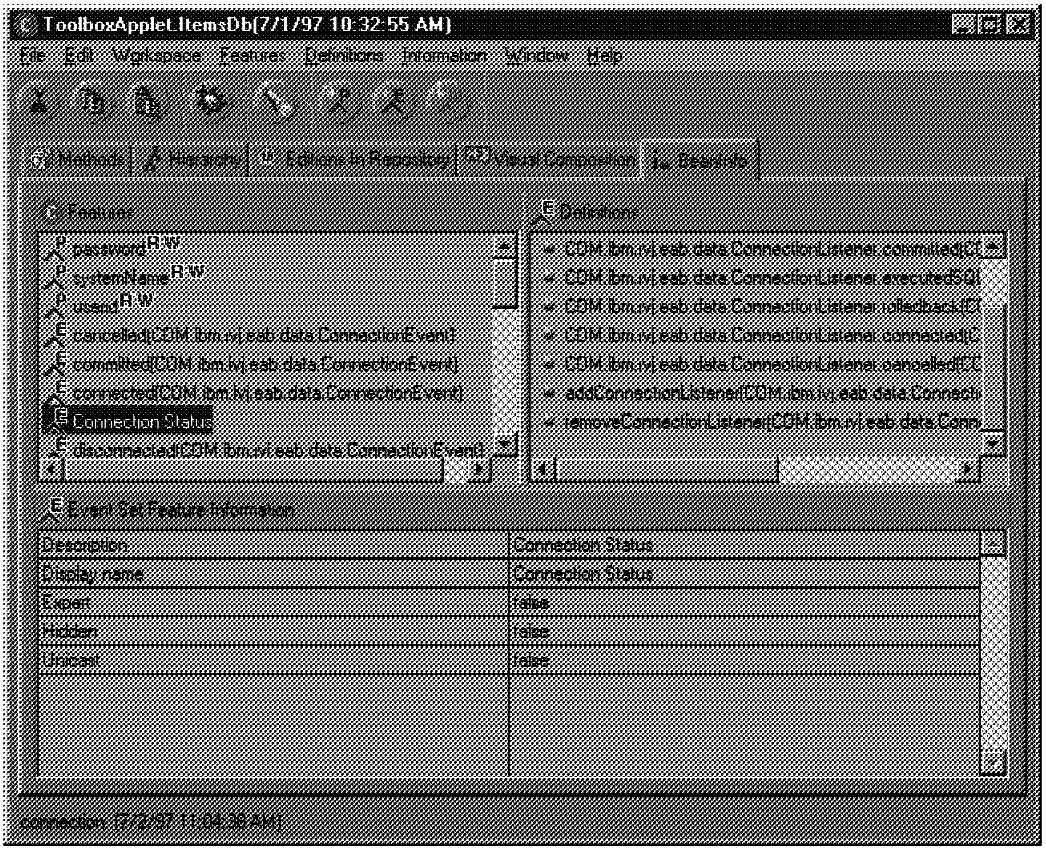
We select "Features" from the menu bar, and then select "New Event Set Feature..."



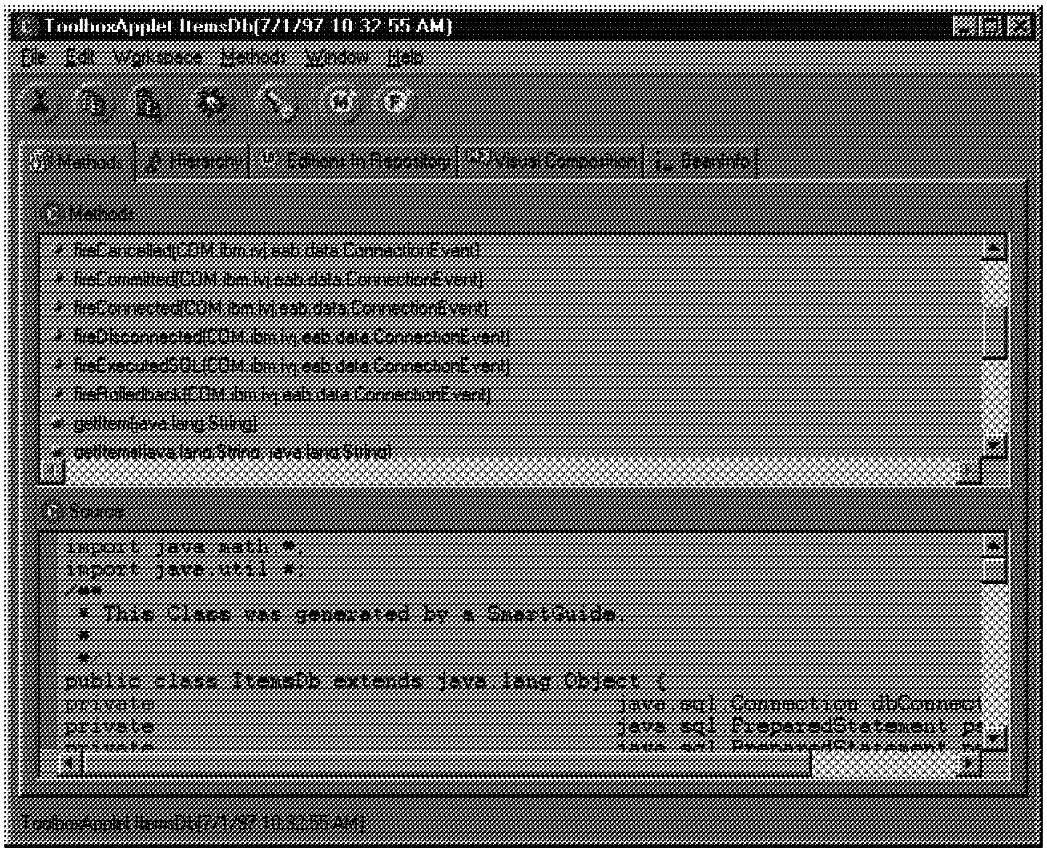
We use the drop down list box to choose "connection" as the "Event name" and **COM.ibm.ivj.eab.data.ConnectionListener** as the Event listener. We then click on "Next".



We fill in the information as shown in the preceding figure where the display name and short description let you recognize the events easily. We then click on Finish.



Now, the previous events and other methods are added automatically.



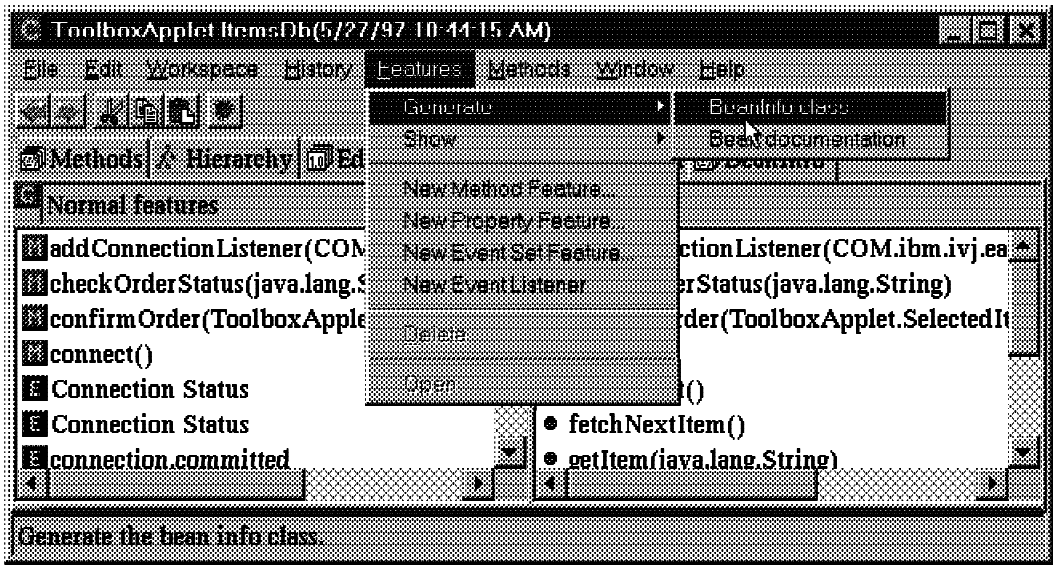
Also, the preceding methods are created as the support functions for firing an event. Whenever you want to fire an event, you use the fire*Event*() function. To illustrate, we fire the "Connected" event at the end of the ItemsDb "Connect()" method.

```
public String connect () {
try{
Class.forName("COM.ibm.as400.access.Driver");
dbConnect =
java.sql.DriverManager.getConnection("jdbc:as400:          .//" +
  systemName +
  "/csdb;naming=system;errors=full;date
  format=iso",userid,password);
psItem =  dbConnect.prepareStatement("SELECT * FROM
  CSDB/ITEM WHERE IID =  ?");
psItemRange =  dbConnect.prepareStatement("SELECT *
  FROM CSDB/ITEM WHERE IID =  ? AND IID =  ?");
psCustomerDb =  dbConnect.prepareStatement("SELECT
  CID FROM CSDB/CSTMR WHERE CID =  ? AND CDID=001 AND
  CWID='0001'");
psQuantityInHand =
  dbConnect.prepareStatement("SELECT STQTY FROM CSDB/STOCK
  WHERE STWID= '0001' AND STIID=?");
} catch (Exception e) {
System.out.println("connect(): "+e);
e.printStackTrace();
return "Connect: "+e;}
fireConnected(new
    COM.ibm.ivj.eab.data.ConnectionEvent(this," "));
return "Connect Successfully";
}
```

We add the following statement:

"**fireConnected(new COM.ibm.ivj.eab.data.ConnectionEvent(this," "));**"

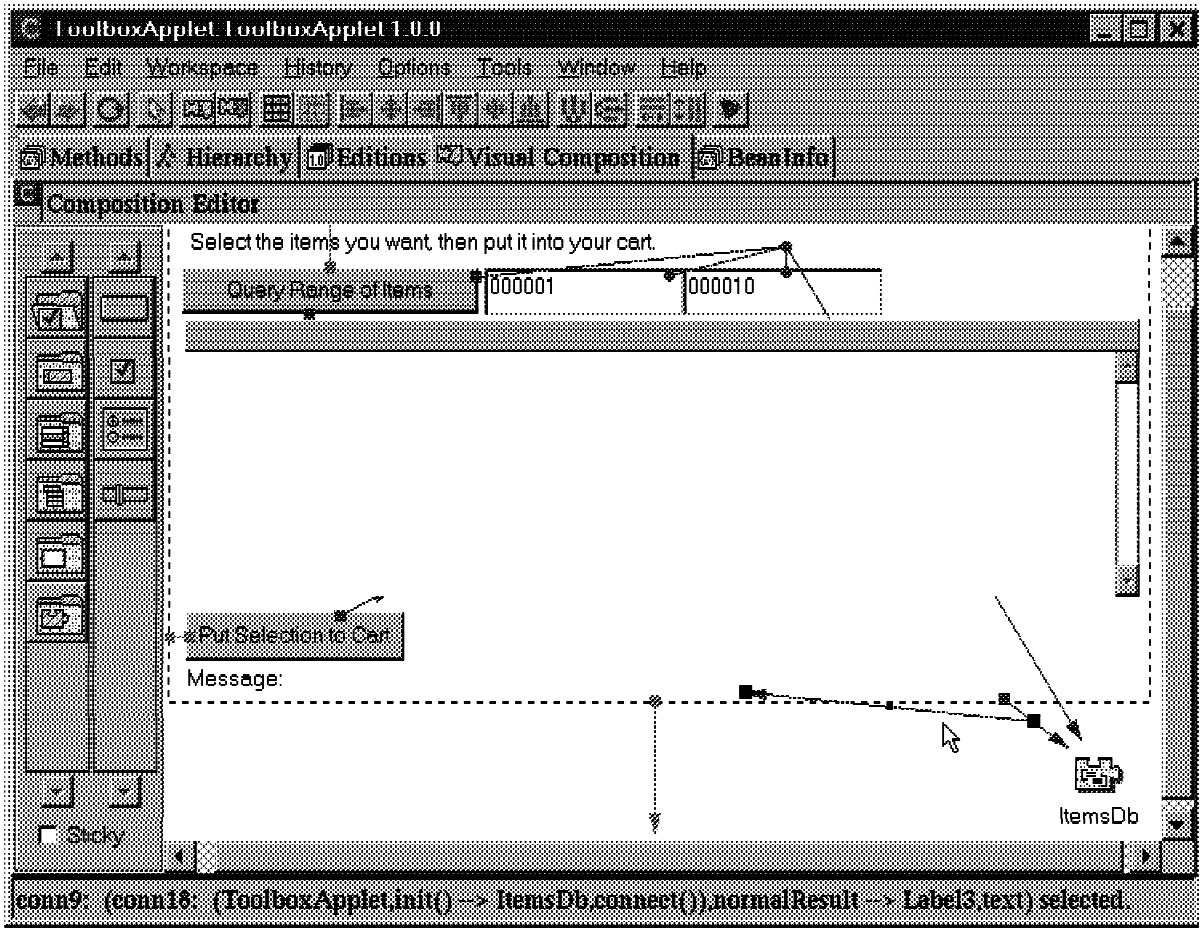
to the end of the "Connect()" method as previously shown, just before the last return statement, so that if the connection and all prepare statements are done successfully, we fire the "Connected" event with "this" as the source of the event.



Finally, we choose "Features", "Generate", and "BeanInfo Class" to generate a complete "ItemsDbBeanInfo" Class to make "ItemsDb" a real and complete JavaBean.

We have finished all of the definitions for our "ItemsDb" Bean. To show the advantage of using JavaBeans, we modify the "ToolboxApplet" class. In fact, without any modification, the "ToolboxApplet" still works.

7.4.2 Modification of "ToolboxApplet" Class



The "ToolboxApplet" class now looks similar to the preceding figure. We are making the following change:

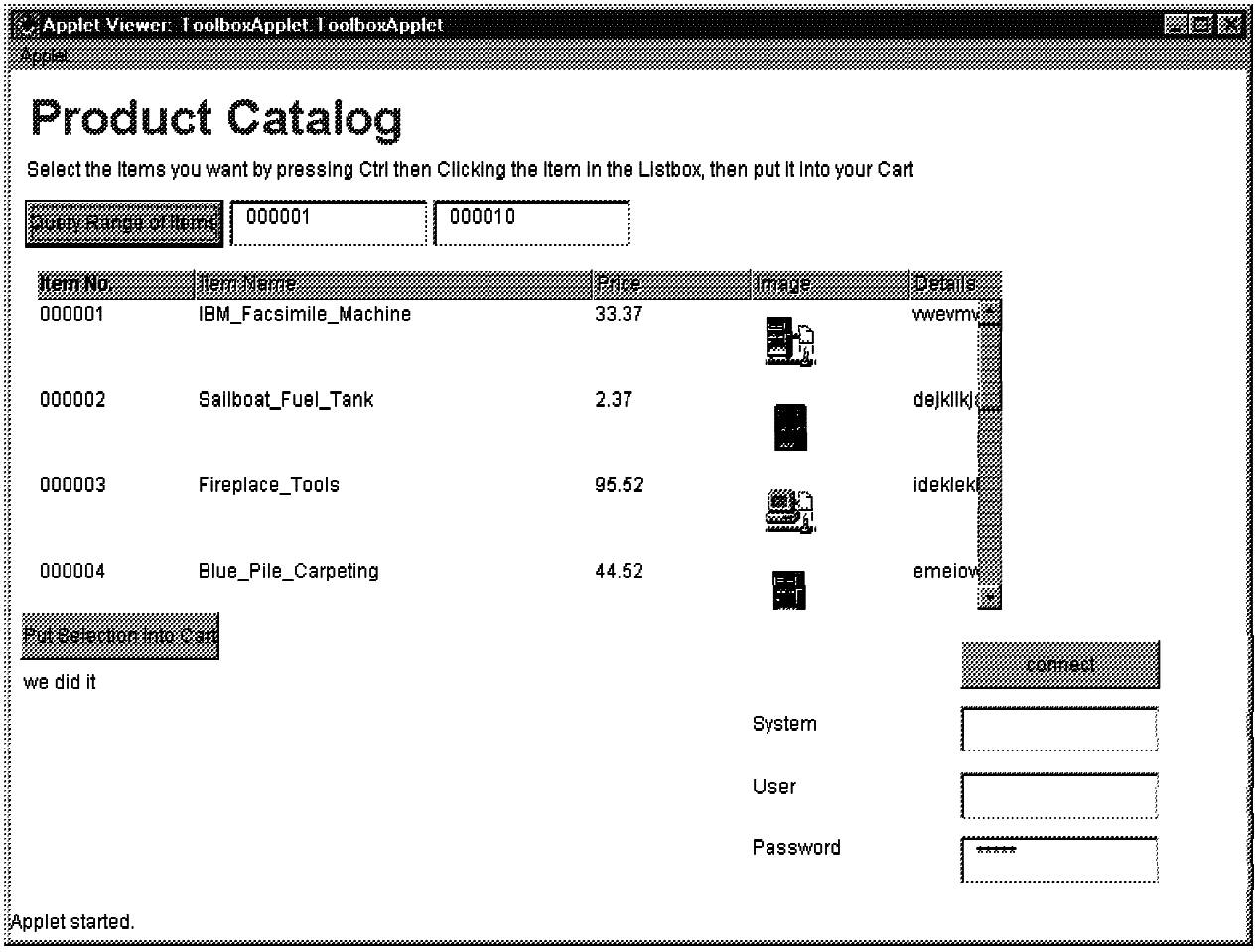
The normal result of the connection **ToolboxApplet,start()** to **ItemsDb,connect** is currently connected to the Message textbox. If the connection method runs successfully, we display the return value "Connect Successfully" in the text box.

The new connection is as follows:

We connect the firing of the ConnectionEvent, which we added to the connection method. We connect **ItemsDb,connection.connected** to **Label3,setText()**

We connect the "Connected" event to the Text and set the text **parameter** to whatever we want (for example "We did it").

7.4.2.1 Additional Modifications



We can now make some additional changes to the "ToolboxApplet" to make advantage of the new capabilities of the "ItemsDb" class.

- We change the connection "start() -- connect()" to a button named **Connect** that we connect to the ItemsDb connect method, so that we can manually start the connection rather than connecting when the applet starts.
- We change the "Query Range of Items" button so that when the applet is loaded, the button is disabled and we enable the button only when we connect successfully to the AS/400 system. We use the firing of the "connected" event to enable the button.
- We pass in the System Name, UserID, and Password from screen-text fields rather than storing them in variables.

Because we can use the "CONNECTED" event, we have more control over the interaction of the parts. We can make changes without changing code. Because of the bean, we have all the "Methods", "Properties", and "Events" available in the visual Editor of any Bean-enabled development environment. This makes is easy to reuse the capabilities of the object without having to read source code or documentation.

7.5 Advanced JavaBeans Concepts

BeanInfo class

If a bean creator does not want to leave property, method, and event finding up to the Introspector or wants 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 is there to make it easier for a developer to quickly add only some BeanInfo by overriding the methods already present in the SimpleBeanInfo class.

Note: Some builder tools do not use the Introspection if a BeanInfo file is present, so you must list all properties, methods, and events you want visible in the BeanInfo class if you list any.

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 a couple of methods need to be added to the main class to support bound property listeners.

Contrained properties

A contrained or vetoable property is similar to a bound property, but not only are listeners notified when the property has changed, they have the opportunity to disallow a change to occur. For example, a person might 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 not only lets you to send all of the contents of the property at once, but allows you to read and write one item in the array at a time. The following example 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];
}
```

```
import java.beans.*;
public class BeanNameBeanInfo extends SimpleBeanInfo {
 public PropertyDescriptor[] getPropertyDescriptors() {
 try {
IndexedPropertyDescriptor pd = new IndexedPropertyDescriptor("names",
BeanName, "getNames", "setNames", "getIndexNames", "setIndexNames");
PropertyDescriptor allDescriptors[] = { pd };
return allDescriptors;
 } catch (Exception e) {
return null;
 }
 }
}
```

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.

7.5.1 What Makes a Good JavaBean

Before going out and converting/updating all of your Java classes to beans, you should decide which Java classes are best suited to become beans. The basic question to ask yourself is this: "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 might 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, color) simply have get and set methods. So, if you follow the naming conventions when making any object, less work must be done to make your object a bean.

7.5.2 References and More Information

- JavaBeans For Dummies. Emily Vander Veer.
- Sun Website: http://java.sun.com/beans

Chapter 8. Java on AS/400 System

The focus of this redbook has been on developing AS/400 client/server applications in which the client code is developed using Java. Currently we can only run Java on the client. The reason for this is that there is not a generally available Java Virtual Machine (JVM) for the AS/400 system.

In the future, this will change; IBM has announced plans to support Java on the AS/400 system. In this chapter, we provide a preview of what the Java support on the AS/400 system will be.

8.1 Java on the AS/400 System

The current status of Java running on the AS/400 system is:

- Currently there is a JDK 1.0.2 available from IBM Hursley Lab:
  - Initially without AWT GUI classes.
  - Available as a "Technology preview" to get you started:

    AVAILABLE @ www.hursley.ibm.com/javainfo
- In the future, there will be a JDK 1.1 implementation:
  - It will provide a JVM that runs in the AS/400 System Licensed Internal Code.
  - An AS/400 Java compiler will also be provided.
- The AS/400 Toolbox classes will be available on the AS/400 system for Java application development.
- VisualAge for Java will be able to produce both client and AS/400 Java applications and applets.
- Any Java development tool can be used to write programs that run in the AS/400 system JVM. VisualAge for Java will provide unique support for the AS/400 system.

8.2 AS/400 Java Virtual Machine

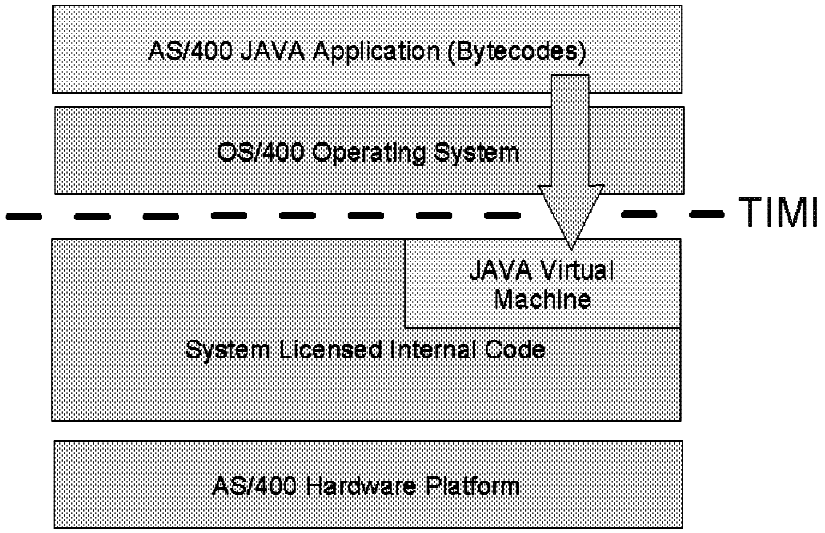


Figure 42. AS/400 JVM

This picture shows the AS/400 architecture. At the bottom, we have the AS/400 hardware platform. Originally, this platform was implemented on a CISC processor. With the announcement of the PowerPC RISC Technology, the hardware platform is now implemented on a Power PC processor.

Above the processor is the System Licensed Internal Code (SLIC). It is sometimes called the microcode. This is a layer of software that implements the basic functions of the operating system such as task management, memory management, and data management.

The Technology Independent Machine Interface (TIMI) provides the interface to the SLIC layer. This is sometimes referred to as the machine interface (MI). Nothing below this layer is public and applications cannot access functions hidden by this interface. This is how the AS/400 system provides for a seamless technology transition without any impact to applications.

Above the machine interface is the operating system itself. Because the operating system is above the MI, most of it is portable and technology independent.

Above the operating system are the applications. They are independent of the underlying technology.

The Java Virtual Machine is ported to many hardware and software platforms. It provides a uniform, platform independent interface for Java applications. This interface is implemented through the Java APIs. Java applications only use this set of APIs; they never see the underlying physical implementation.

Java applications sit on top of the Java Virtual Machine. Compiled Java applications are in the form of byte-codes. The byte-codes are interpreted by the Java Virtual Machine.

To run Java applications on the AS/400 system, we have to implement a Java Virtual Machine. The AS/400 JVM is implemented inside the microcode as an extension to the SLIC. This provides for good performance for the JVM and also maintains the technology independence of the AS/400 system. The AS/400 system can present both its TIMI and Java interface to others.

8.3 Java on the AS/400 Server

Java programs can run on the AS/400 system as either:

- Interpreted byte-codes
- Native AS/400 Java compiled programs

A new CRTJVAPGM command allows for interpreted and compiled modes. A compiled version should perform well. Java byte-codes are encapsulated in hidden AS/400 *SRVPGM objects. They are associated with Java class files in the IFS.

- Native AS/400 Java compiler:

  This creates an AS/400 *SRVPGM object and links it to the Java class (bytecodes) produced by any Java programming tool. The compiler has the capability to create two flavors of a Java Program. The first type is interpreted by the native Java Virtual Machine (JVM) running inside the System Licensed Internal Code (SLIC). The second encapsulates the byte-codes into native AS/400 instructions to provide for a much faster execution. The compiler is invoked by a command such as CRTJVAPGM. This command requires parameters such as:

  – CLSF (class file name):

    The name of the IFS file containing the Java class file.

  – GENTYPE (type of AS/400 program to be created):

    - *COMPILED
    - *INTERPRET

  – Other parameters such as the ones found on other CRTxxxPGM commands:

    - OPTIMIZE (optimization level)
    - REPLACE (whether to replace an existing program)
    - ENBPFRCOL(whether collection of performance data is enabled).

  A hidden AS/400 *SRVPGM object is created. Java is and will be IFS based. That is, all Java files such as a source file (java suffix) or bytecodes (class suffix) are stream files stored in the IFS. They are not QSYS type objects. The compiler will create an AS/400 *SRVPGM object. Unlike any other AS/400 *SRVPGM type object, it will not appear in any library, thus it is not directly accessible. The compiler provides a pointer into the Java class file and the *SRVPGM object is only accessible through this pointer. The *SRVPGM object associated with a Java class file is saved and restored automatically when the Java class file is saved and restored through the use of standard IFS SAV and RST commands.

8.4 Java Applications on Server

Java applications on the server can:

- Call AS/400 ILE C + + service programs
  - Using the Java built-in Native Method Invocation.
- Run as batch programs (no GUI operations):
  - Remote AWT classes may allow GUI support.
- Use the AS/400 Toolbox for Java classes.
- Interface with client-based Java programs:
  - Using the Java Remote Method Invocation interface.
  - Using the Java sockets support.

Now let's look at some of the capabilities this is going to offer.

- Call AS/400 ILE C programs:

  The Java Virtual Machine implementation will enable a Java program to call an ILE C program. This will make use of the standard Java Native Method Invocation (NMI) capability. It provides for a tight integration of Java applications with existing applications.

  Note: The ability to call RPG or COBOL programs using JNI may be provided with some restrictions due to thread safety issues. The goal is to make this support available with no restrictions.

- Batch programs (no GUI operation):

  It does not make sense to have an AS/400 program try to perform graphical user interface operations because the AS/400 system has no native GUI capable devices. The remote AWT support may be used to provide a "thin" client/server implementation. That is, the program logic resides on the server with only the graphical user interface done on the client.

- AS/400 Toolbox class interface:

  In this redbook, we cover all of the functions that a Java application (or applet) can be enabled to when using the AS/400 Toolbox for Java classes. As these classes are 100% pure Java classes, they can also be executed on the AS/400 system. Having the AS/400 Toolbox for Java classes available on the AS/400 system provides all AS/400 based (server) applications with the same functions and capabilities of client/server applications. For example, they can execute batch commands, call existing AS/400 programs, use data queues, and interface to the printing support.

- JDBC:

  The JDBC interface to AS/400 databases or stored procedures is also available. Rather than using the toolbox classes, it uses native AS/400 support.

8.5 AS/400 Java Remote Method Invocation

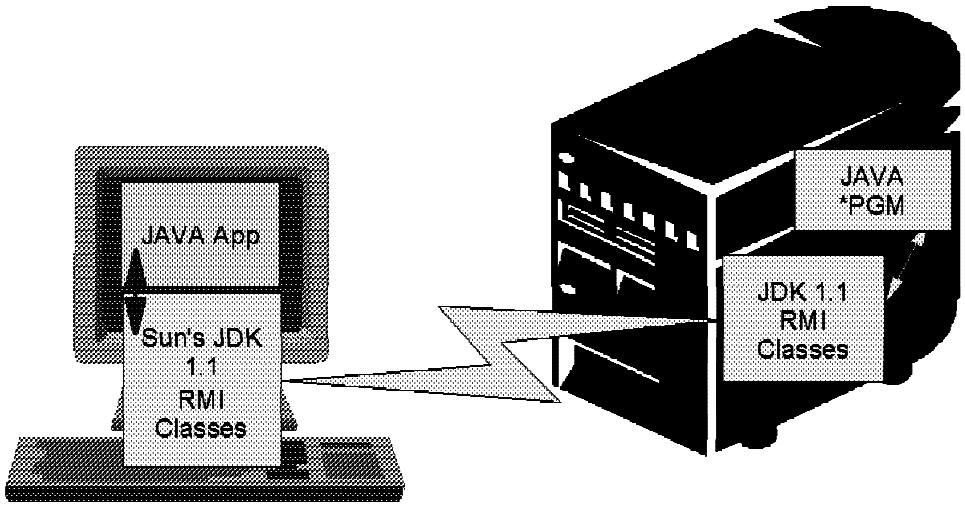


Figure 43. RMI

This chart illustrates how distributed client/server JAVA applications are implemented with the AS/400 system using Java Remote Method Invocation. The AS/400 server is on one side. The Java enabled client is on the other side. When starting a program on the client, this program makes uses of standard JDK 1.1 classes. Among these classes are the Remote Method Invocation classes. RMI classes are capable of opening a Sockets connection to a server. If the client is running a Java application, this can be any server in the network. If it is running an applet, it can only be the server the applet was downloaded from. Once the connection is granted, the RMI classes on the client start communicating with their equivalent classes on the server side. In turn, the RMI classes on the server invoke a Java program on the AS/400 system. Because the AS/400 Toolbox for Java classes are 100% pure Java classes, they also run on the AS/400 system, thus, providing the AS/400 side Java applications with easy access to any AS/400 resources.

8.6 AS/400 Java Proxy Interface

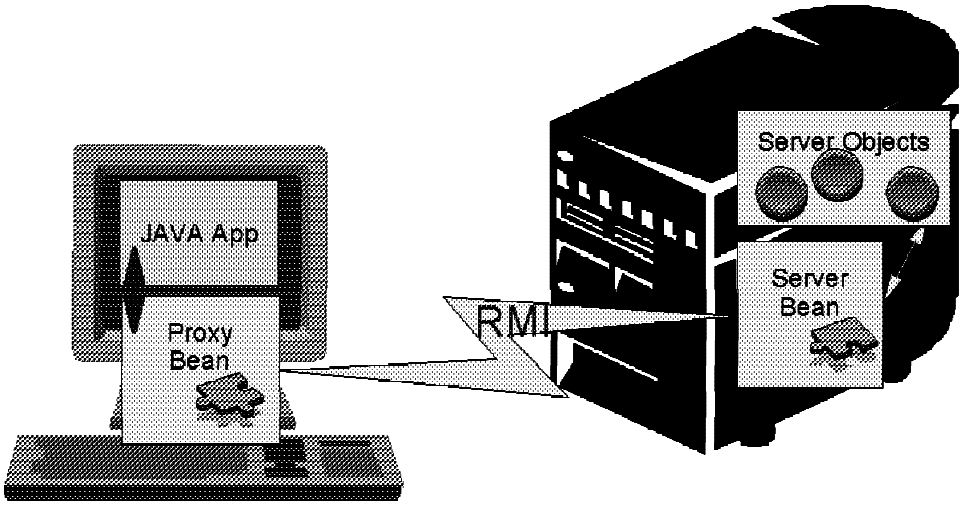


Figure 44. Proxy

Here we show how a distributed Java client/server application behaves when it uses the proxy concept. First, we need a server on the network. This can be an AS/400 system. Somewhere in the network, we have a Java-enabled client.

On the client, we start a Java application or a Java applet. This application needs to communicate with some other object to perform some task. As far as the application is concerned, everything looks as if the object was resident on the same machine as the rest of the Java application. In reality, only a proxy of the real object (that is, some sort of shadow) is present on the client side. This proxy can be automatically created by the Proxy Builder included with VisualAge for Java Enterprise Edition.

When the Java application actually needs to communicate with the remote object (for example, to invoke a method of the remote object), it, in fact, invokes the method. This is intercepted by the proxy and the remote invocation call is sent on the network using the Remote Method Invocation API to the server. On the server side, the counter part (called the server bean) receives the Method Invocation Call sent by the Proxy bean on the client side. It then passes the method call to the real object that resides on the server side. The real object executes the request. If needed, information such as return value can be passed back using the same technique to the requesting application back on the client side. This is quite simple and effective. It allows the application developers to distribute objects on the network without changing the application.

When compared to the Remote Method Invocation technique, the main difference is that the application is not aware of the fact that the object it is communicating with is now a remote object instead of being a local object. The proxy concept is really powerful in a distributed client/server environment.

8.7 VisualAge for Java - AS/400 Feature

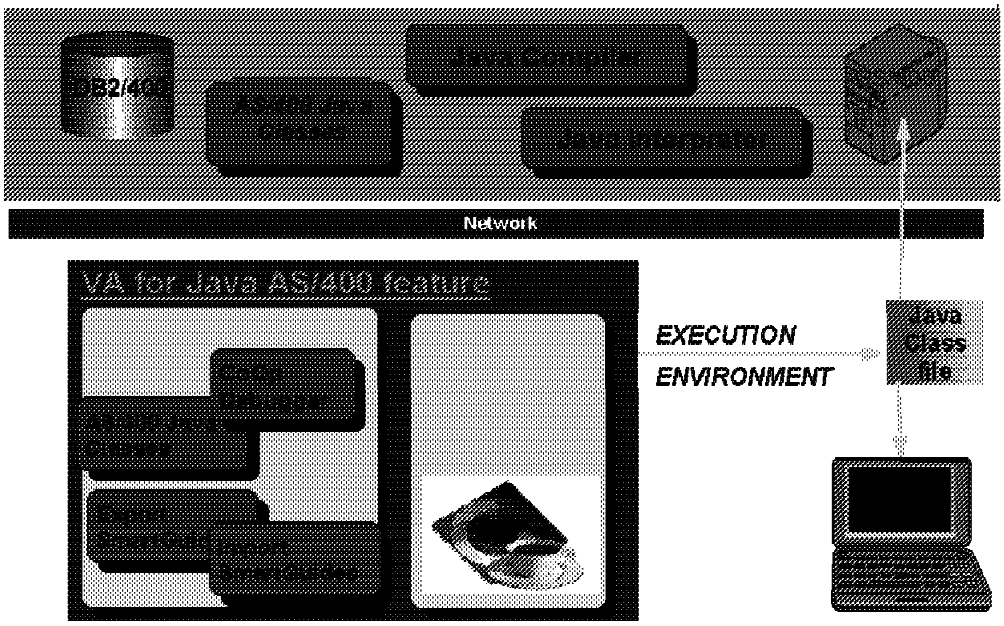


Figure 45. VA for Java - AS/400 Feature

VisualAge for Java is available today. An AS/400 feature will be added to the next release of VisualAge for Java. It will include AS/400 specific functions to help AS/400 programmers develop both client and server Java programs that target the AS/400 system. Functions available in the AS/400 feature are:

- CoOperative Debugger enabled for AS/400 Java programs (same debugger in VisualAge C++/400 and CODE/400)
- AS/400 Toolbox support for AS/400 Java applications/applets
- Integration with AS/400 Java compiler ("export SmartGuide")
- AS/400-programmer-unique documentation and samples
- AS/400 legacy access bean-builder SmartGuides
- Display file (DDS) to Java AWT conversion
- Performance analysis tool for server Java

One of the key components of the AS/400 Feature is a debugger that allows AS/400 Java applications to run under the control of a graphical user interface client-based debugger. This will allow application developers to set breakpoints, view and modify program variables, and change Java code interactively.

The AS/400 Toolbox for Java classes that are available today for client-based programs will be available for server based programs. They will be included with the VisualAge for Java - AS/400 Feature and fully integrated with VisualAge for Java's Integrated Development Environment.

The AS/400 system will provide a special Java compiler to allow Java programs to be compiled into AS/400 *PGM objects. This is to allow Java programs to perform more efficiently on the AS/400 platform. It allows the interpretive nature of the Java Virtual Machine to be bypassed. VisualAge for Java - AS/400

Feature will provide integration between the VisualAge for Java development environment and the AS/400 Java compiler.

For more information see:

www.software.ibm.com/ad/as400/vajava

8.8 Java on the AS/400 Conclusions

- Java is a good fit with AS/400 architecture:
  - JVM interpreted layer similar to MI:
    - Abstraction layer between hardware and applications
    - Single level store good for persistent objects
  - Brings new application development toolset:

    Any Java tool can produce AS/400 applications.
  - Helps solve portability problems.
  - Compiled Java will perform and scale.
  - Provides integration with RPG/COBOL programs.

Java is a good fit for the AS/400 architecture. The concept of a Java Virtual Machine is similar to the AS/400 Machine Interface. Both offer a level of abstraction to shield application developers from the real operating system. The AS/400 system is designed using an object-based architecture. Its use of a single level store concept will be good for maintaining persistent objects.

Java on the AS/400 system will make new and more modern application development tools available to application developers. Java is an object-oriented language and when used properly, provides all of the inherent benefits of object-oriented application development. One of the main benefits of Java is its portability. Application developers can embrace one programming language for all of the platforms they need to support.

One of the biggest concerns of Java application developers is performance. Java is an interpreted language and this implies slow performance. The AS/400 system, by providing compiled Java support, can overcome the interpreted nature of Java.

Java on the AS/400 system will provide full integration with the AS/400 Toolbox for Java classes. This will give application developers access to the full range of AS/400 resources including already existing programs.

Appendix A. Example Programs

The Java client programs and the AS/400 programs and libraries used in this redbook are available to be downloaded through the Internet. These examples were developed using VisualAge for Java Enterprise edition. OS/400 V3R2, V3R7 or later is required. The following VisualAge for Java projects are available:

- AppletWorkshop

  Internet "Shopping" applets, see Chapter 6, “Developing AS/400 Java Applets” on page 155.

- AppletWorkshopListbox

  Multi-column listbox for "Shopping" applet.

- DaxPartsProject

  Dax example "Parts Ordering" application, see Chapter 5, “Enterprise Access Builder For Data (DAX)” on page 135.

- Taligent

  Multi-column listbox for Toolbox examples.

- TeamLabs

  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 61.

  JavaBean Fancy Label example, see Chapter 7, “JavaBeans” on page 189.

- Workshop

  Toolbox examples for JDBC, see Chapter 3, “AS/400 Toolbox for Java” on page 61.

Important Information

These example programs have not been subjected to any formal testing. They are provided "AS-IS"; they should be used for reference only. Please refer to the Special Notices section at the back of this document for more information.

A.1 Downloading the Files from the Internet Web Site

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

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

Click on Downloads and then select directory **SG242152**. In the SG242152 directory, click on readme.txt.

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

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
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                   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
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* SUBROUTINE BADOPT
C*
C* AN UNRECOGNIZED OPTION PARAMETER WAS SET - RETURN 'U' FOR
C* UNKNOWN.
C*
C*---------------------------------------------------------------
C     BADOPT        BEGSR
C                   MOVE      3             #OPT
C                   RETURN
C                   ENDSR
C*
```

B.3 DPCXRPG/RPGLE

```
H*     DPCXRPG
H*
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     IF   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      Position to start of file
C*             F      Fetch the next record based on cursor posn.
C*             E      End the program
C*-------------------------------------------------------------
C     *ENTRY        PLIST
C                   PARM                    OPTION            1
C                   PARM                    PARTNO
C                   PARM                    PARTDS
C                   PARM                    PARTQY
C                   PARM                    PARTPR
C                   PARM                    PARTDT
C     OPTION        CASEQ     'S'           ONEREC
C     OPTION        CASEQ     'A'           ALLREC
C     OPTION        CASEQ     'F'           NXTREC
C     OPTION        CASEQ     'E'           ENDPRG
C                   CAS                     BADOPT
C                   ENDCS
C*-------------------------------------------------------------
C* SUBROUTINE - ONEREC
C* 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
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
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 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.4 DQXRPG/RPGLE

```
H*     DQXRPG
H*
H* This is a never-ending-program that runs in the background
H* as a batch job. It checks the data queue DQINPT for
H* any queue entries received. Once an entry arrives in the
H* data queue, the program retrieves and processes it.
H*
H* This program should be submitted with the SBMJOB command
H* and terminated with ENDJOB OR WRKACTJOB commands, or by
H* placing an entry starting with 'E' on the DQINPT data queue.
H*---------------------------------------------------------------
```

```
H
FPARTS     IF   E           K DISK
D*-------------------------------------------------------------
D* DATA STRUCTURES
D*
D* DATAI - input data record 6 bytes
D* DATAO - output data record 48 bytes
D*-------------------------------------------------------------
D DATAI           DS
D  OPTION                 1      1
D  INPNO                  2      6  0
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
D*-------------------------------------------------------------
D* CONSTANTS
D*
D* DQINPT - data queue used for receiving input records
D* DQOUPT - data queue used for sending records
D* APILIB - library name
D*-------------------------------------------------------------
D DQINPT          C                   CONST('DQINPT    ')
D DQOUPT          C                   CONST('DQOUPT    ')
D LIBL            C                   CONST('APILIB    ')
C*-------------------------------------------------------------
C* MAIN PROGRAM
C*
C* Loop on read to data queue. Action depends on the 'option'
C* flag:
C*          Option    Action
C*             S      Retrieve a single record for supplied key
C*             A      Retrieve all records in file
C*             E      End the program
C*-------------------------------------------------------------
C                   EXSR      RCVDQ
C     OPTION        DOWNE     'E'
C                   EXSR      READR
C                   EXSR      RCVDQ
C                   ENDDO
C                   SETON                                        LR
C*-------------------------------------------------------------
C* SUBROUTINE RCVDQ
C*
C* This subroutine performs the QRCVDTAQ function. Notice that
C* the wait parameter is set to a negative value to force it
C* to wait until a queue entry is available.
C*
C     RCVDQ         BEGSR
C                   MOVE      DQINPT        QUEUEI           10
C                   MOVE      LIBL          LIBLD            10
C                   Z-ADD     6             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
C*---------------------------------------------------------------
C* SUBROUTINE - READR
C* This subroutine retrieves the part number from the data queue
C* DQINPT, searches the data base file PARTS using the part number
C* just received. If the record is found, send the record to the
C* data queue DQOUPT. If option 'A' is received, send all records
C* to the data queue DQOUPT.
C*
C* The 'result' flag is set as follows
C*          Result    Meaning
C*             Y      Record found and being returned
C*             X      Record not found or eof
C*---------------------------------------------------------------
C     READR         BEGSR
C     OPTION        IFEQ      'A'
C* Return all records in the file
C     *LOVAL        SETLL     PARTS
C                   READ      PARTS                                  60
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* 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                   ENDSR
C*---------------------------------------------------------------
C* SUBROUTINE SNDDQ
C*
C* This subroutine performs the QSNDDTAQ function.
C*
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. AS/400 Toolbox Example Java Code

This appendix has the following code:

JDBCExample.java

JDBCExampleDisplayAll.java

ToolboxGUI.java

DisplayAllParts.java

PartsContainer.java

StoredProcedureExample.java

DPCExample.java

DataQueueExample.java

RLExample.java

C.1 JDBCExample.java

```
package WorkShop;

public class JDBCExample extends java.awt.Frame
            implements java.awt.event.ActionListener {
private  java.sql.Connection dbConnect;
private  java.awt.Button ivjButton1 = null;
private  java.awt.Button ivjButton11 = null;
private  java.awt.Button ivjButton12 = null;
private  java.awt.Button ivjButton2 = null;
private  java.awt.Button ivjButton3 = null;
private  java.util.Date ivjDateFactory = null;
private  WorkShop.JDBCExampleDisplayAll ivjDisplayAll = null;
private  COM.ibm.ivj.javabeans.IMessageBox ivjIMessageBox1 = null;
private  java.awt.Label ivjLabel1 = null;
private  java.awt.Label ivjLabel11 = null;
private  java.awt.Label ivjLabel12 = null;
private  java.awt.Label ivjLabel13 = null;
private  java.awt.Label ivjLabel14 = null;
private  java.awt.Label ivjLabel141 = null;
private  java.awt.Label ivjLabel1411 = null;
private  java.awt.Label ivjLabel15 = null;
private  java.awt.Label ivjLabel151 = null;
private  java.awt.Label ivjLabel152 = null;
private  java.awt.TextField ivjTextField1 = null;
private  java.awt.TextField ivjTextField11 = null;
private  java.awt.TextField ivjTextField12 = null;
private  java.awt.TextField ivjTextField13 = null;
private  java.awt.TextField ivjTextField14 = null;
private  java.awt.TextField ivjTextField141 = null;
private  java.awt.TextField ivjTextField142 = null;
private  java.awt.TextField ivjTextField1421 = null;
private  java.awt.TextField ivjTextField15 = null;
private  java.awt.TextField ivjTextField151 = null;
private  java.awt.TextField ivjTextField152 = null;
private  java.sql.PreparedStatement psAllRecord;
private  java.sql.PreparedStatement psSingleRecord;
```

```
public String connectToDB(String systemName, String userid, String password) {
  try { 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",userid,password);

  psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?");
  psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS");

  } catch (Exception e) {showException(e); return "Connect Failed."; };


  return "Connected to AS/400.";
}
/**
 * This method was created by a SmartGuide.
 */
public void dispose () {
  try {
    psSingleRecord.close();
    dbConnect.close();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  if (ivjDisplayAll != null) ivjDisplayAll.dispose();
  super.dispose();
return;
}
/**
 * This method was created by a SmartGuide.
 * @param aDate java.util.Date
 */
public String formatDate (java.util.Date aDate) {
  java.text.DateFormat formatter = new java.text.SimpleDateFormat("hh:mm:ss a");
  return formatter.format(aDate);

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

    if (rs.next()) {
      partDesc.setText(rs.getString("PARTDS"));
      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("");
      partPrice.setText("");
      partDate.setText("");
      return "Record not found.";
    }
  } catch  (Exception e) {e.printStackTrace();showException(e); return null; }

  return "Record found.";
}
public String updateRecord (String partno, String partdesc, String partqty,
```

```
            String partprice, String partdate) {

  // strip the leading dollar sign if it exists...
  String tempPrice = partprice.indexOf('$') == 0 ? partprice.substring(1) : partprice;

  try {
    java.sql.Statement s = dbConnect.createStatement();
    String updatestring = "UPDATE PARTS SET PARTDS = '" + partdesc
     + "',PARTQY = " + partqty +
     ", PARTPR = " + tempPrice + ", PARTDT = '" + partdate + "'
     WHERE PARTNO = " + partno;
    s.executeUpdate(updatestring);
  } catch (Exception e) {showException(e); return "Update failed";}

  return "Record Updated.";
}
}
```

C.2 JDBCExampleDisplayAll.java

```
package WorkShop;

public class JDBCExampleDisplayAll extends java.awt.Frame
            implements java.awt.event.ActionListener {
private  java.sql.Connection dbConnect;
private  java.awt.Button ivjButton1 = null;
private  COM.ibm.ivj.javabeans.IMessageBox ivjIMessageBox1 = null;
private  com.taligent.widget.MultiColumnListbox ivjMultiColumnListbox1 = null;
private  java.sql.PreparedStatement psAllRecord;

/**
 * Constructor that utilizes an existing database connection and precompiled statement.
 * @param dbConnect java.sql.Connection
 * @param psAll java.sql.PreparedStatement
 */
public JDBCExampleDisplayAll (java.sql.Connection dbc,
                java.sql.Prepared Statement psAll) {
  this();
  dbConnect = dbc;
  psAllRecord = psAll;
  this.setUpListBox();
  this.populateListBox();
  this.show(true);
}
public static void main(java.lang.String[] args) {
  try {
      WorkShop.JDBCExampleDisplayAll aJDBCExampleDisplayAll =
            new WorkShop.JDBCExampleDisplayAll();
      aJDBCExampleDisplayAll.setUpListBox();
      aJDBCExampleDisplayAll.show(true);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception occurred in main() of JDBCExampleDisplayAll");
      System.err.println(exception);
    }
}
/**
 * This method was created by a SmartGuide.
 */
public void populateListBox () {
  java.sql.ResultSet rs = null;
  String[] array = new String[5];

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

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

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

      ivjMultiColumnListbox1.addRow(array);
    }
  } catch  (Exception e) {showException(e);}

  return;
}
/**
 * This method was created by a SmartGuide.
 */
public void setUpListBox () {

  ivjMultiColumnListbox1.showBorder().showHorizontalSeparator().
       .showVerticalSeparator();
  ivjMultiColumnListbox1.addColumn("Part#").setWidth(50);
  ivjMultiColumnListbox1.addColumn("Description").setWidth(200);
  ivjMultiColumnListbox1.addColumn("Qty").setWidth(50).setAlignment(2);
  ivjMultiColumnListbox1.addColumn("Price").setWidth(50).setAlignment(2);
  ivjMultiColumnListbox1.addColumn("Received").setWidth(75).setAlignment(2);

  return;
}
/**
 * This method was created by a SmartGuide.
 * @param e java.lang.Exception
 */
public void showException (Exception e) {
  try {
    getIMessageBox1().showException(e);
  } catch (java.lang.Throwable exception) {}

  return;
}
}
```

C.3 ToolboxGUI.java

```
package WorkShop;

public class ToolboxGUI extends java.awt.Panel implements
            java.awt.event.ActionListener {
private  WorkShop.DisplayAllParts aDisplayAllParts = null;
protected  java.util.Vector aToolboxGUIListener = null;
private  java.sql.Connection dbConnect;
private  java.awt.Button ivjButton1 = null;
private  java.awt.Button ivjButton11 = null;
private  java.awt.Button ivjButton12 = null;
private  java.awt.Button ivjButton2 = null;
private  java.awt.Button ivjButton3 = null;
private  java.util.Date ivjDateFactory = null;
private  COM.ibm.ivj.javabeans.IMessageBox ivjIMessageBox1 = null;
private  java.awt.Label ivjLabel1 = null;
private  java.awt.Label ivjLabel11 = null;
private  java.awt.Label ivjLabel12 = null;
private  java.awt.Label ivjLabel13 = null;
private  java.awt.Label ivjLabel14 = null;
private  java.awt.Label ivjLabel141 = null;
private  java.awt.Label ivjLabel1411 = null;
```

```
private  java.awt.Label ivjLabel15 = null;
private  java.awt.Label ivjLabel151 = null;
private  java.awt.Label ivjLabel152 = null;
private  java.awt.TextField ivjTextField1 = null;
private  java.awt.TextField ivjTextField11 = null;
private  java.awt.TextField ivjTextField12 = null;
private  java.awt.TextField ivjTextField13 = null;
private  java.awt.TextField ivjTextField14 = null;
private  java.awt.TextField ivjTextField141 = null;
private  java.awt.TextField ivjTextField142 = null;
private  java.awt.TextField ivjTextField1421 = null;
private  java.awt.TextField ivjTextField15 = null;
private  java.awt.TextField ivjTextField151 = null;
private  java.awt.TextField ivjTextField152 = null;

/**
 * Connect to the AS/400 using JDBC
 * @param systemName java.lang.String
 */
public String connectToDB(String systemName, String userid, String password) {
  try {
    ((PartsContainer)getParent()).connectToDB(systemName, userid, password);

  } catch (Exception e) {showException(e); return "Connect Failed."; };

  return "Connected to AS/400.";
}
/**
 * This method was created by a SmartGuide.
 */
public void displayAll () {
  try {
    aDisplayAllParts = new WorkShop.DisplayAllParts((PartsContainer)(this.getParent()));
  } catch (java.lang.Exception e) {showException(e);}
  return;
}
/**
 * This method was created by a SmartGuide.
 */
public String getPart (String partNo, java.awt.TextField partDesc,
  java.awt.TextField partQty,
  java.awt.TextField partPrice, java.awt.TextField partDate) {
  String retString = null;

  try { retString =
   ((PartsContainer)getParent()).getRecord(partNo, partDesc, partQty,partPrice, partDate);
  } catch (Exception e) {showException(e);}

  return retString;
}
/**
 * This method was created by a SmartGuide.
 * @return java.lang.String
 * @param aParameter int
 */
public String updatePart (String partno, String partdesc,
        String partqty, String partprice, String partdate) {

  String retString = null;

  try {retString=((PartsContainer)getParent()).updateRecord(partno,
           partdesc, partqty, partprice,partdate);
  } catch (Exception e) {showException(e);}
```

```
  return retString;
}
}
```

C.4 DisplayAllParts.java

```
package WorkShop;

public class DisplayAllParts extends java.awt.Frame implements
             java.awt.event.ActionListener {
private  java.awt.Button ivjButton1 = null;
private  COM.ibm.ivj.javabeans.IMessageBox ivjIMessageBox1 = null;
private  com.taligent.widget.MultiColumnListbox ivjMultiColumnListbox1 = null;

/**
 * Constructor
 * @return java.awt.Frame
 */
public DisplayAllParts() {
  super();
  setLayout(null);
  setBackground(new java.awt.Color(192, 192, 192));
  reshape(20, 20, 466, 427);
  setTitle("All Parts");
  this.add("ivjButton1", getButton1());
  this.add("ivjMultiColumnListbox1", getMultiColumnListbox1());
  initConnections();
}
/**
 * Constructor that utilizes an existing database connection and precompiled statement.
 * @param dbConnect java.sql.Connection
 * @param psAll java.sql.PreparedStatement
 */
public DisplayAllParts (PartsContainer aContainer) {
  this();
  this.setUpListBox();
  try {
    aContainer.populateListBox(ivjMultiColumnListbox1);
    this.show(true);
  } catch (Exception e) {showException(e);}

}
/**
 * Method to handle events for the ActionListener interface.
 * @param e java.awt.event.ActionEvent
 */
public void actionPerformed(java.awt.event.ActionEvent e) {
  if ((e.getSource() == getButton1()) ) {
    conn0(e);
  }
}
/**
 * conn0:  (Button1.action.actionPerformed -- DisplayAllParts.dispose())
 * @param e java.awt.event.ActionEvent
 */
private void conn0(java.awt.event.ActionEvent e) {
  try {
    this.dispose();
  } catch (java.lang.Throwable exception) {
  }
}
/**
 * Return the Button1 property value.
 * @return java.awt.Button
```

```
 */
private java.awt.Button getButton1() {
  if (ivjButton1 == null) {
    try {
      ivjButton1 = new java.awt.Button("Close");
      ivjButton1.setName("PBclose");
      ivjButton1.reshape(166, 380, 125, 30);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception creating Button1");
    }
  };
  return ivjButton1;
}
/**
 * Return the IMessageBox1 property value.
 * @return COM.ibm.ivj.javabeans.IMessageBox
 */
private COM.ibm.ivj.javabeans.IMessageBox getIMessageBox1() {
  if (ivjIMessageBox1 == null) {
    try {
      ivjIMessageBox1 = new COM.ibm.ivj.javabeans.IMessageBox();
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception creating IMessageBox1");
    }
  };
  return ivjIMessageBox1;
}
/**
 * Return the MultiColumnListbox1 property value.
 * @return com.taligent.widget.MultiColumnListbox
 */
private com.taligent.widget.MultiColumnListbox getMultiColumnListbox1() {
  if (ivjMultiColumnListbox1 == null) {
    try {
      ivjMultiColumnListbox1 = new com.taligent.widget.MultiColumnListbox();
      ivjMultiColumnListbox1.setName("LBallParts");
      ivjMultiColumnListbox1.reshape(18, 46, 426, 320);
      ivjMultiColumnListbox1.setBackground(new java.awt.Color(255, 255, 255));
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception creating MultiColumnListbox1");
    }
  };
  return ivjMultiColumnListbox1;
}
/**
 * Method to handle old AWT events
 * @return boolean
 * @param evt java.awt.Event
 */
public boolean handleEvent(java.awt.Event evt) {
  /* Note: Changes to this method will not be overwritten when code is re-generated */
  if (evt.id == java.awt.Event.WINDOW_DESTROY) {
    dispose();
    if (getParent() == null) {
      java.lang.System.exit(0);
    };
    return true;
  }
  return super.handleEvent(evt);
}
/**
 * Initializes connections
 */
private void initConnections() {
  /* Initialize the connections for the part */
```

```
  getButton1().addActionListener(this);
}
/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(java.lang.String[] args) {
  /* Note: Changes to this method will not be overwritten when code is re-generated */
  try {
      WorkShop.DisplayAllParts aDisplayAllParts = new WorkShop.DisplayAllParts();
      aDisplayAllParts.show(true);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception occurred in main() of DisplayAllParts");
    }
}
/**
 * This method was created by a SmartGuide.
 */
public void setUpListBox () {
  ivjMultiColumnListbox1.showBorder().showHorizontalSeparator().showVerticalSeparator();
  ivjMultiColumnListbox1.addColumn("Part#").setWidth(50);
  ivjMultiColumnListbox1.addColumn("Description").setWidth(200);
  ivjMultiColumnListbox1.addColumn("Qty").setWidth(50).setAlignment(2);
  ivjMultiColumnListbox1.addColumn("Price").setWidth(50).setAlignment(2);
  ivjMultiColumnListbox1.addColumn("Received").setWidth(75).setAlignment(2);

  return;
}
/**
 * This method was created by a SmartGuide.
 * @param e java.lang.Exception
 */
public void showException (Exception e) {
  try {
    getIMessageBox1().showException(e);
  } catch (java.lang.Throwable exception) {}

  return;
}
}
```

C.5 PartsContainer.java

```
package WorkShop;

/**
 * This Interface was generated by a SmartGuide.
 *
 */
public interface PartsContainer {

/**
 * This method was created by a SmartGuide.
 * @param aParameter int
 */
public void connectToDB(String systemName, String userid, String password)
                     throws Exception;
/**
 * This method was created by a SmartGuide.
 * @param aParameter int
 */
public String getRecord (String partNo, java.awt.TextField partDesc,
    java.awt.TextField partQty,
    java.awt.TextField partPrice, java.awt.TextField partDate) throws Exception;
/**
```

```
 * This method was created by a SmartGuide.
 * @param aListBox COM.taligent.widget.MultiColumnListbox
 */
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
    throws java.lang.Exception;
/**
 * This method was created by a SmartGuide.
 * @param aParameter int
 */
public String updateRecord (String partno, String partdesc, String partqty,
     (String partno, String paString partprice, String partdate) throws Exception;
}
```

C.6 StoredProcedureExample.java

```
package WorkShop;

public class StoredProcedureExample extends java.awt.Frame implements
       WorkShop.PartsContainer {
private  java.sql.CallableStatement callableStmt;
private  java.sql.Connection dbConnect;
private  WorkShop.ToolboxGUI ivjToolboxGUI1 = null;

/**
 * Constructor
 * @return java.awt.Frame
 */
public StoredProcedureExample() {
  super();
  setLayout(null);
  setBackground(new java.awt.Color(192, 192, 192));
  reshape(20, 20, 588, 437);
  setTitle("Stored Procdure Example");
  this.add("ivjToolboxGUI1", getToolboxGUI1());
}
/**
 * Connect to the AS/400 using JDBC
 * @param systemName java.lang.String
 */
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",userid,password);
  try {dbConnect.createStatement().execute("drop procedure apilib.partqry2");
  } catch (Exception e) {};
  dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY2
             (INOUT P1 INTEGER, INOUT P2 INTEGER)
             EXTERNAL NAME APILIB.SPROC2 LANGUAGE RPG GENERAL");
  callableStmt = dbConnect.prepareCall("CALL APILIB.PARTQRY2(?, ?)");

  return;

}
/**
 * This method was created by a SmartGuide.
 */
public void dispose () {
  try {
    callableStmt.close();
    dbConnect.close();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
```

```
  super.dispose();
return;
}
/**
 * Retrieve a single record, using partNo as the key.
 * @param partNo java.lang.String
 * @param partDesc java.awt.TextField
 * @param partQty java.awt.TextField
 * @param partPrice java.awt.TextField
 * @param partDate java.awt.TextField
 */
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));
    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("");
    partPrice.setText("");
    partDate.setText("");
    return "Record not found.";
  }
  return "Record found.";}

/**
 * Return the ToolboxGUI1 property value.
 * @return WorkShop.ToolboxGUI
 */
private WorkShop.ToolboxGUI getToolboxGUI1() {
  if (ivjToolboxGUI1 == null) {
    try {
      ivjToolboxGUI1 = new WorkShop.ToolboxGUI();
      ivjToolboxGUI1.reshape(4, 1, 584, 435);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception creating ToolboxGUI1");
    }
  };
  return ivjToolboxGUI1;
}
/**
 * Method to handle old AWT events
 * @return boolean
 * @param evt java.awt.Event
 */
public boolean handleEvent(java.awt.Event evt) {
  if (evt.id == java.awt.Event.WINDOW_DESTROY) {
    dispose();
    if (getParent() == null) {
      java.lang.System.exit(0);
    };
    return true;
  }
```

```
  return super.handleEvent(evt);
}
/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(java.lang.String[] args) {
  try {
      WorkShop.StoredProcedureExample aStoredProcedureExample =
         new WorkShop.StoredProcedureExample();
      aStoredProcedureExample.show(true);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception occurred in main() of StoredProcedureExample");
    }
}
/**
 * This method was created by a SmartGuide.
 * @param aListBox com.taligent.widget.MultiColumnListbox
 */
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
              throws java.lang.Exception {
  java.sql.ResultSet rs = null;
  String[] array = new String[5];

  callableStmt.setInt(1, 2);
  callableStmt.setInt(2, 0);

  rs = callableStmt.executeQuery();
  while (rs.next()) {

    array[0] = rs.getString(1);
    array[1] = rs.getString(2);
    array[2] = Integer.toString(rs.getInt(3));
    array[3] = "$" + rs.getBigDecimal(4, 2).toString();
    array[4] = rs.getDate(5).toString();

    aListBox.addRow(array);
  }

  return;
}
/**
 * Update the part record with values pass in parameters.  partno is the key field.
 * @return java.lang.String
 * @param partno java.lang.String
 * @param partdesc java.lang.String
 * @param partqty java.lang.String
 * @param partprice java.lang.String
 * @param partdate java.lang.String
 */
public String updateRecord (String partno, String partdesc, String partqty,
          String partprice, String partdate) throws Exception {

  return "Update is not implemented via stored procedure.  Record not updated.";
}
}
```

C.7 DPCExample.java

```
package WorkShop;

import COM.ibm.as400.access.*;
public class DPCExample extends java.awt.Frame implements PartsContainer {
private      WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
private      AS400 as400;
```

```
private      ProgramCall pgm;
private      String progName = "/QSYS.LIB/APILIB.LIB/DPCXRPG.PGM";

/**
 * Constructor
 * @return java.awt.Frame
 */
public DPCExample() {
  super();
  setLayout(null);
  setBackground(new java.awt.Color(192, 192, 192));
  reshape(20, 20, 578, 433);
  setTitle("Distributed Program Call Example");
  this.add("ivjToolboxGUI1", getToolboxGUI1());
}
/**
 * Connect to the AS/400 using AS400 object
 * @param systemName java.lang.String
 */
public void connectToDB(String systemName, String userid, String password)
       throws Exception {
  as400 = new COM.ibm.as400.access.AS400(systemName, userid, password);
  as400.connectService(AS400.COMMAND);
  pgm = new ProgramCall(as400);
  return;

}
/**
 * This method was created by a SmartGuide.
 */
public void dispose () {
  try {
    as400.disconnectAllServices();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  super.dispose();
return;
}
/**
 * Retrieve a single record, using partNo as the key.
 * @param partNo java.lang.String
 * @param partDesc java.awt.TextField
 * @param partQty java.awt.TextField
 * @param partPrice java.awt.TextField
 * @param partDate java.awt.TextField
 */
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
  ProgramParameter[] parmlist = new ProgramParameter[6];
// First parameter is to input action
  AS400Text asFlag = new AS400Text(1);
  parmlist[0] = new ProgramParameter( asFlag.toBytes("S") , 1);
// Second parameter is to input PartNo
  AS400PackedDecimal asPartNo = new AS400PackedDecimal(5,0);
  parmlist[1] = new ProgramParameter( asPartNo.toBytes
          (new java.math.BigDecimal(partNo)) ,3);
// Third parm is output description
  parmlist[2] = new ProgramParameter(25);
// Fourth parm is quantity
  parmlist[3] = new ProgramParameter(3);
// Fifth parm is price
  parmlist[4] = new ProgramParameter(4);
// Sixth parm is date
```

```
  parmlist[5] = new ProgramParameter(10);

// Set the program name and parameter list
  pgm.setProgram( progName, parmlist );

// Run the program
  if (pgm.run() != true) {

    // Note that there was an error
    System.out.println( "program failed:" + progName );
    // Show the messages
    AS400Message[] messagelist = pgm.getMessageList();
    for (int i=0; i  messagelist.length; i++)  {
      // show each message
      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].
          OutputData(),0));
      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));
    }
    else {
      partDesc.setText("");
      partQty.setText("");
      partPrice.setText("");
      partDate.setText("");
      return "Record not found.";
    }
  }

  return "Record found.";
}
/**
 * Return the ToolboxGUI1 property value.
 * @return WorkShop.ToolboxGUI
 */
private WorkShop.ToolboxGUI getToolboxGUI1() {
  if (ivjToolboxGUI1 == null) {
    try {
      ivjToolboxGUI1 = new WorkShop.ToolboxGUI();
      ivjToolboxGUI1.reshape(2, 1, 571, 428);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception creating ToolboxGUI1");
    }
  };
  return ivjToolboxGUI1;
}
/**
 * Method to handle old AWT events
 * @return boolean
 * @param evt java.awt.Event
 */
public boolean handleEvent(java.awt.Event evt) {
  /* Note: Changes to this method will not be overwritten when code is re-generated */
  if (evt.id == java.awt.Event.WINDOW_DESTROY) {
    dispose();
```

```
    if (getParent() == null) {
      java.lang.System.exit(0);
    };
    return true;
  }
  return super.handleEvent(evt);
}
/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(java.lang.String[] args) {
  /* Note: Changes to this method will not be overwritten when code is re-generated */
  try {
      WorkShop.DPCExample aDPCExample = new WorkShop.DPCExample();
      aDPCExample.show(true);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception occurred in main() of DPCExample");
    }
}
/**
 * This method was created by a SmartGuide.
 * @param aListBox com.taligent.widget.MultiColumnListbox
 */
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
                             throws java.lang.Exception {
  String[] array = new String[5];

// Setup the parameters
  ProgramParameter[] parmlist = new ProgramParameter[6];
// First parameter is to input action
  AS400Text asFlag = new AS400Text(1);
  parmlist[0] = new ProgramParameter( asFlag.toBytes("A") , 1);
// Second parameter is to output PartNo
  parmlist[1] = new ProgramParameter(3);
// Third parm is output description
  parmlist[2] = new ProgramParameter(25);
// Fourth parm is quantity
  parmlist[3] = new ProgramParameter(3);
// Fifth parm is price
  parmlist[4] = new ProgramParameter(4);
// Sixth parm is date
  parmlist[5] = new ProgramParameter(10);

// Set the program name and parameter list
  pgm.setProgram( progName, parmlist );

  String flag = null;

  if (pgm.run() != true) {
    // Note that there was an error
    System.out.println( "program failed:" + progName );
    // Show the messages
    AS400Message[] messagelist = pgm.getMessageList();
    for (int i=0; i  messagelist.length; i++)  {
      // show each message
      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) {

          // Note that there was an error
          System.out.println( "program failed:" + progName );
          // Show the messages
          AS400Message[] messagelist = pgm.getMessageList();
          for (int i=0; i  messagelist.length; i++)  {
          // show each message
          System.out.println( messagelist[i] );
          }
        return;
        }
        else {
          flag = (String)(asFlag.toObject(parmlist[0].getOutputData(),0));

          if (flag.equals("Y")) {
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] =((java.math.BigDecimal)(new AS400PackedDecimal(5,0)).toObject(parmlist[3].
              getOutputData(),0)).toString();
array[3] = "$" + ((java.math.BigDecimal)(new AS400PackedDecimal(6,2)).toObject(parmlist[4]
              getOutputData(),0)).toString();
array[4] =(String)(new AS400Text(10)).toObject(parmlist[5].getOutputData(),0);

            aListBox.addRow(array);
          }
        }
      } while (flag.equals("Y"));
    }
  }

  return;
}
}
```

C.8 DataQueueExample.java

```
package WorkShop;
import COM.ibm.as400.access.*;
public class DataQueueExample extends java.awt.Frame implements WorkShop.PartsContainer {
private  COM.ibm.as400.access.AS400 as400;
private  COM.ibm.as400.access.DataQueue dqInput;
private  COM.ibm.as400.access.DataQueue dqOutput;
private  WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
private  COM.ibm.as400.access.RecordFormat rfInput;
private  COM.ibm.as400.access.RecordFormat rfOutput;

/**
 * Constructor
 * @return java.awt.Frame
 */
public DataQueueExample() {
  super();
  setLayout(null);
  setBackground(new java.awt.Color(192, 192, 192));
  reshape(20, 20, 588, 452);
  setTitle("DataQueue Example");
  this.add("ivjToolboxGUI1", getToolboxGUI1());
}
/**
 * Connect to the AS/400 using JDBC
```

```
 * @param systemName java.lang.String
 */
public void connectToDB(String systemName, String userid, String password)
                                throws Exception {
as400 = new COM.ibm.as400.access.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");

  return;

}
/**
 * This method was created by a SmartGuide.
 */
public void dispose () {
  try {
    as400.disconnectAllServices();
    System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};
  super.dispose();
return;
}
/**
 * Retrieve a single record, using partNo as the key.
 * @param partNo java.lang.String
 * @param partDesc java.awt.TextField
 * @param partQty java.awt.TextField
 * @param partPrice java.awt.TextField
 * @param partDate java.awt.TextField
 */
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();
// set values of the input record ...
  Record rInput = rfInput.getNewRecord();
  rInput.setField("flag","S");
  rInput.setField("partno",new java.math.BigDecimal(partNo));

  dqInput.write(rInput.getContents());
  DataQueueEntry dqe = dqOutput.read();

  Record rOutput = rfOutput.getNewRecord(dqe.getByteData());

  if (((String)rOutput.getField("flag")).equals("Y")) {
    partDesc.setText((String)rOutput.getField("partds"));
    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("");
    partPrice.setText("");
    partDate.setText("");
    return "Record not found.";
  }
  return "Record found.";
}
/**
 * Return the ToolboxGUI1 property value.
 * @return WorkShop.ToolboxGUI
 */
private WorkShop.ToolboxGUI getToolboxGUI1() {
```

```
  if (ivjToolboxGUI1 == null) {
    try {
      ivjToolboxGUI1 = new WorkShop.ToolboxGUI();
      ivjToolboxGUI1.reshape(2, 3, 585, 446);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception creating ToolboxGUI1");
    }
  };
  return ivjToolboxGUI1;
}
/**
 * Method to handle old AWT events
 * @return boolean
 * @param evt java.awt.Event
 */
public boolean handleEvent(java.awt.Event evt) {
  /* Note: Changes to this method will not be overwritten when code is re-generated */
  if (evt.id == java.awt.Event.WINDOW_DESTROY) {
    dispose();
    if (getParent() == null) {
      java.lang.System.exit(0);
    };
    return true;
  }
  return super.handleEvent(evt);
}
/**
 * This method was created by a SmartGuide.
 */
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");

// set up the input record format....
  rfInput = new RecordFormat();
  rfInput.addFieldDescription(asFlag);
  rfInput.addFieldDescription(asPartNo);
// set up the output record format....
  rfOutput = new RecordFormat();
  rfOutput.addFieldDescription(asFlag);
  rfOutput.addFieldDescription(asPartNo);
  rfOutput.addFieldDescription(asPartDS);
  rfOutput.addFieldDescription(asPartQy);
  rfOutput.addFieldDescription(asPartPR);
  rfOutput.addFieldDescription(asPartDt);

  return;
}
/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(java.lang.String[] args) {
  /* Note: Changes to this method will not be overwritten when code is re-generated */
  try {
      WorkShop.DataQueueExample aDataQueueExample = new WorkShop.DataQueueExample();
```

```
      aDataQueueExample.show(true);
    } catch (java.lang.Throwable exception) {
      System.err.println("Exception occurred in main() of DataQueueExample");
    }
}
/**
 * This method was created by a SmartGuide.
 * @param aListBox com.taligent.widget.MultiColumnListbox
 */
public void populateListBox (com.taligent.widget.MultiColumnListbox aListBox)
                             throws java.lang.Exception {
  String[] array = new String[5];

  if (rfInput == null) initRecordFormat();

// set values of the input record ...
  Record rInput = rfInput.getNewRecord();
  rInput.setField("flag","A");

  dqInput.write(rInput.getContents());
  String flag = null;
  do {

    Record rOutput = rfOutput.getNewRecord(dqOutput.read().getByteData());
    flag = (String)rOutput.getField("flag");

    if (flag.equals("Y")) {
      array[0] =((java.math.BigDecimal)rOutput.getField("partno")).toString();
      array[1] =(String)rOutput.getField("partds");
      array[2] =((java.math.BigDecimal)rOutput.getField("partqy")).toString();
      array[3] = "$" + ((java.math.BigDecimal)rOutput.getField("partpr")).toString();
      array[4] =(String)rOutput.getField("partdt");

      aListBox.addRow(array);
    }
  } while (flag.equals("Y"));

  return;
}
}
```

C.9 RLExample.java

```
import java.math.*;
import java.io.*;
import java.util.*;
import COM.ibm.as400.access.*;
public class RLExample extends java.awt.Frame implements
   java.awt.event.WindowListener, WorkShop.PartsContainer {
private AS400 as400;
private WorkShop.ToolboxGUI ivjToolboxGUI1 = null;
private KeyedFile myKeyedFile;
}


public void connectToDB(String systemName, String userid, String password) throws Exception {
 as400 = new COM.ibm.as400.access.AS400(systemName, userid, password);
 QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB",
               "PARTS",
               "*FILE",
               "MBR");
 myKeyedFile = new KeyedFile(as400, fileName.getPath());
 try{
    as400.connectService(AS400.RECORDACCESS);}
    catch(Exception e){
```

```
   System.out.println("Unable to connect");
   System.exit(0);
   }
  RecordFormat partsFormat = null;
  try
  {
   AS400FileRecordDescription recordDescription = new
   AS400FileRecordDesccription(as400, "/QSYS.LIB/APILIB.LIB/PARTS.FILE");
   partsFormat = recordDescription.retrieveRecordFormat()[0];
    // Indicate that PARTNO is a key field
    partsFormat.addKeyFieldDescription("PARTNO");
  }
  catch(Exception e)
  {
    System.out.println("Unable to retrieve record format");
    e.printStackTrace();
    System.exit(0);
  }
try{
 myKeyedFile.setRecordFormat(partsFormat);
        // Open the file.
 myKeyedFile.open(AS400File.READ_WRITE, 0,
     AS400File.COMMIT_LOCK_LEVEL_NONE);}
 catch(Exception e){
    System.out.println("Unable to open file");
    System.exit(0);
 }
  return;
  }


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);

        // Read the first record matching theKey
  Record data = myKeyedFile.read(theKey);

        // If the record was not found, null is returned.
  if (data != null)
  {
  partDesc.setText((String)data.getField("PARTDS"));
  partQty.setText(((BigDecimal)data.getField("PARTQY")).toString());
  partPrice.setText("$" + ((BigDecimal)data.getField("PARTPR")).toString(
  partDate.setText((String)data.getField("PARTDT"));
  return "Record found.";

  }
else {
 partDesc.setText("");
   partQty.setText("");
   partPrice.setText("");
   partDate.setText("");
   return "Record not found.";
}
}

public void populateListBox(com.taligent.widget.MultiColumnListbox
       aListBox) throws Exception {

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

```
   {

  // Display each record in the file

  Record record = myKeyedFile.readFirst();
  while (record != null)
  {
    array[0] =((BigDecimal)record.getField("PARTNO")).toString();
    array[1] =(String)record.getField("PARTDS");
    array[2] =((java.math.BigDecimal)record.getField("PARTQY")).toString();
    array[3]="$"+((BigDecimal)record.getField("PARTPR")).toString();
    array[4] =(String)record.getField("PARTDT");

    aListBox.addRow(array);
    record = myKeyedFile.readNext();

  }
 }
  catch(Exception e){
  System.out.println("unable to get all");
  System.exit(0);
 }
  return;}

public void dispose() {
 try {
 // All done with the file
   myKeyedFile.close();
 as400.disconnectAllServices();
 System.out.println("Disconnected from AS/400 OK.");
  } catch (Exception e) {};

  super.dispose();
  System.exit(0);
return;
}
```

Appendix D. Internet Shopping Applet Code Listings

This apppendix has the following listings:

SelectedItems.java ItemsDb.java

ItemsDbBeanInfo.java

ToolboxApplet.java

CartApplet.java

StatusApplet.java

MyListbox.java

MyImage.java

D.1 SelectedItems.java

```
package ToolboxApplet;
import java.util.*;
import java.math.BigDecimal;
/**
* This Class was generated by a SmartGuide.
*
*/
public class SelectedItems extends java.lang.Object {
private static Vector wanted;
static BigDecimal totalAmount;
/**
* This method was created by a SmartGuide.
* @param row java.lang.Object[]
* @param key java.lang.Object
*/
public void addSelectedRow (Object[] row) {
getVector().addElement(row);
if(totalAmount==null) totalAmount=new BigDecimal("0");
totalAmount=totalAmount.add(new BigDecimal((String)row[2]));
return;
}
/**
* This method was created by a SmartGuide.
*/
public void clear () {
wanted.removeAllElements();
return;
}
/**
* This method was created by a SmartGuide.
* @return java.util.Vector
*/
public Vector getVector () {
if(wanted==null) wanted=new Vector();
return wanted;
```

```
}
}
```

D.2 ItemsDb.java

```
package ToolboxApplet;
import java.math.*;
import java.util.*;
/**
* This Class was generated by a SmartGuide.
*
*/
public class ItemsDb extends java.lang.Object {
private java.sql.Connection dbConnect;
private java.sql.PreparedStatement psItem;
private java.sql.PreparedStatement psItemRange;
private java.sql.PreparedStatement psCustomerDb;
private java.sql.PreparedStatement psQuantityInHand;
private java.sql.Statement sGetInetOrderNo;
private String systemName = new String("SYSASM02");
private String userid = new String("UUUUUUUU");
private String password = new String("PPPPPP");
private java.sql.ResultSet rs = null;
public String itemId;
public String itemName;
public BigDecimal itemPriceBigDecimal;
public String itemPrice;
public String itemInfo;
public Object[] row;
public String validCustomerId = null;
protected java.util.Vector aConnectionListener = null;
/**
* Add a COM.ibm.ivj.eab.data.ConnectionListener.
*/
public void
addConnectionListener(COM.ibm.ivj.eab.data.ConnectionListener
newListener) {
if (aConnectionListener == null) {
aConnectionListener = new java.util.Vector();
};
aConnectionListener.addElement(newListener);
}
/**
* This method was created by a SmartGuide.
* @return java.lang.Object[]
* @param OrderIdString java.lang.String
*/
public Vector checkOrderStatus (String orderIdString) {
// return Vector contains CustomerLastName,
CustomerFirstName,Object[],Object[],...
// where Object[] is OrderLineDetail =
[ItemId,ItemName,QtyOrdered,Amount]
Vector orderStatus=new Vector();
if(orderIdString.length()9 ]] orderIdString.length()==0)
return null;
try{
sGetInetOrderNo = dbConnect.createStatement();
rs=sGetInetOrderNo.executeQuery("SELECT OCID,OLINES FROM
CSDB/ORDERS WHERE OWID='0001' AND ODID=001 AND OID="+orderIdString);
rs.next();
String customerId=rs.getString("OCID");
BigDecimal orderLines=rs.getBigDecimal("OLINES",0);
rs=sGetInetOrderNo.executeQuery("SELECT CFIRST,CLAST FROM
CSDB/CSTMR WHERE CWID='0001' AND CDID=001 AND CID='"+customerId+"'");
rs.next();
```

```
String lastName=rs.getString("CLAST");
String firstName=rs.getString("CFIRST");
orderStatus.addElement(lastName);
orderStatus.addElement(firstName);
// Loop for every Order Line Detail
for(int i=1;i = orderLines.intValue();i++) {
Object[] detail=new Object[4];
rs=sGetInetOrderNo.executeQuery("SELECT OLIID,OLQTY,OLAMNT FROM
CSDB/ORDLIN WHERE OLWID='0001' AND OLDID=001 AND
OLOID="+orderIdString+" AND OLNBR="+i);
rs.next();
String itemId=rs.getString("OLIID");
BigDecimal quantity=rs.getBigDecimal("OLQTY",0);
BigDecimal amount=rs.getBigDecimal("OLAMNT",2);
rs=sGetInetOrderNo.executeQuery("SELECT INAME FROM CSDB/ITEM WHERE
IID='"+itemId+"'");
rs.next();
String itemName=rs.getString("INAME");
detail[0]=itemName;
detail[1]=quantity.toString();
detail[2]=amount.toString();
detail[3]=itemId+".GIF";
orderStatus.addElement(detail);
}
System.out.println("CheckOrd: Closed");
sGetInetOrderNo.close();
} catch( Exception e) {
System.out.println("checkOrderStatus: "+e);
return null;}
return orderStatus;
}
/**
* This method was created by a SmartGuide.
*/
public String confirmOrder (SelectedItems cart) {
// Hasn't consider multiple users environment.
// Should consider locking of records, triggers, or stored
Procedure
BigDecimal inetYTD;
BigDecimal nextOrderNo;
String updateString;
Date date;
String sdate;
String stime;
try{
// Any thing in Cart ?
if (cart.getVector()==null) return "Nothing in Cart";
Enumeration enum=cart.getVector().elements();
// Get next Order No. and the Inet YTD Balance
sGetInetOrderNo = dbConnect.createStatement();
rs=sGetInetOrderNo.executeQuery("SELECT DYTD,DNXTOR FROM
CSDB/DSTRCT WHERE DID=001 AND DWID='0001'");
rs.next();
inetYTD=rs.getBigDecimal("DYTD",2);
nextOrderNo=rs.getBigDecimal("DNXTOR",0);
// Add Inet YTD Balance.
inetYTD=inetYTD.add(cart.totalAmount).setScale(2);
// Increment OrderNo. by 1
updateString="UPDATE CSDB/DSTRCT SET DYTD = "+inetYTD+", DNXTOR =
"+ nextOrderNo.add(new BigDecimal("1")).setScale(0)+" WHERE DID=001
AND DWID='0001'";
sGetInetOrderNo.executeUpdate(updateString);
// Insert Order Line
// get the Current Date(8 chars) and Time (6 chars)
date=new Date();
sdate="19"+((date.getYear()10)?
```

```
("0"+date.getYear()):Integer.toString(date.getYear()) )
+((date.getMonth()10)?
("0"+(date.getMonth()+1)):Integer.toString((date.getMonth()+1)) )
+((date.getDate()10)?
("0"+date.getDate()):Integer.toString(date.getDate()) );
stime=((date.getHours()10)?
("0"+date.getHours()):Integer.toString(date.getHours()) )
+((date.getMinutes()10)?
("0"+date.getMinutes()):Integer.toString(date.getMinutes()) )
+((date.getSeconds()10)?
("0"+date.getSeconds()):Integer.toString(date.getSeconds()) );
updateString="INSERT INTO CSDB/ORDERS (OWID, ODID, OCID, OID,
OENTDT, OENTTM, OLINES, OLOCAL) VALUES ('0001',001,'"
+validCustomerId+"',"+nextOrderNo+","+sdate+","+stime+","+cart.getVector().size()+",2)";
sGetInetOrderNo.executeUpdate(updateString);
// Update Stock Info and Insert Order Line Details one by one.
int orderLineNo=1;
while(enum.hasMoreElements()) {
Object[] element=((Object[])enum.nextElement());
// Add Order Line
updateString="INSERT INTO CSDB/ORDLIN
(OLOID,OLDID,OLWID,OLNBR,OLSPWH,OLIID,OLQTY,OLAMNT,OLDSTI) VALUES ("
+nextOrderNo+",001, '0001',"+orderLineNo+",'0001','"
+element[0]+"',"+"001,"+element[2]+",'Internet Applet Order')";
sGetInetOrderNo.executeUpdate(updateString);
orderLineNo+=1;
// Decrease Qty in Hand in Stock
rs=sGetInetOrderNo.executeQuery("SELECT STQTY FROM CSDB/STOCK
WHERE STWID='0001' AND STIID='"+element[0]+"'");
rs.next();
BigDecimal quantityInHand=rs.getBigDecimal("STQTY",0).subtract(new
BigDecimal("1"));
updateString="UPDATE CSDB/STOCK SET
STQTY="+quantityInHand.toString()
+", STYTD="+sdate+" WHERE STWID='0001' AND
STIID='"+element[0]+"'";
sGetInetOrderNo.executeUpdate(updateString);
};
// CleanUp Selected Items and Clear Listbox in Cart.
cart.clear();
// Release resources
sGetInetOrderNo.close();
return "Order No.:"+nextOrderNo+"\n (Remember this Order Number
for Checking Order Status)";
} catch(Exception e)
{System.out.println("confirmOrder:"+e);e.printStackTrace();}
return "Exception Occured, check Java Console";
}
/**
* This method was created by a SmartGuide.
* @return java.lang.String
*/
public String connect () {
try{
Class.forName("COM.ibm.as400.access.AS400JDBCDriver");
dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" +
systemName +
"/csdb;naming=system;errors=full;date
format=iso",userid,password);
psItem = dbConnect.prepareStatement("SELECT * FROM CSDB/ITEM WHERE
IID = ?");
psItemRange = dbConnect.prepareStatement("SELECT * FROM CSDB/ITEM
WHERE IID = ? AND IID = ?");
psCustomerDb = dbConnect.prepareStatement("SELECT CID FROM
CSDB/CSTMR WHERE CID = ? AND CDID=001 AND CWID='0001'");
psQuantityInHand = dbConnect.prepareStatement("SELECT STQTY FROM
```

```
CSDB/STOCK WHERE STWID= '0001' AND STIID=?");
} catch (Exception e) {
System.out.println("connect(): "+e);
e.printStackTrace();
return "Connect: "+e;}
fireConnected(new COM.ibm.ivj.eab.data.ConnectionEvent(this," "));
return "Connect Successfully";
}
/**
* This method was created by a SmartGuide.
*/
public void disconnect () throws Exception {
dbConnect.close();
psItem.close();
psItemRange.close();
psCustomerDb.close();
psQuantityInHand.close();
fireDisconnected(new COM.ibm.ivj.eab.data.ConnectionEvent(this));
return;
}
/**
* This method was created by a SmartGuide.
* @return ToolboxApplet.ItemsDb
*/
public ItemsDb fetchNextItem () {
try {
if (rs.next()) {
itemId=rs.getString("IID");
itemName=rs.getString("INAME");
itemPriceBigDecimal=rs.getBigDecimal("IPRICE",2);
itemPrice=itemPriceBigDecimal.toString();
itemInfo=rs.getString("IDATA");
}
else {
itemId=null;
itemName=null;
itemPriceBigDecimal=null;
itemPrice=null;
itemInfo=null;
}
} catch (Exception e) {System.out.println("fetchnext fail: "+e);}
return this;
}
/**
* This method was created by a SmartGuide.
*/
protected void finalize() {
try { disconnect();super.finalize(); } catch(Throwable t)
{System.out.println(t);}
return;
}
/**
* Fire (signal) the cancelled event.
*/
protected void fireCancelled(COM.ibm.ivj.eab.data.ConnectionEvent
arg1) {
if (aConnectionListener == null) {
return;
};
int currentSize = aConnectionListener.size();
COM.ibm.ivj.eab.data.ConnectionListener tempListener = null;
for (int index = 0; index  currentSize; index++){
tempListener =
(COM.ibm.ivj.eab.data.ConnectionListener)aConnectionListener.elementAt(index);
if (tempListener != null) {
tempListener.cancelled(arg1);
```

```
};
};
}
/**
* Fire (signal) the committed event.
*/
protected void fireCommitted(COM.ibm.ivj.eab.data.ConnectionEvent
arg1) {
if (aConnectionListener == null) {
return;
};
int currentSize = aConnectionListener.size();
COM.ibm.ivj.eab.data.ConnectionListener tempListener = null;
for (int index = 0; index  currentSize; index++){
tempListener =
(COM.ibm.ivj.eab.data.ConnectionListener)aConnectionListener.elementAt(index);
if (tempListener != null) {
tempListener.committed(arg1);
};
};
}
/**
* Fire (signal) the connected event.
*/
protected void fireConnected(COM.ibm.ivj.eab.data.ConnectionEvent
arg1) {
if (aConnectionListener == null) {
return;
};
int currentSize = aConnectionListener.size();
COM.ibm.ivj.eab.data.ConnectionListener tempListener = null;
for (int index = 0; index  currentSize; index++){
tempListener =
(COM.ibm.ivj.eab.data.ConnectionListener)aConnectionListener.elementAt(index);
if (tempListener != null) {
tempListener.connected(arg1);
};
};
}
/**
* Fire (signal) the disconnected event.
*/
protected void
fireDisconnected(COM.ibm.ivj.eab.data.ConnectionEvent arg1) {
if (aConnectionListener == null) {
return;
};
int currentSize = aConnectionListener.size();
COM.ibm.ivj.eab.data.ConnectionListener tempListener = null;
for (int index = 0; index  currentSize; index++){
tempListener =
(COM.ibm.ivj.eab.data.ConnectionListener)aConnectionListener.elementAt(index);
if (tempListener != null) {
tempListener.disconnected(arg1);
};
};
}
/**
* Fire (signal) the executedSQL event.
*/
protected void
fireExecutedSQL(COM.ibm.ivj.eab.data.ConnectionEvent arg1) {
if (aConnectionListener == null) {
return;
};
int currentSize = aConnectionListener.size();
```

```
COM.ibm.ivj.eab.data.ConnectionListener tempListener = null;
for (int index = 0; index  currentSize; index++){
tempListener =
(COM.ibm.ivj.eab.data.ConnectionListener)aConnectionListener.elementAt(index);
if (tempListener != null) {
tempListener.executedSQL(arg1);
};
};
}
/**
* Fire (signal) the rolledback event.
*/
protected void fireRolledback(COM.ibm.ivj.eab.data.ConnectionEvent
arg1) {
if (aConnectionListener == null) {
return;
};
int currentSize = aConnectionListener.size();
COM.ibm.ivj.eab.data.ConnectionListener tempListener = null;
for (int index = 0; index  currentSize; index++){
tempListener =
(COM.ibm.ivj.eab.data.ConnectionListener)aConnectionListener.elementAt(index);
if (tempListener != null) {
tempListener.rolledback(arg1);
};
};
}
/**
* This method was created by a SmartGuide.
* @return ToolboxApplet.ItemsDb
* @param itemno java.lang.String
*/
public ItemsDb getItem (String itemno) {
try {
psItem.setString(1, itemno);
rs = psItem.executeQuery();
fetchNextItem();
} catch (Exception e) {System.out.println("getItem fail: "+e);}
return this;
}
/**
* This method was created by a SmartGuide.
* @return ToolboxApplet.ItemsDb
* @param partnoMin java.lang.String
* @param partnoMax java.lang.String
*/
public ItemsDb getItems (String itemnoMin, String itemnoMax) {
if(itemnoMax.length()==0) {
getItem(itemnoMin);
}
else {
try {
psItemRange.setString(1, itemnoMin);
psItemRange.setString(2, itemnoMax);
rs = psItemRange.executeQuery();
} catch (Exception e) {System.out.println("getItemS fail: "+e);}
}
return this;
}
/**
* This method was created by a SmartGuide.
* @return java.lang.String
*/
public String getPassword () {
return password;
}
```

```
/**
* This method was created by a SmartGuide.
* @return java.lang.String
*/
public String getSystemName () {
return systemName;
}
/**
* This method was created by a SmartGuide.
* @return java.lang.String
*/
public String getUserId () {
return userid;
}
/**
* This method was created by a SmartGuide.
* @return java.math.BigDecimal
* @param itemNo java.lang.String
*/
public BigDecimal quantityInHand (String itemNo) {
try{
// Get next Order No. and the Inet YTD Balance
psQuantityInHand.setString(1,itemNo);
rs=psQuantityInHand.executeQuery();
rs.next();
return rs.getBigDecimal("STQTY",0);
} catch(Exception e) {System.out.println(e);}
return null;
}
/**
* Add a COM.ibm.ivj.eab.data.ConnectionListener.
*/
public void
removeConnectionListener(COM.ibm.ivj.eab.data.ConnectionListener
newListener) {
if (aConnectionListener != null) {
aConnectionListener.removeElement(newListener);
};
}
/**
* This method was created by a SmartGuide.
* @param tPassword java.lang.String
*/
public void setPassword (String tPassword) {
password=tPassword;
return;
}
/**
* This method was created by a SmartGuide.
* @param sysname java.lang.String
*/
public void setSystemName(String tSysname) {
systemName=tSysname;
return;
}
/**
* This method was created by a SmartGuide.
* @param tUserId java.lang.String
*/
public void setUserId (String tUserId) {
userid=tUserId;
return;
}
/**
* This method was created by a SmartGuide.
* @return java.lang.String
```

```
*/
public String toString () {
// for debugging purpose
return "["+itemId+"] ["+itemName+"] ["+itemPrice+"]
["+itemInfo+"]";
}
/**
* This method was created by a SmartGuide.
* @return java.lang.Boolean
* @param CustomerId java.lang.String
*/
public boolean verifyCustomer (String customerId) {
boolean isvalid=false;
try {
psCustomerDb.setString(1, customerId);
rs = psCustomerDb.executeQuery();
if(rs.next()) {isvalid=true; validCustomerId=customerId;}
} catch (Exception e) { validCustomerId=null;}
return isvalid;
}
}
```

D.3 ToolboxApplet.java

```
import itsc.taligent.util.*;
import itsc.taligent.widget.*;
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.net.URL;
import java.util.*;
public class ToolboxApplet extends java.applet.Applet implements
java.awt.event.ActionListener {
private java.lang.String imagePath = "file:/C:/mytest/solution/";
private java.awt.Button ivjButton1 = null;
private java.awt.Button ivjButton2 = null;
private ToolboxApplet.ItemsDb ivjItemsDb = null;
private java.awt.Label ivjLabel1 = null;
private java.awt.Label ivjLabel2 = null;
private itsc.taligent.widget.MyListbox ivjListbox = null;
private java.awt.TextField ivjTextField1 = null;
private java.awt.TextField ivjTextField2 = null;
static ToolboxApplet.SelectedItems selected = new SelectedItems();
/**
* Method to handle events for the ActionListener interface.
* @param e java.awt.event.ActionEvent
*/
public void actionPerformed(java.awt.event.ActionEvent e) {
if ((e.getSource() == getButton1()) ) {
conn8(e);
}
if ((e.getSource() == getButton1()) ) {
conn1(e);
}
if ((e.getSource() == getButton1()) ) {
conn12(e);
}
if ((e.getSource() == getButton2()) ) {
conn7(e);
}
if ((e.getSource() == getButton2()) ) {
```

```
conn19(e);
}
}
/**
* This method was created by a SmartGuide.
*/
public void addAllRows () {
while(getItemsDb().fetchNextItem().itemId!=null) {
addListboxRow();
getListbox().repaint();
};
return;
}
/**
* This method was created by a SmartGuide.
*/
public void addListboxRow () {
Object myObject[]=new Object[5];
myObject[0]=getItemsDb().itemId;
myObject[1]=getItemsDb().itemName;
myObject[2]=getItemsDb().itemPrice;
// URL baseURL=getCodeBase();
String imageName=getItemsDb().itemId+".GIF";
try {URL baseURL=new URL(imagePath);
myObject[3]=new
MyImage(getImage(baseURL,imageName),getListbox());}
catch(Exception e){ myObject[3]="Not Loaded";
myObject[4]=getItemsDb().itemInfo;
getListbox().addRow(myObject);
return; }
myObject[4]=getItemsDb().itemInfo;
getListbox().addRow(myObject);
return;
}
/**
* conn1: (Button1.action.actionPerformed --
ItemsDb.getItems(java.lang.String, java.lang.String))
* @return ToolboxApplet.ItemsDb
* @param e java.awt.event.ActionEvent
*/
private ToolboxApplet.ItemsDb conn1(java.awt.event.ActionEvent e)
{
ToolboxApplet.ItemsDb conn1Result = null;
try {
conn1Result = getItemsDb().getItems(getTextField1().getText(),
getTextField2().getText());
} catch (java.lang.Throwable exception) {
}
return conn1Result;
}
/**
* conn12: (Button1.action.actionPerformed --
ToolboxApplet.addAllRows()V)
* @param e java.awt.event.ActionEvent
*/
private void conn12(java.awt.event.ActionEvent e) {
try {
this.addAllRows();
} catch (java.lang.Throwable exception) {
```

```
}
}
/**
* conn16: (ToolboxApplet.init() -- ToolboxApplet.MyInit()V)
*/
private void conn16() {
try {
this.MyInit();
} catch (java.lang.Throwable exception) {
}
}
/**
* conn19: (Button2.action.actionPerformed --
Listbox.deselectAllRows())
* @return itsc.taligent.widget.MultiColumnListbox
* @param e java.awt.event.ActionEvent
*/
private itsc.taligent.widget.MultiColumnListbox
conn19(java.awt.event.ActionEvent e) {
itsc.taligent.widget.MultiColumnListbox conn19Result = null;
try {
conn19Result = getListbox().deselectAllRows();
} catch (java.lang.Throwable exception) {
}
return conn19Result;
}
/**
* conn7: (Button2.action.actionPerformed --
ToolboxApplet.getSelectedIndexes()Ljava.lang.String;)
* @return java.lang.String
* @param e java.awt.event.ActionEvent
*/
private java.lang.String conn7(java.awt.event.ActionEvent e) {
java.lang.String conn7Result = null;
try {
conn7Result = this.getSelectedIndexes();
} catch (java.lang.Throwable exception) {
}
return conn7Result;
}
/**
* conn8: (Button1.action.actionPerformed -- Listbox.clear())
* @return itsc.taligent.widget.MultiColumnListbox
* @param e java.awt.event.ActionEvent
*/
private itsc.taligent.widget.MultiColumnListbox
conn8(java.awt.event.ActionEvent e) {
itsc.taligent.widget.MultiColumnListbox conn8Result = null;
try {
conn8Result = getListbox().clear();
} catch (java.lang.Throwable exception) {
}
return conn8Result;
}
/**
* conn9: (ToolboxApplet.init() -- ItemsDb.connect())
* @return java.lang.String
*/
private java.lang.String conn9() {
```

```
java.lang.String conn9Result = null;
try {
conn9Result = getItemsDb().connect();
} catch (java.lang.Throwable exception) {
}
return conn9Result;
}
/**
* Handle the Applet destroy method.
*/
public void destroy() {
/* Handle the Applet destroy method. */
super.destroy();
}
/**
* Information about this applet.
* @return java.lang.String
*/
public java.lang.String getAppletInfo() {
return
"ToolboxApplet\n" +
"\n" +
"This applet was generated by a VisualAge SmartGuide.\n" +
"";
}
/**
* Return the Button1 property value.
* @return java.awt.Button
*/
private java.awt.Button getButton1() {
if (ivjButton1 == null) {
try {
ivjButton1 = new java.awt.Button("Query Range of Items");
ivjButton1.reshape(9, 59, 185, 28);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Button1");
}
};
return ivjButton1;
}
/**
* Return the Button2 property value.
* @return java.awt.Button
*/
private java.awt.Button getButton2() {
if (ivjButton2 == null) {
try {
ivjButton2 = new java.awt.Button("Put Selection to Cart");
ivjButton2.reshape(11, 274, 136, 30);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Button2");
}
};
return ivjButton2;
}
/**
* Return the ItemsDb property value.
* @return ToolboxApplet.ItemsDb
*/
```

```
private ToolboxApplet.ItemsDb getItemsDb() {
if (ivjItemsDb == null) {
try {
ivjItemsDb = new ToolboxApplet.ItemsDb();
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating ItemsDb");
}
};
return ivjItemsDb;
}
/**
* Return the Label1 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel1() {
if (ivjLabel1 == null) {
try {
ivjLabel1 = new java.awt.Label("Product Catalog");
ivjLabel1.setFont(new java.awt.Font("", 1, 30));
ivjLabel1.reshape(14, 6, 290, 25);
ivjLabel1.setForeground(new java.awt.Color(0, 0, 255));
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Label1");
}
};
return ivjLabel1;
}
/**
* Return the Label2 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel2() {
if (ivjLabel2 == null) {
try {
ivjLabel2 = new java.awt.Label("Select the items you want, then
put it into your cart.");
ivjLabel2.reshape(14, 33, 297, 19);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Label2");
}
};
return ivjLabel2;
}
/**
* Return the Listbox property value.
* @return itsc.taligent.widget.MyListbox
*/
private itsc.taligent.widget.MyListbox getListbox() {
if (ivjListbox == null) {
try {
ivjListbox = new itsc.taligent.widget.MyListbox();
ivjListbox.reshape(9, 89, 599, 181);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Listbox");
}
};
return ivjListbox;
}
/**
```

```
* This method was created by a SmartGuide.
* @return java.lang.String
*/
public String getSelectedIndexes() {
int[] indexes=getListbox().getSelectedIndexes();
for(int i=0;iindexes.length;i++) {
selected.addSelectedRow(getListbox().getRow(indexes[i]));
}
return "Selected:"+selected.getVector().toString();
}
/**
* Return the TextField1 property value.
* @return java.awt.TextField
*/
private java.awt.TextField getTextField1() {
if (ivjTextField1 == null) {
try {
ivjTextField1 = new java.awt.TextField("000001");
ivjTextField1.reshape(198, 59, 125, 30);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating TextField1");
}
};
return ivjTextField1;
}
/**
* Return the TextField2 property value.
* @return java.awt.TextField
*/
private java.awt.TextField getTextField2() {
if (ivjTextField2 == null) {
try {
ivjTextField2 = new java.awt.TextField("000010");
ivjTextField2.reshape(322, 59, 125, 30);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating TextField2");
}
};
return ivjTextField2;
}
/**
* Method to handle events for the Applet interface.
*/
public void init() {
/* Method to handle events for the Applet interface. */
super.init();
try {
setLayout(null);
reshape(14, 8, 615, 327);
this.add("ivjButton1", getButton1());
this.add("ivjTextField1", getTextField1());
this.add("ivjTextField2", getTextField2());
this.add("ivjButton2", getButton2());
this.add("ivjLabel1", getLabel1());
this.add("ivjLabel2", getLabel2());
this.add("ivjListbox", getListbox());
conn16();
conn9();
initConnections();
```

```
} catch (java.lang.Throwable exception) {
System.err.println("Exception occurred in init() of
ToolboxApplet");
}
}
/**
* Initializes connections
*/
private void initConnections() {
/* Initialize the connections for the part */
getButton1().addActionListener(this);
getButton2().addActionListener(this);
}
/**
* main() is executed when your applet is run standalone.
*/
public static void main (String args[]) {
//ToolboxApplet applet = new ToolboxApplet();
//uvm.applet.AppletFrame frame = new
uvm.applet.AppletFrame("Applet");
// frame.add("Center", applet);
// frame.resize(350, 250);
// frame.show();
// applet.init();
// applet.start();
}
/**
* This method was created by a SmartGuide.
*/
public void MyInit () {
String columns[] = {"Item No.","Item
Name","Price","Image","Details"};
getListbox().addColumns(columns);
getListbox().getColumnInfo(0).setWidth(100);
getListbox().getColumnInfo(1).setWidth(250);
getListbox().getColumnInfo(2).setWidth(100);
getListbox().getColumnInfo(3).setWidth(100);
getListbox().getColumnInfo(4).setWidth(100);
getListbox().setMultipleSelections(true);
getListbox().setPreferedRowHeight(50);
getListbox().reshape();
return;
}
/**
* Handle the Applet start method.
*/
public void start() {
/* Handle the Applet start method. */
super.start();
}
/**
* Handle the Applet stop method.
*/
public void stop() {
/* Handle the Applet stop method. */
super.stop();
}
}
```

D.4 CartApplet.java

```
import java.applet.*;
import java.awt.*;
import java.util.*;
public class CartApplet extends java.applet.Applet implements
java.awt.event.ActionListener, java.awt.event.KeyListener {
ToolboxApplet.SelectedItems cart = new SelectedItems();
private java.awt.Button ivjButton1 = null;
private java.awt.Button ivjButton2 = null;
private ToolboxApplet.ItemsDb ivjItemsDb = null;
private java.awt.Label ivjLabel1 = null;
private java.awt.Label ivjLabel2 = null;
private java.awt.Label ivjLabel3 = null;
private itsc.taligent.widget.MyListbox ivjListbox = null;
private COM.ibm.ivj.javabeans.IMessageBox ivjMessageBox = null;
private java.awt.TextField ivjTextField1 = null;
/**
* Method to handle events for the ActionListener interface.
* @param e java.awt.event.ActionEvent
*/
public void actionPerformed(java.awt.event.ActionEvent e) {
if ((e.getSource() == getButton1()) ) {
conn29(e);
}
if ((e.getSource() == getButton1()) ) {
conn0(e);
}
if ((e.getSource() == getButton2()) ) {
conn17(e);
}
if ((e.getSource() == getButton2()) ) {
conn9(e);
}
if ((e.getSource() == getButton2()) ) {
conn10(e);
}
}
/**
* conn0: (Button1.action.actionPerformed --
CartApplet.showCart()V)
* @param e java.awt.event.ActionEvent
*/
private void conn0(java.awt.event.ActionEvent e) {
try {
this.showCart();
} catch (java.lang.Throwable exception) {
}
}
/**
* conn1: (CartApplet.init() -- CartApplet.MyInit()V)
*/
private void conn1() {
try {
this.MyInit();
} catch (java.lang.Throwable exception) {
}
}
/**
```

```
* conn10: (Button2.action.actionPerformed -- Listbox.clear())
* @return itsc.taligent.widget.MultiColumnListbox
* @param e java.awt.event.ActionEvent
*/
private itsc.taligent.widget.MultiColumnListbox
conn10(java.awt.event.ActionEvent e) {
itsc.taligent.widget.MultiColumnListbox conn10Result = null;
try {
conn10Result = getListbox().clear();
} catch (java.lang.Throwable exception) {
}
return conn10Result;
}
/**
* conn17: (Button2.action.actionPerformed --
ItemsDb.confirmOrder(ToolboxApplet.SelectedItems))
* @return java.lang.String
* @param e java.awt.event.ActionEvent
*/
private java.lang.String conn17(java.awt.event.ActionEvent e) {
java.lang.String conn17Result = null;
try {
conn17Result = getItemsDb().confirmOrder(cart);
conn8(conn17Result);
} catch (java.lang.Throwable exception) {
}
return conn17Result;
}
/**
* conn18: (TextField1.key.keyReleased --
CartApplet.validateCustomerNo(Ljava.lang.String;Ljava.awt.Component;)V)
* @param e java.awt.event.KeyEvent
*/
private void conn18(java.awt.event.KeyEvent e) {
try {
this.validateCustomerNo(getTextField1().getText(), getButton2());
} catch (java.lang.Throwable exception) {
}
}
/**
* conn29: (Button1.action.actionPerformed -- Listbox.clear())
* @return itsc.taligent.widget.MultiColumnListbox
* @param e java.awt.event.ActionEvent
*/
private itsc.taligent.widget.MultiColumnListbox
conn29(java.awt.event.ActionEvent e) {
itsc.taligent.widget.MultiColumnListbox conn29Result = null;
try {
conn29Result = getListbox().clear();
} catch (java.lang.Throwable exception) {
}
return conn29Result;
}
/**
* conn30: (CartApplet.init() -- ItemsDb.connect())
* @return java.lang.String
*/
private java.lang.String conn30() {
java.lang.String conn30Result = null;
```

```
try {
conn30Result = getItemsDb().connect();
} catch (java.lang.Throwable exception) {
}
return conn30Result;
}
/**
* conn8: ( (Button2,action.actionPerformed --
ItemsDb,confirmOrder(ToolboxApplet.SelectedItems)).normalResult
-- MessageBox.show(java.lang.String))
* @param conn17Result java.lang.String
*/
private void conn8(java.lang.String conn17Result) {
try {
getMessageBox().show(conn17Result);
} catch (java.lang.Throwable exception) {
}
}
/**
* conn9: (Button2.action.actionPerformed -- TextField1.text)
* @param e java.awt.event.ActionEvent
*/
private void conn9(java.awt.event.ActionEvent e) {
try {
getTextField1().setText("");
} catch (java.lang.Throwable exception) {
}
}
/**
* Handle the Applet destroy method.
*/
public void destroy() {
/* Handle the Applet destroy method. */
super.destroy();
}
/**
* Information about this applet.
* @return java.lang.String
*/
public java.lang.String getAppletInfo() {
return
"CartApplet\n" +
"\n" +
"This applet was generated by a VisualAge SmartGuide.\n" +
"";
}
/**
* Return the Button1 property value.
* @return java.awt.Button
*/
private java.awt.Button getButton1() {
if (ivjButton1 == null) {
try {
ivjButton1 = new java.awt.Button("Look into Cart");
ivjButton1.reshape(7, 8, 125, 30);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Button1");
}
};
```

```
return ivjButton1;
}
/**
* Return the Button2 property value.
* @return java.awt.Button
*/
private java.awt.Button getButton2() {
if (ivjButton2 == null) {
try {
ivjButton2 = new java.awt.Button("Confirm Order");
ivjButton2.reshape(476, 249, 125, 30);
ivjButton2.setEnabled(false);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Button2");
}
};
return ivjButton2;
}
/**
* Return the ItemsDb property value.
* @return ToolboxApplet.ItemsDb
*/
private ToolboxApplet.ItemsDb getItemsDb() {
if (ivjItemsDb == null) {
try {
ivjItemsDb = new ToolboxApplet.ItemsDb();
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating ItemsDb");
}
};
return ivjItemsDb;
}
/**
* Return the Label1 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel1() {
if (ivjLabel1 == null) {
try {
ivjLabel1 = new java.awt.Label("Customer No.");
ivjLabel1.reshape(388, 218, 86, 30);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Label1");
}
};
return ivjLabel1;
}
/**
* Return the Label2 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel2() {
if (ivjLabel2 == null) {
try {
ivjLabel2 = new java.awt.Label("My Shopping Cart");
ivjLabel2.setFont(new java.awt.Font("", 1, 30));
ivjLabel2.reshape(288, 4, 313, 34);
ivjLabel2.setForeground(new java.awt.Color(0, 0, 255));
} catch (java.lang.Throwable exception) {
```

```
System.err.println("Exception creating Label2");
}
};
return ivjLabel2;
}
/**
* Return the Label3 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel3() {
if (ivjLabel3 == null) {
try {
ivjLabel3 = new java.awt.Label("Total Amount:");
ivjLabel3.reshape(6, 221, 339, 29);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Label3");
}
};
return ivjLabel3;
}
/**
* Return the Listbox property value.
* @return itsc.taligent.widget.MyListbox
*/
private itsc.taligent.widget.MyListbox getListbox() {
if (ivjListbox == null) {
try {
ivjListbox = new itsc.taligent.widget.MyListbox();
ivjListbox.reshape(8, 39, 592, 172);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Listbox");
}
};
return ivjListbox;
}
/**
* Return the MessageBox property value.
* @return COM.ibm.ivj.javabeans.IMessageBox
*/
private COM.ibm.ivj.javabeans.IMessageBox getMessageBox() {
if (ivjMessageBox == null) {
try {
ivjMessageBox = new COM.ibm.ivj.javabeans.IMessageBox();
ivjMessageBox.setTitle("Message Box");
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating MessageBox");
}
};
return ivjMessageBox;
}
/**
* Return the TextField1 property value.
* @return java.awt.TextField
*/
private java.awt.TextField getTextField1() {
if (ivjTextField1 == null) {
try {
ivjTextField1 = new java.awt.TextField(0);
ivjTextField1.reshape(477, 218, 125, 30);
```

```
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating TextField1");
}
};
return ivjTextField1;
}
/**
* Method to handle events for the Applet interface.
*/
public void init() {
/* Method to handle events for the Applet interface. */
super.init();
try {
setLayout(null);
reshape(19, 10, 612, 324);
this.add("ivjButton1", getButton1());
this.add("ivjButton2", getButton2());
this.add("ivjTextField1", getTextField1());
this.add("ivjLabel1", getLabel1());
this.add("ivjLabel3", getLabel3());
this.add("ivjLabel2", getLabel2());
this.add("ivjListbox", getListbox());
conn1();
conn30();
initConnections();
} catch (java.lang.Throwable exception) {
System.err.println("Exception occurred in init() of CartApplet");
}
}
/**
* Initializes connections
*/
private void initConnections() {
/* Initialize the connections for the part */
getButton1().addActionListener(this);
getButton2().addActionListener(this);
getTextField1().addKeyListener(this);
}
/**
* Method to handle events for the KeyListener interface.
* @param e java.awt.event.KeyEvent
*/
public void keyPressed(java.awt.event.KeyEvent e) {
}
/**
* Method to handle events for the KeyListener interface.
* @param e java.awt.event.KeyEvent
*/
public void keyReleased(java.awt.event.KeyEvent e) {
if ((e.getSource() == getTextField1()) ) {
conn18(e);
}
}
/**
* Method to handle events for the KeyListener interface.
* @param e java.awt.event.KeyEvent
*/
public void keyTyped(java.awt.event.KeyEvent e) {
}
```

```
/**
* main entrypoint - starts the part when it is run as an
application
* @param args java.lang.String[]
*/
public static void main(java.lang.String[] args) {
/* Note: Changes to this method will not be overwritten when code
is re-generated */
try {
ToolboxApplet.CartApplet aCartApplet = new
ToolboxApplet.CartApplet();
java.awt.Frame frame = new java.awt.Frame(aCartApplet.toString());
frame.add("Center", aCartApplet);
try {
java.lang.Class aClass =
java.lang.Class.forName("uvm.abt.edit.ExitButton");
frame.add("South", (java.awt.Component)aClass.newInstance());
} catch (java.lang.Throwable exception) {
}
frame.resize(aCartApplet.size());
aCartApplet.init();
aCartApplet.start();
frame.show(true);
aCartApplet.destroy();
} catch (java.lang.Throwable exception) {
System.err.println("Exception occurred in main() of CartApplet");
}
}
/**
* This method was created by a SmartGuide.
*/
public void MyInit () {
// To initialize the MultiColumnListBox
String columns[] = {"Item No.","Item Name","Price","Qty in
Stock","Image","Details"};
getListbox().addColumns(columns);
getListbox().getColumnInfo(0).setWidth(100);
getListbox().getColumnInfo(1).setWidth(250);
getListbox().getColumnInfo(2).setWidth(100);
getListbox().getColumnInfo(3).setWidth(100);
getListbox().getColumnInfo(4).setWidth(100);
getListbox().getColumnInfo(5).setWidth(100);
getListbox().setPreferedRowHeight(50);
getListbox().reshape();
return;
}
/**
* paint() draws the text on the drawing area.
*/
public void paint (java.awt.Graphics g) {
}
/**
* This method was created by a SmartGuide.
* @return java.lang.String
*/
public void showCart () {
Object myObject[]=new Object[6];
try {
if(cart.getVector()!=null) {
```

```
Enumeration enum=cart.getVector().elements();
while(enum.hasMoreElements()) {
Object[] element=((Object[])enum.nextElement());
myObject[0]=element[0]; //ItemId
myObject[1]=element[1]; //ItemName
myObject[2]=element[2]; //Price
//[4] is qty in stock.
myObject[3]=(getItemsDb().quantityInHand(((String)element[0]))).toString();
myObject[4]=element[3]; // Image
myObject[5]=element[4]; //Details
getListbox().addRow(myObject);
};
getListbox().repaint();
getLabel3().setText("Total Amount: "+cart.totalAmount.toString());
return;
}
} catch(Exception e) {e.printStackTrace();System.out.println(e); }
return;
}
/**
* Handle the Applet start method.
*/
public void start() {
/* Handle the Applet start method. */
super.start();
}
/**
* Handle the Applet stop method.
*/
public void stop() {
/* Handle the Applet stop method. */
super.stop();
}
/**
* This method was created by a SmartGuide.
* @param customerNo java.lang.String
*/
public void validateCustomerNo (String customerNo,Component
button) {
if(customerNo.length()==4) {
if(getItemsDb().verifyCustomer(customerNo))
button.enable();
}
else button.disable();
return;
}
}
```

D.5 StatusApplet.java

```
import itsc.taligent.widget.*;
import java.applet.*;
import java.awt.*;
import java.net.URL;
import java.util.*;
public class StatusApplet extends java.applet.Applet implements
java.awt.event.ActionListener {
private java.lang.String imagePath = "file:/C:/mytest/solution/";
```

```
private java.awt.Button ivjButton1 = null;
private ToolboxApplet.ItemsDb ivjItemsDb = null;
private java.awt.Label ivjLabel1 = null;
private java.awt.Label ivjLabel2 = null;
private itsc.taligent.widget.MyListbox ivjListbox = null;
private java.awt.Panel ivjPanel1 = null;
private java.awt.TextField ivjTextField1 = null;
/**
* Method to handle events for the ActionListener interface.
* @param e java.awt.event.ActionEvent
*/
public void actionPerformed(java.awt.event.ActionEvent e) {
if ((e.getSource() == getButton1()) ) {
conn13(e);
}
}
/**
* conn13: (Button1.action.actionPerformed --
StatusApplet.fillListbox(Ljava.lang.String;)V)
* @param e java.awt.event.ActionEvent
*/
private void conn13(java.awt.event.ActionEvent e) {
try {
this.fillListbox(getTextField1().getText());
} catch (java.lang.Throwable exception) {
}
}
/**
* conn24: (StatusApplet.init() -- StatusApplet.MyInit()V)
*/
private void conn24() {
try {
this.MyInit();
} catch (java.lang.Throwable exception) {
}
}
/**
* conn34: (StatusApplet.init() -- ItemsDb.connect())
* @return java.lang.String
*/
private java.lang.String conn34() {
java.lang.String conn34Result = null;
try {
conn34Result = getItemsDb().connect();
} catch (java.lang.Throwable exception) {
}
return conn34Result;
}
/**
* Handle the Applet destroy method.
*/
public void destroy() {
/* Handle the Applet destroy method. */
super.destroy();
}
/**
* This method was created by a SmartGuide.
* @param orderStatus java.util.Vector
*/
```

```
public void fillListbox (String orderId) {
Vector orderStatus=getItemsDb().checkOrderStatus(orderId);
if(orderStatus==null) {
getLabel2().setText("Order No. "+orderId+" Not Found !!!");
return; }
Enumeration detailLine=orderStatus.elements();
String lastName=((String)detailLine.nextElement());
String firstName=((String)detailLine.nextElement());
getLabel2().setText("Order "+orderId+" was Ordered by
"+firstName+" "+lastName);
while(detailLine.hasMoreElements()) {
Object[] detail= ((Object[])detailLine.nextElement());
String imageString=((String)detail[3]);
try {
URL baseURL=new URL(imagePath);
detail[3]=new MyImage(getImage(baseURL,imageString),getListbox());
} catch (Exception e) {detail[3]="Not Loaded";}
getListbox().addRow(detail);
}
getListbox().repaint();
return;
}
/**
* Gets the applet information.
* @return java.lang.String
*/
public java.lang.String getAppletInfo() {
return "ToolboxApplet.StatusApplet created using the VisualAge for
Java Version 1.0";
}
/**
* Return the Button1 property value.
* @return java.awt.Button
*/
private java.awt.Button getButton1() {
if (ivjButton1 == null) {
try {
ivjButton1 = new java.awt.Button("Search for the Order");
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Button1");
}
};
return ivjButton1;
}
/**
* Return the ItemsDb property value.
* @return ToolboxApplet.ItemsDb
*/
private ToolboxApplet.ItemsDb getItemsDb() {
if (ivjItemsDb == null) {
try {
ivjItemsDb = new ToolboxApplet.ItemsDb();
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating ItemsDb");
}
};
return ivjItemsDb;
}
/**
```

```
* Return the Label1 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel1() {
if (ivjLabel1 == null) {
try {
ivjLabel1 = new java.awt.Label("Check Order Status");
ivjLabel1.setFont(new java.awt.Font("", 1, 30));
ivjLabel1.setForeground(new java.awt.Color(0, 0, 255));
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Label1");
}
};
return ivjLabel1;
}
/**
* Return the Label2 property value.
* @return java.awt.Label
*/
private java.awt.Label getLabel2() {
if (ivjLabel2 == null) {
try {
ivjLabel2 = new java.awt.Label("Status of Order:");
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Label2");
}
};
return ivjLabel2;
}
/**
* Return the Listbox property value.
* @return itsc.taligent.widget.MyListbox
*/
private itsc.taligent.widget.MyListbox getListbox() {
if (ivjListbox == null) {
try {
ivjListbox = new itsc.taligent.widget.MyListbox();
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Listbox");
}
};
return ivjListbox;
}
/**
* Return the Panel1 property value.
* @return java.awt.Panel
*/
private java.awt.Panel getPanel1() {
java.awt.GridBagConstraints constraintsLabel1 = new
java.awt.GridBagConstraints();
java.awt.GridBagConstraints constraintsTextField1 = new
java.awt.GridBagConstraints();
java.awt.GridBagConstraints constraintsButton1 = new
java.awt.GridBagConstraints();
java.awt.GridBagConstraints constraintsLabel2 = new
java.awt.GridBagConstraints();
if (ivjPanel1 == null) {
try {
ivjPanel1 = new java.awt.Panel(new java.awt.GridBagLayout());
```

```
constraintsLabel1.gridx = 0; constraintsLabel1.gridy = 0;
constraintsLabel1.gridwidth = 1; constraintsLabel1.gridheight = 1;
constraintsLabel1.anchor = 10;
((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getLabel1(),
constraintsLabel1);
getPanel1().add(getLabel1());
constraintsTextField1.gridx = 0; constraintsTextField1.gridy = 1;
constraintsTextField1.gridwidth = 1;
constraintsTextField1.gridheight = 1;
constraintsTextField1.fill = 2;
constraintsTextField1.anchor = 10;
constraintsTextField1.weightx = 100.0;
constraintsTextField1.weighty = 100.0;
((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getTextField1(),
constraintsTextField1);
getPanel1().add(getTextField1());
constraintsButton1.gridx = 1; constraintsButton1.gridy = 1;
constraintsButton1.gridwidth = 1; constraintsButton1.gridheight =
1;
constraintsButton1.anchor = 10;
constraintsButton1.weightx = 100.0;
constraintsButton1.weighty = 100.0;
((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getButton1(),
constraintsButton1);
getPanel1().add(getButton1());
constraintsLabel2.gridx = 0; constraintsLabel2.gridy = 2;
constraintsLabel2.gridwidth = 2; constraintsLabel2.gridheight = 1;
constraintsLabel2.fill = 2;
constraintsLabel2.anchor = 10;
constraintsLabel2.weightx = 100.0;
constraintsLabel2.weighty = 100.0;
((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getLabel2(),
constraintsLabel2);
getPanel1().add(getLabel2());
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating Panel1");
}
};
return ivjPanel1;
}
/**
* Return the TextField1 property value.
* @return java.awt.TextField
*/
private java.awt.TextField getTextField1() {
if (ivjTextField1 == null) {
try {
ivjTextField1 = new java.awt.TextField(0);
} catch (java.lang.Throwable exception) {
System.err.println("Exception creating TextField1");
}
};
return ivjTextField1;
}
/**
```

```
* Method to handle events for the Applet interface.
*/
public void init() {
/* Method to handle events for the Applet interface. */
super.init();
try {
java.awt.GridBagConstraints constraintsPanel1 = new
java.awt.GridBagConstraints();
java.awt.GridBagConstraints constraintsListbox = new
java.awt.GridBagConstraints();
setLayout(new java.awt.GridBagLayout());
reshape(16, 16, 550, 298);
constraintsPanel1.gridx = 0; constraintsPanel1.gridy = 0;
constraintsPanel1.gridwidth = 1; constraintsPanel1.gridheight = 1;
constraintsPanel1.anchor = 17;
constraintsPanel1.weightx = 50.0;
constraintsPanel1.weighty = 50.0;
((java.awt.GridBagLayout)
this.getLayout()).setConstraints(getPanel1(), constraintsPanel1);
this.add(getPanel1());
constraintsListbox.gridx = 0; constraintsListbox.gridy = 1;
constraintsListbox.gridwidth = 3; constraintsListbox.gridheight =
1;
constraintsListbox.fill = 1;
constraintsListbox.anchor = 10;
constraintsListbox.weightx = 100.0;
constraintsListbox.weighty = 100.0;
((java.awt.GridBagLayout)
this.getLayout()).setConstraints(getListbox(), constraintsListbox);
this.add(getListbox());
conn24();
conn34();
initConnections();
} catch (java.lang.Throwable exception) {
System.err.println("Exception occurred in init() of
StatusApplet");
}
}
/**
* Initializes connections
*/
private void initConnections() {
/* Initialize the connections for the part */
getButton1().addActionListener(this);
}
/**
* main entrypoint - starts the part when it is run as an
application
* @param args java.lang.String[]
*/
public static void main(java.lang.String[] args) {
/* Note: Changes to this method will not be overwritten when code
is re-generated */
try {
ToolboxApplet.StatusApplet aStatusApplet = new
ToolboxApplet.StatusApplet();
java.awt.Frame frame = new
java.awt.Frame(aStatusApplet.toString());
frame.add("Center", aStatusApplet);
```

```
try {
java.lang.Class aClass =
java.lang.Class.forName("uvm.abt.edit.ExitButton");
frame.add("South", (java.awt.Component)aClass.newInstance());
} catch (java.lang.Throwable exception) {
}
frame.resize(aStatusApplet.size());
aStatusApplet.init();
aStatusApplet.start();
frame.show(true);
aStatusApplet.destroy();
} catch (java.lang.Throwable exception) {
System.err.println("Exception occurred in main() of
StatusApplet");
}
}
/**
* This method was created by a SmartGuide.
*/
public void MyInit () {
String columns[] = {"Item Name","Qty ordered","Total
Amount","Image"};
getListbox().addColumns(columns);
getListbox().getColumnInfo(0).setWidth(250);
getListbox().getColumnInfo(1).setWidth(100).setAlignment(ListboxColumn.RIGHT);
getListbox().getColumnInfo(2).setWidth(100).setAlignment(ListboxColumn.RIGHT);
getListbox().getColumnInfo(3).setWidth(100);
getListbox().setPreferedRowHeight(50);
getListbox().reshape();
return;
}
/**
* Handle the Applet start method.
*/
public void start() {
/* Handle the Applet start method. */
super.start();
}
/**
* Handle the Applet stop method.
*/
public void stop() {
/* Handle the Applet stop method. */
super.stop();
}
}
```

D.6 MyListbox.java

```
package itsc.taligent.widget;
/**
* This Class was generated by a SmartGuide.
*
*/
public class MyListbox extends MultiColumnListbox {
int thisa = 0;
int thisb = 0;
int thisc = 0;
```

```
int thisd = 0;
/**
* This method was created by a SmartGuide.
*/
public void reshape () {
super.reshape(thisa,thisb,thisc,thisd);
return;
}
/**
* This method was created by a SmartGuide.
* @param a int
* @param b int
* @param c int
* @param d int
*/
public void reshape (int a,int b,int c,int d) {
try {
thisa=a;
thisb=b;
thisc=c;
thisd=d;
super.reshape(a,b,c,d);
show();
}
catch (Throwable e)
{ System.out.println("Exception in MyListbox.reshape()");
e.printStackTrace(); System.out.println(e); }
return;
}
}
```

D.7 MyImage.java

```
package itsc.taligent.widget;
import java.awt.*;
import java.awt.image.*;
import java.net.URL;
/**
* This Class was generated by a SmartGuide.
*
*/
public class MyImage extends java.lang.Object implements
itsc.taligent.widget.Paintable, ImageObserver {
private Image image;
private MyListbox myListbox;
/**
* This method was created by a SmartGuide.
* @param imageIN java.awt.Image
*/
public MyImage (java.awt.Image imageIN) {
image=imageIN;
}
/**
* This method was created by a SmartGuide.
* @param imageIN java.awt.Image
* @param myListbox com.taligent.widget.MyListbox
*/
public MyImage (java.awt.Image imageIN, MyListbox listboxIN) {
```

```
image=imageIN;
myListbox=listboxIN;
}
/**
* This method was created by a SmartGuide.
* @return boolean
* @param image java.awt.Image
* @param infoflags int
* @param a int
* @param b int
* @param c int
* @param d int
*/
public boolean imageUpdate(Image image, int infoflags, int a, int
b, int c, int d) {
// infoflags will be set to ALLBITS if image loading is finished.
if((infoflags]ALLBITS) == ALLBITS) {myListbox.repaint();return
false;}
return true;
}
/**
* This method was created by a SmartGuide.
* @param g java.awt.Graphics
*/
public void paint(Graphics g) {
g.drawImage(image,0,0,this);
}
/**
* This method was created by a SmartGuide.
* @return java.awt.Dimension
*/
public Dimension size () {
return new Dimension(image.getWidth(null),image.getWidth(null));
}
}
```

Appendix E. 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.

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

Advanced Function Printing	AFP
AIX®	Application System/400®
AS/400®	CICS®
Client Access	Client Access/400
DB2®	DB2® /400
IBM®	OS/2®
OS/400®	PowerPC®
S/370	VisualAge®

400®

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

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

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

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

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

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

Appendix F. Related Publications

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

F.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 281.

- *AS/400 Client/Server Performance Using the Windows Clients*, SG24-4526-01

F.2 Redbooks on CD-ROMs

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

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

F.3 Other Publications

These publications are also relevant as further information sources:

- *Java in a Nutshell*, ISBN 1-56592-183-6
- *Java Developer's Reference*, ISBN 1-57521-129-7
- *Object Oriented Technology: A Manager's Guide*, ISBN 0-201-56358-4

How to Get ITSO Redbooks

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

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

How IBM Employees Can Get ITSO Redbooks

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

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

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

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

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

  ```
  TOOLCAT REDBOOKS
  ```

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

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

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

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  ```

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

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

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

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

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

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

  IBM employees may obtain LIST3820s of redbooks from this page.

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

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

Redpieces

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

How Customers Can Get ITSO Redbooks

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

- **Online Orders** —send orders to:

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

- **Telephone orders**

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

- **Mail Orders** —send orders to:

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

- **Fax** —send orders to:

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

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

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

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

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

- **Internet Listserver**

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

Redpieces

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

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

□ Invoice to customer number

□ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

AFP	advanced function printing
APA	all points addressable
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
HTML	Hypertext Markup Language
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
ITSO	International Technical Support Organization
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Development Toolkit
JIT	Just in Time Compiler
JVM	Java Virtual Machine
MI	Machine Interface
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OOP	Object Oriented Programming
PROFS	Professional Office System
RAD	Rapid Application Development
RMI	Remote Method Invocation
SCS	SNA Character Set
SLIC	System Licensed Internal Code
SSL	secure sockets layer
TIMI	Technology Independent Machine Interface
UML	Unified Methodology Language
URL	Universal Resource Locator
WWW	World Wide Web

Index

M

N

O

P

R

S

T

U

V

W

ITSO Redbook Evaluation

Accessing the AS/400 System with Java
SG24-2152-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@vnet.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_____

_____

_____

_____

_____

This soft copy for use by IBM employees only.

Printed in U.S.A.