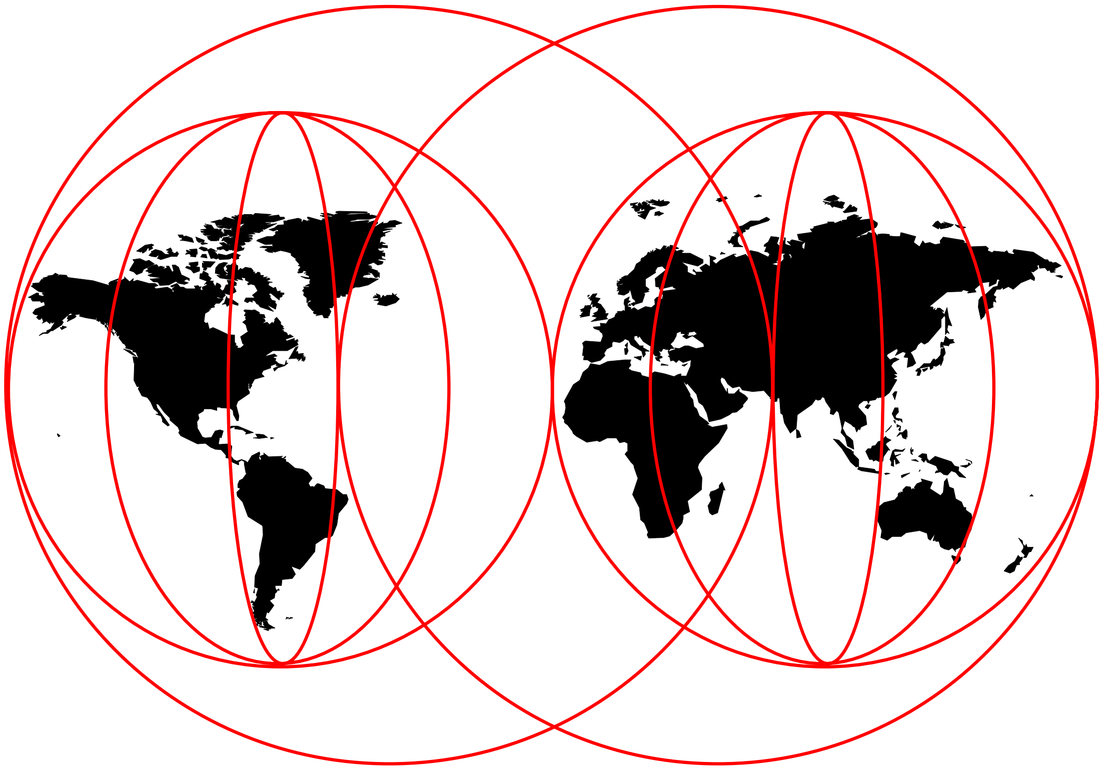
e-business Application Solutions on OS/390 Using Java: Volume I

A. Louwe Kooijmans, J. Cheong, R. Conway, T. Knopp, C. Lee
B. O'Donnell, H. Potter, J. Scanlon, B.M. Steinke, E. Van Aerschot



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5342-00

International Technical Support Organization

SG24-5342-00

e-business Application Solutions on OS/390 Using Java: Volume I

May 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, "Special Notices" on page 327.

First Edition (May 1999)

This edition applies to the following products:

- OS/390 Version 2 Release 5
- Java Development Kit Version 1.1.6 for OS/390 (beta)
- Java Development Kit Version 1.1.4 for OS/390
- DB2 for OS/390 Java Database Connectivity
- Lotus Domino Go Webserver Release 5.0
- WebSphere Application Server for OS/390 V1.1 (beta)
- IBM CICS Transaction Server for OS/390 Release 3 (LA)
- CICS Gateway for Java Version 2.0
- IBM DB2 Server for OS/390 Version 5
- IBM DB2 Connect (Personal Edition), Version 5.0
- IBM IMS Transaction Server for OS/390 Version 6
- IMS TCP/IP OTMA Connection
- MQSeries for MVS/ESA V1.2
- MQSeries Bindings for Java for OS/390 (beta)
- VisualAge for Java, Enterprise Edition for OS/390
- Windows NT Version 4.00 Workstation (with fix pack 3)

Note

This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
522 South Road
Poughkeepsie, New York 12601-5400

Contents

Figures

Tables

Preface

This redbook covers many aspects of Java in relation to the OS/390 platform. It focuses mainly on running server-side Java in the form of servlets or Java Server Pages, and provides a technical overview of the new native code compiler for Java on OS/390. It also explains SQLJ, which is a new technique for accessing DB2 databases.

A CD-ROM is included that provides easy-to-use sample applications.

The sample code associated with this redbook is also available in softcopy on the Internet from the redbooks Web server.

Point your Web browser to:

```
ftp://www.redbooks.ibm.com/redbooks/sg245342
```

Alternatively, you can go to the following URL and select **Additional Materials**:

```
http://www.redbooks.ibm.com
```

An earlier redbook, *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142, covers general topics regarding running Java on OS/390 and gives examples of solutions that use Java applets to communicate with OS/390 back-end systems. For the most complete understanding of how to use Java on OS/390, we recommend that you have both redbooks and the CD-ROM on your bookshelf.

This redbook is divided into several parts:

Part 1. An Overview of the Java Application Environment on OS/390
In this part we introduce you to the environment in which a Java enterprise application is likely to be implemented. We present the most widely used components and back-end subsystems and describe how they can be integrated with Java.

Part 2. Configuring the Java Application Environment on OS/390
In this part we describe the configuration of the most essential components on OS/390 and explain how to configure your workstation environment for developing your applications using VisualAge for Java Version 2 for OS/390 and NetObjects Fusion.

Part 3. Develop Application Solutions for OS/390 Using Java
In this part we show how you can build applications in Java using existing transactions and databases. Our focus is on the usage of servlets and JavaServer Pages to connect to DB2, IMS and CICS.

In Chapter 9, “DB2 Access” on page 117, we focus on a new way of accessing DB2 from Java: SQLJ.

In Chapter 10, “Develop Java Solutions for CICS on OS/390” on page 155, we show how you can develop and test on Windows NT and run the same application on an OS/390.

In Chapter 11, “Accessing IMS Transactions from the Web” on page 207, we show how you can access IMS from a Webserver via different methods.

Part 4. Using Servlets and JavaServer Pages on OS/390

In this part we give an overview of servlets and JavaServer Pages and how they compare.

Part 5. Using VisualAge for Java ET/390 and HPJ/390

In this part we explain how VisualAge for Java supports the OS/390 platform with some specific features especially designed for developers who want to use OS/390 as the runtime platform.

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 Poughkeepsie Center.

Alex Louwe Kooijmans is a senior IT Specialist at the International Technical Support Organization, Poughkeepsie Center. Before joining the ITSO, he worked as an application developer in MVS and AS/400 environments. During the last few years, he mainly held assignments as an IT architect in IBM Global Services projects. He holds a bachelor's degree in Computer Science and teaches and writes extensively on Java in relation to the OS/390 platform.

Jeong Sik Cheong is a sales specialist in South Korea. He has two years of experience in the Network Computing field. He holds a bachelor's degree in Agro-chemistry from Seoul National University. His areas of expertise include Web, Java, VisualAge for Java and Web-to-CICS solutions. He has written extensively on JavaServer Pages.

Rich Conway is an Advisory Technical Support Specialist at the International Technical Support Organization, Poughkeepsie Center. He has 19 years experience in MVS and OS/390 as a systems programmer. While working at the ITSO, he has been a project leader and has written on e-business and UNIX Systems Services on the OS/390 platform.

Trevor Knopp is a Technical Architect in IBM New Zealand and has seven years of experience in designing object-oriented solutions. He holds a Masters degree in Medical Physics from the University of Otago. His areas of expertise include object-oriented design and software development.

Chor Hock Lee is a Solution Architect in IBM Australia. He has 14 years of experience in the Information Technology field. He holds a Bachelor of Science degree in Computer Science from the University of New South Wales, Sydney, Australia. His areas of expertise include e-business and Application Architecture, Design and Development. He has written extensively on integrating CICS (on the S/390 in particular) and Internet technologies.

Bill O'Donnell is one of the owners of OAS Software Consulting in the USA, specializing in Java and Web deployment on OS/390. He has 15 years of experience in Technical Support for OS/390. His areas of expertise include the OS/390 Operating System and Java and Web deployment on OS/390.

Hilon Potter is a Senior Software Engineer in the US. He has 17 years of experience in MVS. His areas of expertise include OS/390 Webserver, UNIX System Services, and OS/390 new technology in general.

John Scanlon is a Senior Engineer in IBM Poughkeepsie, NY USA. He has 34 years of experience in Computer Systems. He holds a degree in Systems and Information Science from Syracuse University. His areas of expertise include Computer Systems Architecture, Network Architecture and Application Development. He has written extensively on Logical Partitioning and Multimedia Content Delivery.

Boris-Michael Steinke is an IT Specialist in Germany. He has eight years of experience in object-oriented technologies and client/server computing. He has studied business management at Ludwig Maxemilian University in Munich, Germany. His areas of expertise include distributed object-orientated application development in OO languages like C++ and Java. Currently, he is working as a Senior Consultant for a German IT company, mainly on projects with financial service companies. He is responsible for the development of OO approach models and design patterns for distributed C/S applications based on Java using Internet technologies. He also teaches OO methodologies and the Java and C++ programming language for IBM education and training in Germany and Switzerland.

Egide Van Aerschot has 30 years of field experience with many Finance and Government customers. He was responsible for many projects involving S/390 mainframes related to IMS, DB2 and MQSeries. He designed and installed several client server projects with Distributed Computer Environment (DCE) and Advanced Program-to-Program Communication (APPC). In recent years he participated in several residencies at the International Technical Support Organization and was involved in the design of solutions to access existing data and transactions from the Internet.

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

Terry Barthel
International Technical Support Organization, Poughkeepsie Center

Ian Burrows
IBM, Toronto Lab

Ella Buslovich
International Technical Support Organization, Poughkeepsie Center

Clarence Clark
IBM, Poughkeepsie

Christine Casey
IBM, Endicott

Evgeny Deborin
International Technical Support Organization, San Jose Center

Carol Dixon
International Technical Support Organization, Poughkeepsie Center

Mike Fulton
IBM, Toronto Lab

Bob Haimowitz
International Technical Support Organization

Sally Howard
Java Technology Centre, Hursley Park, UK

Mike Oliver
IBM, Poughkeepsie

Brian Peacock
Java Technology Centre, Hursley Park, UK

Roland Trauner
International Technical Support Organization, Poughkeepsie

Tommy Toomire
IBM, Santa Theresa Lab

Lulu Wong
IBM, Toronto Lab

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 337 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Part 1. Overview of Java Application Environment on OS/390

In general, new technologies may bring many improvements to the Information Systems environment, but we also expect those technologies to deliver the same quality as we were used to in the traditional environment.

It may not be a good idea to sacrifice performance, security or reliability in an effort to just get a fancy GUI. Especially in the OS/390 environment, in which factors as reliability, availability, security and performance are characteristic, we need to carefully design solutions based on new technology.

In many cases, the ideal application is most probably a mix of both new technology and proven traditional technology. So the question can be asked, what is “new technology,” actually?

The definition of “new technology” may change every day, but within the context of this book, we define new technology as `components that fit into the following categories of products`:

- Java language and runtime environment
- Enterprise JavaBeans
- Component Broker
- Distributed Object-Oriented applications
- e-business connectors between Java and subsystems
- Webservers

In this part of the redbook we update you with the latest information about components in the area of e-business application enablement on OS/390.

In Chapter 1, “The e-business Application Framework” on page 3, we explain how IBM's e-business Application Framework is supported on OS/390 by means of the WebSphere family of products.

In Chapter 2, “Java Client/Server” on page 11, we show you how the client/server model can be used to build Java Enterprise applications.

Finally, in Chapter 3, “OS/390 Components” on page 19, we dedicate a section to each component on OS/390 supporting the e-business Application Framework.

Chapter 1. The e-business Application Framework

The e-business Application Framework is a platform-transparent "model" supporting all potential functions of a Web-based e-business application.

Figure 1 gives a logical view of the framework in case an OS/390 server would be used. But again, the OS/390 server in the picture can be easily replaced by any other server as long as the server is compliant to the framework.

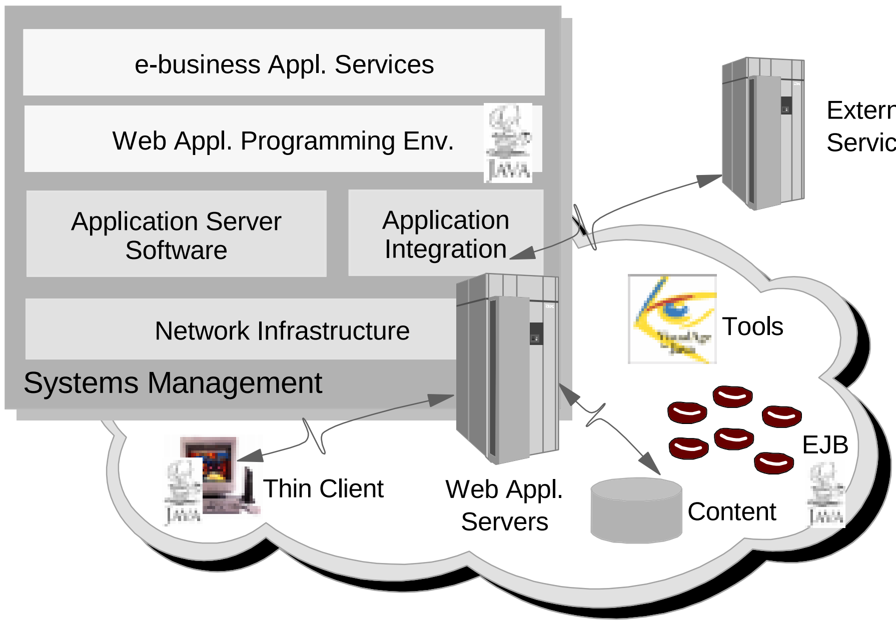


Figure 1. Logical View of the e-business Application Framework

The e-business Application Framework is composed of the following key elements:

Clients

Clients should be as thin as possible without any local application logic.

Network Infrastructure

This supports network, directory, security, file access and print services.

Application Server Software

This supports the applications in terms of database access, transaction services, message queuing and so on.[1]

Connectors

These provide access to traditional existing data and applications.[2]

1 In the Network Computing Framework, this building block was also knows as "Foundation Services."

2 In the Network Computing Framework, this building block was also known as "Connectors."

Web Application Programming Environment

This is the actual runtime environment of an application.

e-business Application Services

These are the building blocks or frameworks at an application level that can be used to easily assemble an application for a particular industry or solution segment.

Systems Management

This supports the full life cycle of an application from installation and configuration, to the monitoring of its operational characteristics, to the controlled update of changes.

Development Tools

These are used to create applications.

In 1.1, "Application Server Software" through 1.3, "Development Tools" on page 7, we discuss the components of the e-business Application Framework that are important to understand when reading this redbook.

1.1 Application Server Software

The Application Server Software, also known as "Foundation Services," provides the home for the business logic of the application and includes the HTTP server, database and transaction services, mail and community services, groupware services and messaging services. By means of Java classes and Beans, integrated solutions can be built that combine the usage of various services into one e-business application.

In the context of this book, important services are the following:

1.1.1 HTTP Server

The HTTP server is the entry point for client requests and is responsible for the execution of the first middle-tier logic of an e-business application. This middle-tier logic is preferably implemented as a servlet or JavaServer Page.

If needed, the HTTP server also hosts the connectors to existing applications and data. If Java servlets or JavaServer Pages are used, connectors are available to almost all transaction, database and file systems on OS/390. Refer to Figure 10 on page 16 for an overview of the available connectors.

Note: In the e-business application framework, the Application Server is also responsible for IIOP communication between application components. At the time of writing, no such support is generally available on OS/390 yet.

1.1.2 Database Services

The Database Services leverage existing business logic to build new Web-based applications. Stored procedures and database application programs are supported using the Java and JavaBeans programming model. In combination with servlets and JavaServer Pages providing the business logic, the data access logic is integrated using the JDBC and SQLJ specification. Refer to 3.3.1, "DB2" on page 22 for an overview of SQLJ and JDBC and to Chapter 9, "DB2 Access" on page 117 for details.

1.1.3 Transaction Services

Transactional *processing* has become the most important concept in enterprise applications, and transaction *services* have become more important than ever in the world of e-business applications.

In this world, a client request must be seen as a *unit of work*. Client requests must either be 100 percent executed, or else rolled back. A two-phase commit protocol is critical.

1.1.4 Messaging Services

The two most important benefits of messaging services are:

- Asynchronous processing
- Decoupling of systems

We discuss these benefits in detail in the following sections.

1.1.4.1 Asynchronous Processing

Asynchronous processing can be best compared to a situation where you would have a telephone conversation. Imagine the world without answering devices, as was the case some twenty years ago. You could only deliver your message by phone to another person by having that person actually on the phone. Otherwise, you would have to call that person again and again until he or she picks up the phone.

By using an answering device on the other side, you could deliver the message anyhow, even when the other person did not pick up the phone. You would not have to call again and again.

This situation also applies to systems that communicate with each other. When the sending system cannot reach the receiving system, you do not want to have the sending system blocked because the receiving system is not available.
Using message services, the sending system will keep the message in a queue until the receiving system is available again. It will then release the message.

Another feature of messaging services is that each message gets a unique identifier, so that the receiving system can confirm back that the message has been received.

1.1.4.2 Decoupling of Systems

Messaging services also *decouple* systems. Figure 2 on page 6 gives an example of a messaging service where a client would communicate with a host system. The client can also be on the same computer as the host system.

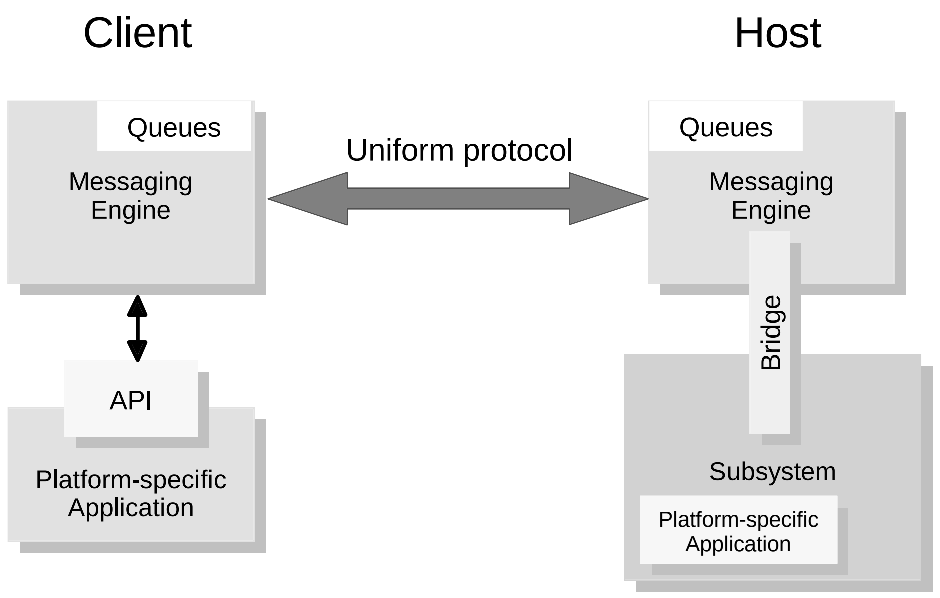


Figure 2. Messaging Services

Typically, instead of sending a system-specific message directly to the host application, the client will first convert its message to the uniform protocol required by the messaging service. (APIs are available to do this, including when Java is used). Then, the generic message will be sent to the host system, where it will be converted back to the specific format required by the back-end system.

Imagine now that, for some reason, the back-end application will be moved to another location, eventually even to another subsystem or platform. In that case, you only have to use another bridge on the host system; the client remains unchanged.

1.2 Application Integration

The function of application integration, also known as “connectors,” is to connect the logical middle tier to existing applications and data. The main benefit of a connector is that it hides the specific protocol as required by the existing transaction, database or file system.

For instance, programming a Web-client communicating with IMS via the OTMA protocol can be more difficult than talking HTTP/Java to the logical middle tier and then using a connector to take care of the communication between the middle tier and IMS via the OTMA protocol.

Note that (without opening an in-depth discussion about performance), the user-friendliness route is not necessarily the same route as the performance route. In some cases, connectors may impose additional overhead, especially in transactional environments with high throughput. In those cases, even the slightest overhead may have to be cut out, leading to a more tailor-made solution without connectors.

Figure 3 on page 7 shows the basic principle of a connector.

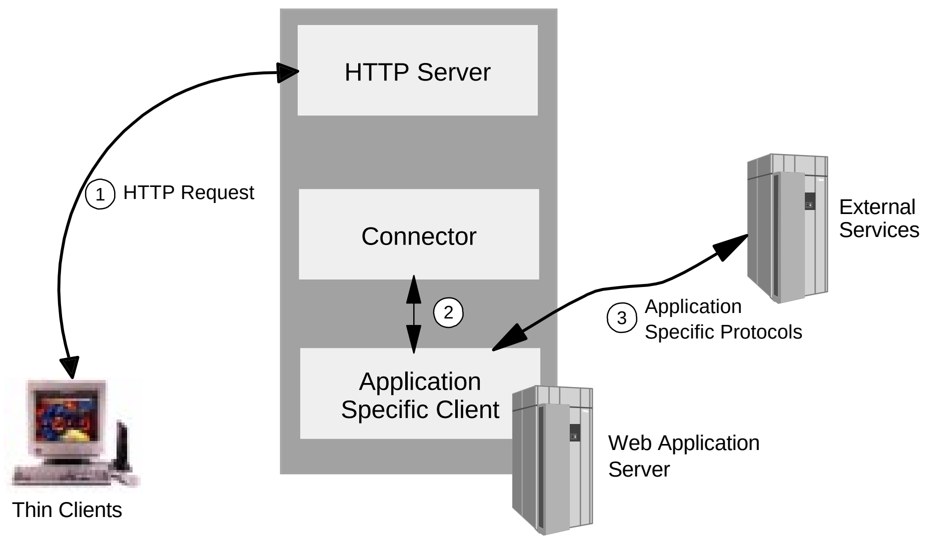


Figure 3. Connectors in the e-business Application Framework

1.3 Development Tools

Currently, IBM's WebSphere Studio can be used to create parts of e-business applications. The WebSphere Studio V1.0 comes with the following products:

- Workbench
- NetObjects Beanbuilder V1.0
- NetObjects Fusion V3.0.1
- VisualAge for Java, Professional Edition, Version 2.0
- WebSphere Application Server, Version 1.1
- Apache HTTP Server, Version 1.3.1
- Another base HTTP server

Attention

The exact content of WebSphere Studio may change. You can assume, however, that NetObjects and VisualAge for Java are part of the core of the WebSphere development environment.

You can find more details on NetObjects in Chapter 7, “NetObjects Fusion (NOF) Version 3” on page 93.

Refer to *Programming with VisualAge for Java Version 2*, SG24-5264, for more details about VisualAge for Java Version 2.0.

1.4 Java - the Strategic Language for e-business Applications

The Java language is the strategic language to use for e-business applications. It is in the nature of Java to be used in a multiplatform distributed application environment, and the enormous availability of APIs and JavaBeans makes it very easy to access any type of service on any type of computer. We can really say that you are able to create a complete solution using only one programming language and thus using only one type of skill.

On the server side, servlets, Enterprise JavaBeans and JavaServer Pages will form the Web Application Programming environment. More than ever, Java is seen as the strategic programming language, of course, also on OS/390.

Table 1 illustrates the Java APIs to be used for the Network Infrastructure.

Table 1. Java APIs to Support the Network Infrastructure

Service	Protocol Standard	API
Directory	LDAP	JNDI
Security	CDSA, SSL, IPsec, x.509V3	JSSL, JCE
Network	TCP/IP	JDK java.net
File	AFS/DFS	JDK java.io
Print	IPP/DFS	JDK java.2d, JNPAPI

Table 2 illustrates the Java APIs to be used for the Foundation Services.

Table 2. Java APIs to Support the Foundation Services

Service	Protocol Standard	API
Mail and Community	SMTP, POP3, IMAP4, IRC, NNTP, FTP, ICalendar	Java Notes API
Groupware	n/a	Java Notes API
Data	ODBC, DRDA	JDBC, SQLJ, EJB
Transactions	CORBA IIOP/OTS	EJB, JTS
Message Queuing	BMQS	JMS

Table 3 illustrates the Java APIs to be used for the Web Application Programming Environment.

Table 3. Java APIs to Support the Web Application Programming Environment

Service	Protocol Standard	API
Web Server	HTTP, HTML, XML	Servlets, Server-side includes
Web Browser	HTTP, HTML, XML	Applets, DOM Level 1
Component Model	CORBA IIOP	JavaBeans
Business Component Model	CORBA IIOP	EJB, RMI
Scripting	ECMAScript	JSP

1.5 IBM WebSphere Application Server

The e-business Application Framework is mainly filled in by a family of products called Websphere. IBM is committed to supporting Websphere's functionality on all IBM platforms, including AIX, OS/390, AS/400, OS/2 and Windows.

At the time of writing, Lotus Domino Go Webserver Release 5.0 is available for OS/390 with WebSphere Application Server 1.0 to support servlets and JavaServer Pages. The planned GA date for WebSphere Application Server for OS/390 V1.1 tied into IBM HTTP Server Version 5.1, which in turn, is tied into OS/390 V2R7, is March 1999. Figure 4 gives a high-level view of the Websphere functionality.

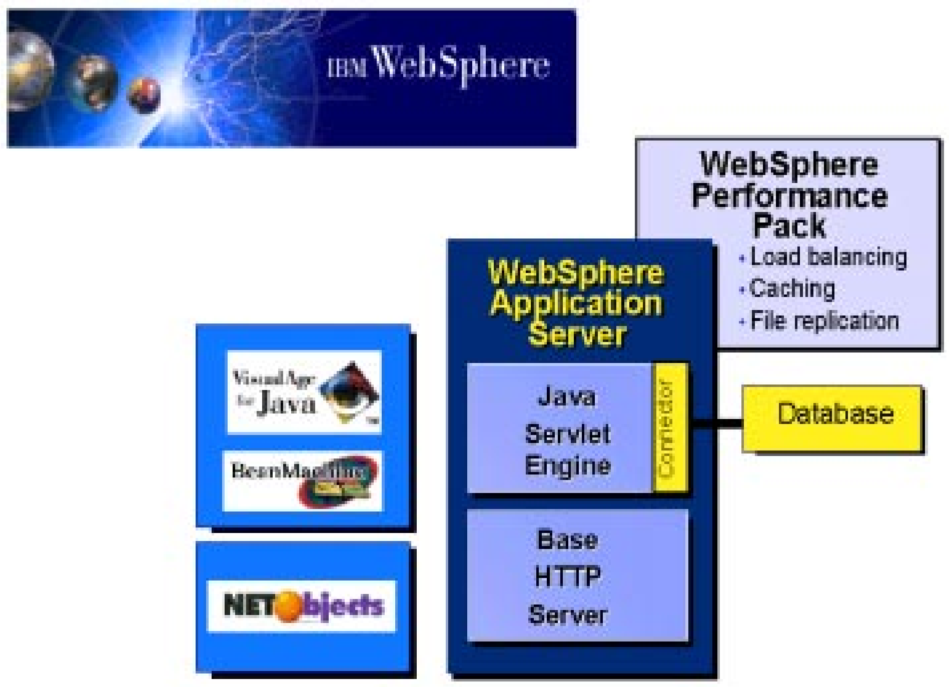


Figure 4. High-Level Overview of WebSphere Application Server

Note: The functions of the WebSphere Performance Pack are integrated as a standard feature in HTTP Server Version 5.1 for OS/390.

1.6 Development of e-business Applications for OS/390

After reading the previous sections, you might ask how OS/390 is supported.

Currently, WebSphere Application Server Version 1.1 is available on OS/390, supporting servlets, JavaServer Pages and connectors to the following:

- IMS

  For more details on connecting to IMS from the Webserver, refer to Chapter 11, “Accessing IMS Transactions from the Web” on page 207.

- CICS

  For more details on connecting to IMS from the Webserver, refer to Chapter 10, “Develop Java Solutions for CICS on OS/390” on page 155.

- Relational databases using JDBC

  For more details on connecting to IMS from the Webserver, refer to Chapter 9, “DB2 Access” on page 117.

- DB2 using SQLJ

  For more details on connecting to IMS from the Webserver, refer to Chapter 9, “DB2 Access” on page 117.

- MQSeries

  The MQSeries bindings required to access the MQSeries Manager on OS/390 are currently in beta. For more details on connecting to IMS from the Webserver, refer to 11.3, “Access to IMS Using a Servlet/MQI” on page 235.

- MVS datasets using JRIO

  Those include access to sequential, partitioned and VSAM datasets. Both entry sequence datasets and key sequence datasets are supported.

VisualAge for Java Enterprise Edition Version 2.0 can be used to build Java and JavaBeans to run on OS/390. NetObjects Fusion and NetObjects Scriptbuilder can be used to create Web sites.

Chapter 2. Java Client/Server

As explained extensively in *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142, there are many ways to create a Java client/server application with OS/390 as the server platform. The need to have a clear separation between presentation logic, business logic, and data access logic has never been as important as in the highly security-sensitive Internet world we live in today.

In 2.1, "Why Client/Server" we discuss the many advantages of client/server computing in an Internet world. In 2.2, "Java Client/Server Scenarios with OS/390 Today" on page 12, we will give you an update on client/server models using an OS/390 server.

2.1 Why Client/Server

Having a clear separation between the presentation logic, business logic, and data access logic gives you the following advantages:

- Different technology

  In Web applications, the presentation logic may be implemented using other technology than, for instance, the data access logic. It may be easier to implement the front-end using HTML, while the data access logic needs to be implemented using Java code with JDBC calls.

- Skills

  Typically, the presentation layer of a Web application contains many graphics. Building the presentation layer requires the use of specific tools capable of generating GUIs. However, the creation of data access logic may require in-depth knowledge of accessing the database in the most efficient way.

  This is just an example of how the different layers in a client/server application require different skills. If you have a true separation between the layers, you may only need one skill profile per component.

- Maintenance

  Separating the presentation logic, business logic and data access logic just makes it easier to maintain the application. We would not want to have huge programs mixing GUI code, business logic, and data access routines.

- Security

  Especially in the Internet world, where programs are executed in the browser of the client, you have to be careful with the code you put in the downloaded programs. For example, a smart user may try to intercept the downloaded code and find out the internals of a sensitive account balance application.

- Performance

  Special consideration must be given to the performance aspects of the Web application. Executing the data access logic as part of the same program that does the GUI may give different performance results than implementing a separate data access module on the same server as where the database is implemented.

- Reliability

Having a server-side program in charge of data access and eventually database connectivity gives better opportunities for maintaining the state of connections to the database than having a direct connection from Internet client to the database across the network.

- Scalability

  The thinner the client, the more scalable the application will be. Actually, the only thing we would like to distribute to clients is the front-end of the application. Ideally, this front-end would have only one server partner to communicate with. The server partner will take care of accessing the right database or transaction system on either the same server or another server.

2.2 Java Client/Server Scenarios with OS/390 Today

In this section we provide an update of the various infrastructures and models that can be used to create a Java client/server application.

2.2.1 The Client

The client can be either “thin” or “fat.” A thin client will not require any specific systems software to run an application, except for the base operating system and a browser. Also, a thin client will not have components of the application stored locally on the harddisk.

For intranet types of applications we may consider a fat client in very specific situations, but it is obvious that a fat client will have a higher maintenance and systems management cost. Also, an application requiring a fat client is less scalable and has more security exposures than an application using a thin client.

For Internet types of applications, there is no choice: the client has to be thin.

2.2.2 OS/390

OS/390 is an ideal environment to implement business logic, data access logic, and “connection” logic to get to databases, files and transaction systems. Scalability can be achieved easily by using the WebSphere Application Server, or a tailor-made Java application server, or by just using the socket interfaces of DB2, CICS or IMS.

In the following sections we present the various possibilities of designing a Web application using Java-and-WebSphere technology.

2.2.3 Communication Protocols

When designing a distributed object-oriented application, you may choose between the following protocols to use between the client and the server:

- HyperText Transfer Protocol (HTTP)
- Remote Method Invocation (RMI)
- Internet Inter-ORB Protocol (IIOP)
- Direct sockets

Figure 5 on page 13 shows the protocols that are available for OS/390 at the time of writing of this book.

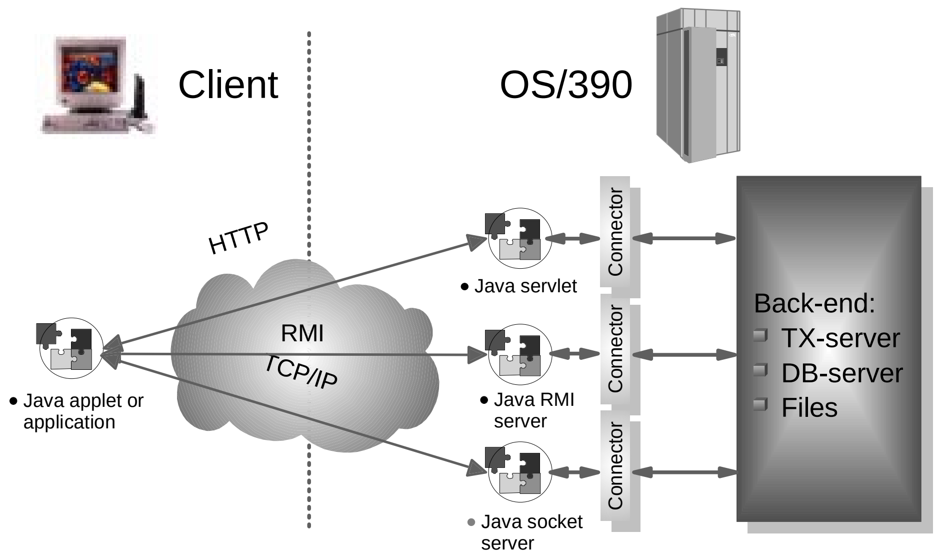


Figure 5. Communication Protocols using OS/390

2.2.3.1 HyperText Transfer Protocol (HTTP)

HTTP is the most widely used and known protocol for Web applications. As shown in Figure 6, documents are passed back and forth between a Web browser and Web server using HTTP in most Internet applications.

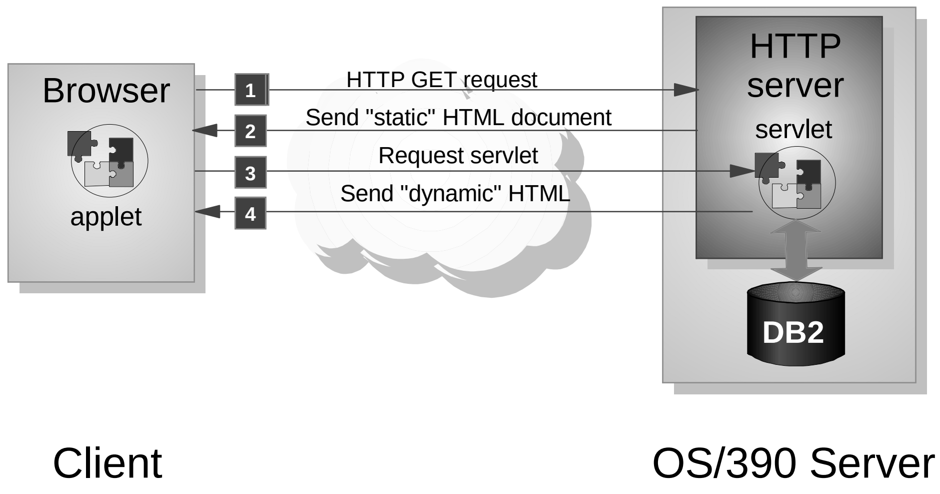


Figure 6. Communication between Client and OS/390 Server Using HTTP

To do a basic Internet application based on static HTML documents and eventually Java applets in the front-end, you will only need a HTTP Webserver installed on the server. However, HTTP is not seen as the strategic protocol for transactional Web applications with a high throughput and requiring a state.

The `java.net` package contains classes for use with HTTP.

2.2.3.2 Remote Method Invocation (RMI)

RMI is SUN's standard protocol for communication between Java objects residing on different computers. It is quite easy to implement, as some IDEs support automatic generation of the communication layer. RMI is standard supported in JVMs of 1.1 and higher.

An RMI server application can run stand-alone on OS/390, it does not require a Web server, and it can perform anything a regular Java application can do on OS/390.

Ideally, the Java application server should be a multithreaded application, and connectors should be used to communicate with the back-end systems.

Figure 7 shows an example of an RMI client/server application accessing a DB2 database on OS/390. The DB2 access is done with JDBC calls.

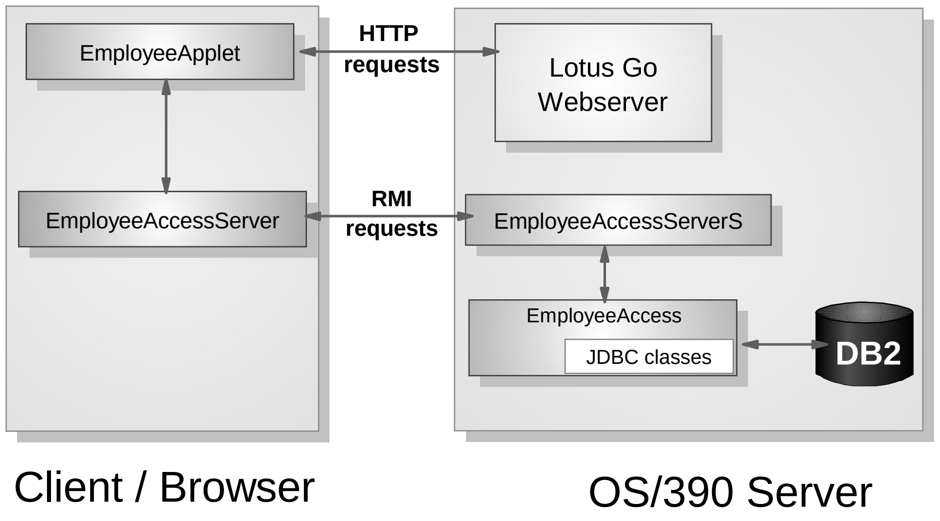
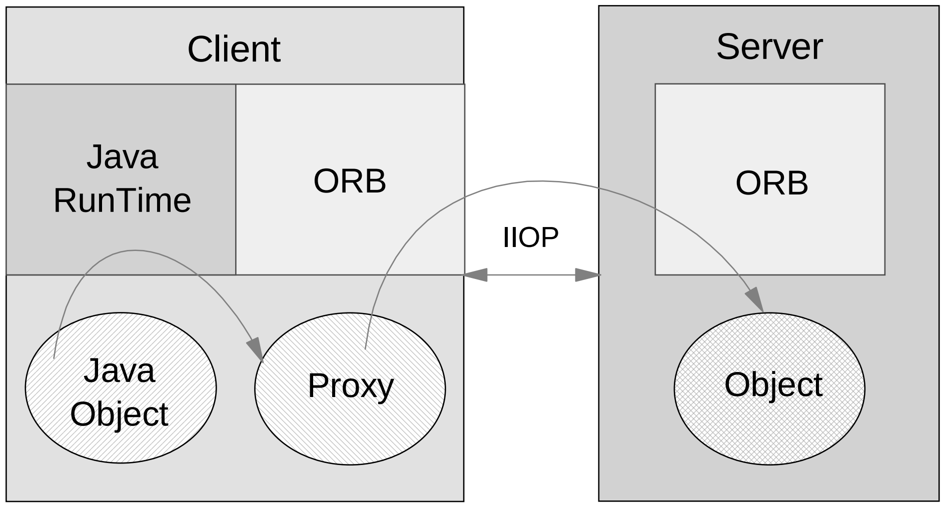


Figure 7. An Example of an RMI Application Accessing DB2 on OS/390

2.2.3.3 Internet Inter-ORB Protocol (IIOP)

IIOP is the standard protocol of CORBA. A key component in the CORBA infrastructure is the “ORB.” Both the client and server will need an ORB to communicate. On the client side, the ORB may be implemented as a plug-in in the Web browser. On OS/390, IBM is committed to tie Component Broker's ORB into the WebSphere Application Server.

Figure 8 on page 15 shows the communication using IIOP/CORBA in a simplified way.



Proxy invokes method on remote object using ORB infrastructure

Figure 8. Communication between Client and OS/390 Server Using IIOP

IIOP is a more advanced protocol than HTTP or RMI, as it supports “services,” like transaction services and messaging services.

At the time of writing, there is no generally available CORBA/IIOP support on OS/390 yet. IBM's Component Broker/390 will support IIOP in the near future on OS/390, and IBM CICS Transaction Server for OS/390 Release 3 (LA) will support incoming IIOP messages from CORBA 2.0 compliant clients.

Refer to *Integrating Java with Existing Data and Applications*, SG24-5142, for an overview of CORBA and IIOP Component Broker on OS/390.

2.2.3.4 Direct Sockets

Figure 9 illustrates the use of a socket server on OS/390.

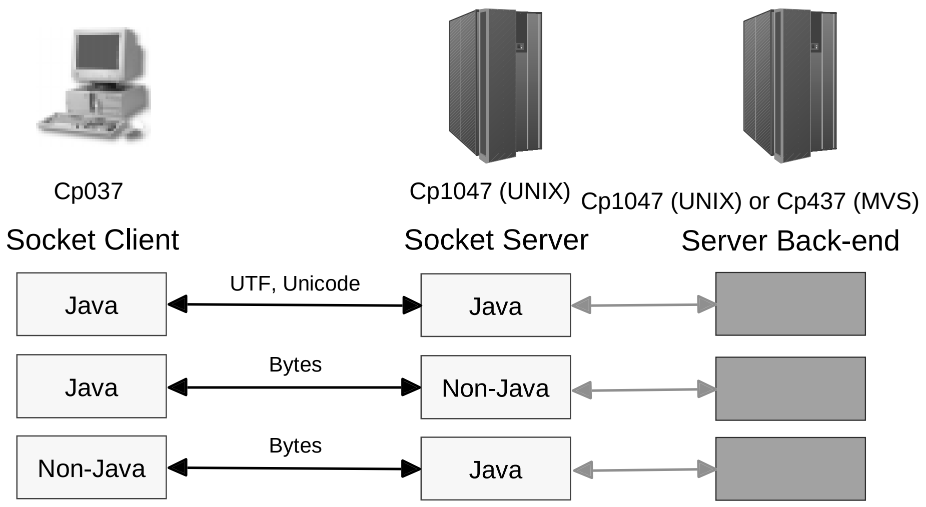


Figure 9. Communication between Client and OS/390 Server Using Sockets

You can do almost anything using sockets. Basically, sockets are under the covers of other protocols too. However, note that when using sockets, you will have to take care of recovery, security and, eventually, ASCII/EBCDIC conversion yourself.

Most subsystems on OS/390 have a socket interface that can be accessed directly from a Java applet or application running on the client, but you can also create a Java application server that handles the client requests and communicates with the back-end systems on OS/390.

As previously mentioned, ideally the Java application server should be a multithreaded application and connectors should be used to communicate with the back-end systems.

2.3 Accessing Back-End Systems on OS/390

It is difficult to consider e-business applications without also thinking of reusing existing data and programs on OS/390. It is estimated that about 70% of the mission-critical enterprise data is stored on mainframes running OS/390. Today, a variety of Java APIs and connectors are available to access the subsystems of OS/390. Most of them can be used to not only directly access the subsystem from the client, but also access it from a servlet or JavaServer Page (JSP) running in the WebSphere Application Server or Lotus Domino Go Webserver Release 5.0. They all support access from a Java application or a Java application server running on OS/390.

Figure 10 gives the status of the various APIs, or e-business connectors, as of the date of publication of this book.

	Function	**Available/planned**	**Direct access from:**
DB2	JDBC SQL/J	Available now Downloadable	OS/390 applications/ servlets
CICS	CICS Gateway for Java JCICS	Available now GA planned for 1Q/99	OS/390 appl./ servlets and TCP/IP clients CICS/390 Java Transactions
IMS	IMS TCP/IP OTMA Connection	Available now	OS/390 appl./ servlets and TCP/IP clients
MQSeries	MQSeries Client for Java MQSeries bindings for OS/390	Available now Available now in Beta	OS/390 appl./ servlets and TCP/IP clients OS/390 appl./ servlets
VSAM files	Record I/O	Technology Preview	OS/390 appl./ servlets
C/C++ modules	JNI	Available	OS/390 appl./ servlets
ASM, COBOL, PL/1	JNI	Available	Via JNI (C stub)
GUI	AWT via X11 Remote AWT	Available Downloadable	N/A N/A

Figure 10. Available Java APIs/Connectors for Accessing Back-End Systems on OS/390

In this redbook we document most of the connection scenarios that have not yet been documented in *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142.

Chapter 3. OS/390 Components

In this chapter we give an overview of the key products available on OS/390 to support Web-enabled e-business applications.

In the context of this book, we assume a physical two-tier configuration, where OS/390 runs the business logic and data access logic, and the Web client runs the presentation logic. In order to run business logic on OS/390 using the e-business Application Framework, the minimum to be installed is a Web Application Server like WebAS, and Java support. Back-end systems like IMS, CICS, or MQSeries are optional components, but most OS/390 installations do run at least a transaction monitor and a relational database. For this reason we provide a brief section about each of them.

3.1 Web Application Server Software on OS/390

IBM introduced the first release of the Webserver for OS/390 in December 1995. This release was called the IBM Internet Connection Server for MVS/ESA. Since December 1995, the Webserver for OS/390 has changed its name, as new features became available. These names included IBM Internet Connection Server for MVS/ESA, IBM Internet Connection Secure Server for OS/390, and Lotus Domino Go Webserver. At the time of writing, IBM's Webserver on OS/390 is called Lotus Domino Go Webserver Release 5.0.

In June 1998, IBM introduced Lotus Domino Go Webserver Release 5.0, including a component called ServletExpress (SE) 1.0. In November 1998, ServletExpress was replaced with the WebSphere Application Server for OS/390 V1.0 (WebAS). ServletExpress or WebSphere Application Server is the component that supports servlets for Lotus Domino Go Webserver Release 5.0.

In March 1999, WebSphere Application Server for OS/390 V1.1 will be a standard plug-in of IBM HTTP Server for OS/390 Version 5.1. The IBM HTTP Server will be tied into OS/390 Version 2 Release 7, delivering the same functionality as WebSphere Application Server Version 1.1 on other platforms.

Lotus Domino Go Webserver Release 5.0 includes the following features:

- CGI Support
- Remote Server Configuration
- EBCDIC/ASCII files access
- Thread-level security
- Surrogate UserID
- MVS UserID
- SSL V2 and V3
- Proxy functions
- Internet Connection API
- Logging and reporting
- Performance Enhancements
- SNMP MIB
- Automatic Browser Detection
- PICS Server Support
- Cookie support
- Page counter

- Certificate authentication
- Certification authority
- Internal Java Servlet Support
- FASTCGI Support
- RAS enhancement
- Cryptographic key size selection
- Export security 128-bit encryption
- Web Traffic Express
- LDAP
- Y2K-ready

For more information on the OS/390 Webserver, refer to the following publications:

- *Domino Go Webserver 5.0 for OS/390: Webmaster's Guide*, SC31-8691
- *Domino Go Webserver 5.0 for OS/390: Planning for Installation*, SC31-8690
- *Domino Go Webserver 5.0 for OS/390: Messages*, SC31-8692
- *OS/390 R6 Domino Go Webserver 5.0 Web Programming Guide*, SC34-4743

3.2 Java Environment

Today, programs written in the Java language are supported on OS/390 in different ways. In the following sections, we give a brief overview of what is possible today.

Attention

In this redbook we use the term HPJ for the native code compiler for Java on OS/390 because HPJ was IBM's internal code name.

However, the official product name is VisualAge for Java, Enterprise Edition for OS/390.

3.2.1 Java Bytecode

Java can be run as bytecode on OS/390 through a Java Virtual Machine (JVM). The Java Virtual Machine has been generally available on OS/390 since September 1997 and is implemented as a so-called “external” JVM. Figure 11 on page 21 shows where the JVM is located on OS/390.

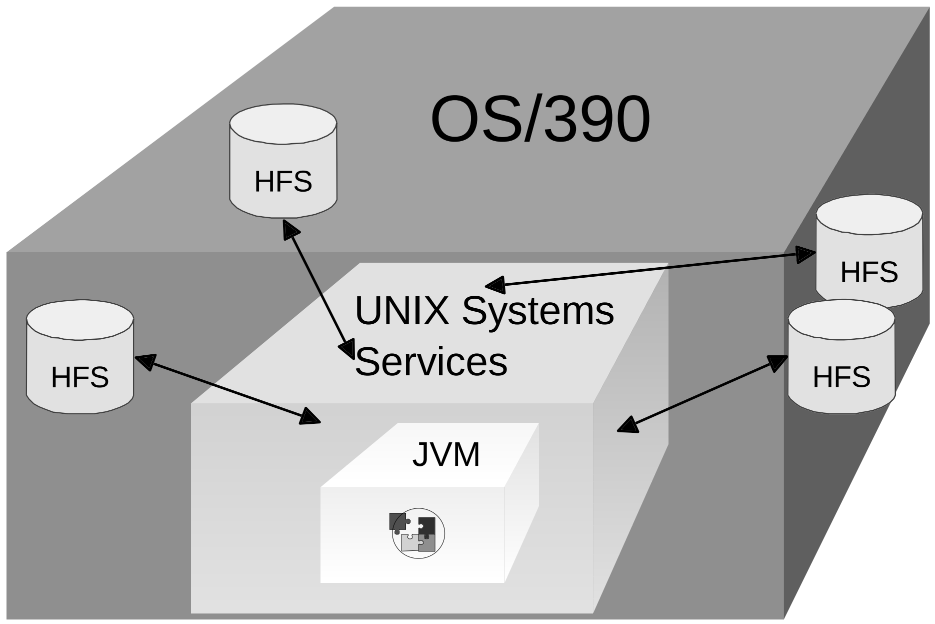


Figure 11. Implementation of the JVM on OS/390

Lotus Domino Go Webserver Release 5.0 and WebSphere Application Server for OS/390 V1.1 use a so-called “internal” JVM to run servlets. The JVM product as such, being the Java classes and C DLLs, is the same as used in the external JVM. When using ServletExpress or WebAS, you actually point to the JVM classes and DLLs in the properties files.

In the near future, IMS and CICS will implement similar “internal” JVMs to run Java transactions, while DB2 will implement a similar “internal” JVM to run Java stored procedures.

Servlets are currently only supported in bytecode form. Refer to *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142, for more details about the JVM on OS/390.

3.2.2 Java Object Code

In October 1998, IBM released the Java Native Code Compiler for OS/390, also known as “High Performance Java for OS/390” (HPJ/390) and “High Performance Compiler for Java for OS/390” (HPCJ/390). This compiler turns Java bytecode into object code, with the primary objective of simply speeding up the execution of the application.

The Java object code can be executed as a Java application in the OS/390 UNIX System Services or as a CICS transaction in CICS Transaction Server 1.3. It needs a specific environment in order to compile and run the code and in 5.2, “VisualAge for Java, Enterprise Edition for OS/390” on page 58 we discuss the configuration of the environment for the native code compiler. In Chapter 18, “Using HPJ/390 - Scenarios” on page 273, we give a brief overview of the usage of HPJ/390.

3.2.3 Java e-business Connectors on OS/390

Java components can be integrated with other subsystems on OS/390 in many ways. *Integrating Java with Existing Data and Applications*, SG24-5142 discusses several e-business connectors and how they can be applied in an Internet environment. Figure 12 gives a brief overview of the integration between Java applications and servlets on one side, and existing database and transaction servers on the other side.

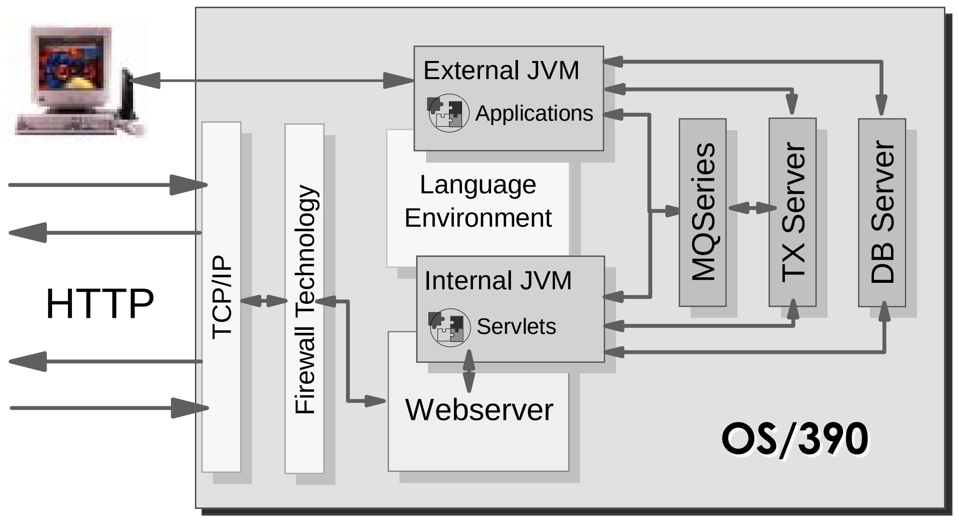


Figure 12. Integration between Java and Subsystems on OS/390

3.3 Back-End Systems on OS/390

In 3.3.1, “DB2” through 3.3.5, “MVS Datasets” on page 25, we briefly highlight how Java is supported in the major subsystems on OS/390.

3.3.1 DB2

In order for Java applications to use DB2, the programmer can use either JDBC or SQLJ to access DB2 data from Java. Both methods can be used from a servlet or a JavaServer Page or a Java application.

The main difference between JDBC and SQLJ is that JDBC uses “dynamic” SQL calls and SQLJ uses “static” SQL calls. From a programming point of view, they are very similar.

SQLJ has the advantage that it runs faster, but you need to do a few steps more in order to prepare the program. You can find all the details regarding this process in 9.2, “SQLJ Implementation for DB2 on OS/390” on page 128, including examples.

A minor inconvenience that we found during the writing of this book was that, as the SQLJ statements have a specific syntax (a statement always starts with a “#”), it is hard to fully develop the code in an Integrated Development Environment (IDE).

In contrast, JDBC runs slower, but it is easier to prepare the program.

3.3.1.1 JDBC

The DB2 for OS/390 JDBC driver provides Java applications a program interface (API) to access DB2 on OS/390 using a local DB2 attachment (RRS). The DB2 for OS390 JDBC driver is implemented as a JDBC-ODBC bridge (known as a Type 1 JDBC driver). A Type 1 driver maps all JDBC method invocations to ODBC calls. Therefore, the DB2 for OS390 V5.1 JDBC driver requires that the DB2 for OS/390 Version 5.1 CLI driver support be installed.

To understand JDBC, it is helpful to know about its purpose and background. Sun Microsystem's JavaSoft developed the specifications for a set of APIs that allow Java applications to access relational data. The purpose of the APIs is to provide a generic interface for writing platform-independent applications that can access any SQL database. The APIs are defined within classes that support basic SQL functionality for connecting to a database, executing SQL statements, and processing results. Together, these interfaces and classes represent the JDBC capabilities by which a Java application can access relational data.

JDBC offers a number of advantages for accessing DB2 data:

- Using the Java Language, you can write an application on any platform and execute it on any platform with a Java Virtual Machine (JVM) installed.
- JDBC combines the benefit of running your applications in an OS/390 environment with the portability and ease of writing Java applications.
- The ability to develop an application once and execute it anywhere offers the potential benefits of reduced development, maintenance, and systems management cost, and flexibility in supporting diverse hardware and software configurations.
- The JDBC interface offers the ability to change between drivers and access a variety of databases without recoding your Java programs.
- JDBC applications do not require a precompile or bind.

3.3.1.2 SQLJ

SQLJ is another interface from Java applications to DB2. SQLJ provides support for embedded static SQL statements in Java applications. It is the static SQL equivalent of JDBC. Some of the major differences between SQLJ and JDBC are:

- In most cases, SQLJ source programs are smaller than equivalent JDBC programs because the SQLJ program preparation process provides programming interfaces that you must write explicitly in your JDBC programs.
- SQLJ does data type checking during the program preparation process and enforces strong typing between table columns and Java host expressions. JDBC passes values to and from SQL tables without compile time data type checking.
- In SQLJ programs, you can embed Java host expressions in SQL statements. JDBC requires a separate call statement for each bind variable, and specifies the binding by position numbers.

IBM announced early availability of SQLJ support in DB2 for OS/390 Version 5 via APAR PQ19814 on October 30, 1998.

3.3.1.3 Resource Management

When using JDBC or SQLJ on OS/390, you will need to select the type of attachment facility to connect to DB2. JDBC or SQLJ can use either the DB2 for OS/390 Call Attachment Facility (CAF) or the DB2 for OS/390 Recoverable Resource Manager Services Attachment Facility (RRSAF).

An application program can use the Recoverable Resource Manager Services Attachment Facility (RRSAF) to connect to and use DB2 to process SQL statements, commands, or Instrumentation Facility Interface (IFI) calls. RRSAF uses OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS).

CAF is a part of the DB2 code that allows some of the same function as RRSAF. CAF only requires DB2 for OS/390 Version 5.1, and RRSAF requires both DB2 for OS/390 Version 5.1 and OS/390 RRS, which is included with OS/390 Version 2 Release 5 and higher.

For Java servlets using JDBC or SQLJ on OS390, resource protection is very important. When making a connection to DB2, the connection can be made either in the servlet `init()` method or in the `doGet` or `doPost` method.

- By performing the connection in the `doGet` or `doPost` method, the connection and disconnection to DB2 is performed for every client request. In this scenario, use either CAF or RRSAF.
- By using the `init()` method to connect to DB2, the connection is only made at the initialization time (or load time) of the servlet. In this scenario, you can only use RRSAF. RRSAF is the facility that allows you to maintain a connect for the life of the servlet.

For more information on JDBC or SQLJ, refer to *DB2 for OS/390 Application Programming Guide and Reference for Java Version 5*, SC26-9547, which is available from URL:

```
http://www.ibm.com/software/data/db2/os390/sqlj.html
```

3.3.2 CICS and Java

CICS on OS/390 supports Java in two ways:

1. External Java applications can communicate with a CICS transaction via a connector, called the CICS Gateway for Java. The CICS Gateway for Java can accept input from remote clients via TCP/IP, or from a “local” client on the same OS/390.

   The local Java client can be a Java servlet, a JavaServer Page or a Java application. Logical three-tier applications can be created easily by calling the gateway from a servlet or JavaServer Page.

2. CICS transactions can be written in the Java language and can be run as a CICS transaction in CICS Transaction Server 1.3. In this case the Java language can be used as a substitute for C, COBOL or PL/1.

More details about the CICS Gateway for Java can be found in Chapter 10, “Develop Java Solutions for CICS on OS/390” on page 155. Regarding the CICS support for transactions written in Java, another redbook is planned to be published in early 1999.

3.3.3 MQSeries

There are two packages that are relevant if you want to build applications on OS/390:

- MQSeries Bindings for Java on OS/390

  A Java program can communicate with an MQSeries Manager using the MQSeries Bindings for Java on OS/390[3] .

  As MQSeries has connectivity options to CICS, IMS and DB2, it is not hard to establish a connection between a Java program on one side and another program in any of the MQSeries-supported subsystems on the other side.

  The Java client program can be a servlet or a JavaServer Page or a Java application.

  Refer to 11.3, “Access to IMS Using a Servlet/MQI” on page 235 for details on the MQSeries Bindings for Java.

  The beta package can be downloaded from URL:

  ```
  http://www.ibm.com/software/ts/mqseries/beta/mqmvsjb.html
  ```

- MQSeries Client for Java

  This package is discussed in detail in *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142. It gives you APIs to use on the client side, from both applets and Java applications, that allow you to send and receive messages using the MQ protocol. You will not need any code on your local hard disk to use this package. The package is fully Java-enabled and can be dynamically donwloaded into your browser.

3.3.4 IMS

IMS supports Java directly by means of a connector called TCP/IP OTMA Connection (TOC). The IMS TOC converts incoming TCP/IP messages into the IMS-specific Open Transaction Manager Access (OTMA) protocol. APIs are available for the Java client program to build the incoming messages and receive back the result messages. The IMS TOC runs on OS/390, preferably on the same system as the back-end IMS system.

Another “connector” is the IMS/ESA OTMA Callable Interface, program number 5655-158. This interface provides APIs that can be used to communicate with IMS from a C program using the OTMA protocol. No direct support from Java is available yet, but you can consider writing your own Java classes using JNI calls to a C program issuing the C/I calls.

3.3.5 MVS Datasets

Access to MVS datasets from Java is currently supported on OS/390 by the Java Record IO (JRIO) product. At the time of writing, the product is in a “technology preview” status and is downloadable from URL:

```
http://www.ibm.com/s390/java/jrio.html
```

JRIO supports the following datasets:

[3] The MQSeries Bindings for Java on OS/390 are in beta at the time of writing.

- Sequential
- Partitioned
- Entry sequence VSAM
- Key sequence VSAM

Of course, the JRIO classes can be used in servlets and JavaServer Pages, making it easy to create a Web interface for your MVS datasets.

3.4 CORBA Server on OS/390

Common Object Request Broker Architecture (CORBA) is the standard distributed object architecture developed by the Object Management Group consortium (OMG). Since OMG was founded, its mission has been to define open standards in software development so that objects written by different vendors in different languages, running on any platform, could interoperate in a distributed environment.

IBM plans to introduce CORBA for OS/390 as a component of IBM Component Broker for OS/390.

Refer to *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142, for more information.

3.5 RMI Server on OS/390

Java uses a distributed computing mechanism called Remote Method Invocation (RMI). Using RMI, you can use a remote object on a server from a client, send objects from a client to a server where they can run, and implement easily maintainable, distributed objects across a corporation or Internet.

RMI can be used on OS/390 today; however, RMI currently has no security model. The use of RMI should be restricted to secured cooperative environments, such as corporate intranets. Sun will include Internet Inter-ORB Protocol (IIOP) in the next iterations of RMI, so that RMI communications will benefit from the security model of CORBA. Refer to *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142, for more information.

Part 2. Configure Java Application Environment on OS/390

In this part we describe the minimum configuration you need on OS/390 to run Java client/server applications. Within the context of this book we assume that you will need a Webserver on OS/390.

Of course, you can also build applications by means of your own Java application server on OS/390 that handles the traffic with the clients and the access to files, databases and transactions, but the Webserver will give you many standard features that you otherwise would have to take care of yourself.
The latest version of WebAS supports both servlets and JavaServer Pages. You can call Beans from servlets and JSPs that can do anything a regular Java application could do.

The configuration for the Webserver is described in Chapter 4, "Configuration of the OS/390 Web Server" on page 29.

In this part we also describe the configuration for the latest version of VisualAge for Java: VisualAge for Java, Enterprise Edition for OS/390. VisualAge for Java, Enterprise Edition for OS/390 goes beyond the workstation and adds new features on OS/390. This gives you a true cross-platform development environment for Java specifically supporting the OS/390 developer.

5.2, "VisualAge for Java, Enterprise Edition for OS/390" on page 58 describes the configuration of the OS/390-related components and Chapter 6, "Configuring VisualAge for Java on the Workstation" on page 71 gives you all the details you need in order to set up the workstation to use the OS/390 features, called ET/390.

Chapter 4. Configuration of the OS/390 Web Server

In this chapter we discuss the configuration of the Lotus Domino Go Webserver Release 5.0 including ServletExpress and WebSphere Application Server for OS/390 V1.1.

Important

Before reading this chapter, find out which Webserver product you want to configure:

- Domino Go Webserver Version 5.0, including ServletExpress, or
- Domino Go Webserver Version 5.0, including WebSphere Application Server Version 1.1

4.1 Configuring Lotus Domino Go Webserver Release 5.0

Once the installation tasks for the Domino Go Webserver have been completed and some basic configuration done, you are ready to start serving HTML pages to your clients. The following section contains additional tips for configuring Domino Go Webserver Version 5.

4.1.1 Setting Up Server Configuration Files

The Lotus Domino Go Webserver Release 5.0 configuration file (often referred to as the httpd.conf file) may be located in any HFS directory. If you plan to run multiple servers, you will need multiple configuration files, one per server. You can place all of these in the /etc directory, or you can have them located in their own managed directories, such as /web/server1/httpd.conf and /web/server2/httpd.conf.

In managing your configuration file, you can either use the online Configuration and Administration forms (the server must be running to use these), or you can edit the configuration file manually.

We assume that you have started your server using the default IBM-supplied configuration file. You can now continue using this file and customize it to your needs. Because of the many parameter and directive changes from release to release, you cannot use a downlevel copy of the httpd.conf file with a new level of the Domino Go Webserver. If migrating to a new release, it is best to start out with the sample httpd.conf file for that new release and retro-fit your local customization to it.

After installation, your server has one authorized user ID that can be used to access the Configuration and Administration forms. By default, the authorized user ID is WEBADM. If you selected a different user ID during Domino Go Webserver installation, you should edit the httpd.conf file and change all occurrences of *WEBADM,webadm* to the user ID you selected.

4.1.1.1 Basic Single Server Configuration

If you followed the standard installation procedure for the Domino Go Webserver, no customization work is needed to enable the Webserver to run.

Standard, in this case, means the following:

- The Webserver files have been installed in /usr/lpp/internet/server_root Check if Frntpage.html exists in /usr/lpp/internet/server_root/pub
- You are using the default TCP/IP port (80) to access the Webserver

If you start up your Webserver, you should be able to access it using a Web browser through TCP/IP. However, you will be prompted to provide a valid OS/390 user ID and a password because the initial setting requires this.

The following describes some of the first changes you will probably want to make to the IBM-supplied default configuration file.

Specify the Default Access Control User ID: If you wish to allow access to your Webserver to anyone, without the need for user verification, you might want to change the "UserID" directive from

```
UserID   %%CLIENT%%
```

to

```
UserID   PUBLIC
```

This assumes that you are using an OS/390 (RACF) user ID of PUBLIC as your default access user ID.

Restarting the Webserver after making this change should enable you to access the Webserver's home page without providing a user ID.

Specify the Authorized Administrator User ID: After installation, your server has one authorized user ID that can be used to access the Configuration and Administration forms. By default, the authorized user ID is WEBADM. If you selected a different user ID during Domino Go Webserver installation, you should edit the httpd.conf file and change all occurrences of WEBADM and webadm to the user ID you selected. Figure 13 shows the default statements that you will need to update if you selected a different user ID.

However, access to the configuration page in order to work with the remote administration forms will still be protected. This will be enforced by the definition shown in Figure 13.

```
Protection IMW_Admin {
        ServerId        IMWEBSRV_Administration
        AuthType        Basic
        PasswdFile      %%SAF%%
        Mask            WEBADM,webadm
}

Protect /admin-bin/* IMW_Admin WEBADM
Protect /reports/*   IMW_Admin WEBADM
Protect /Usage*      IMW_Admin WEBADM
```

Figure 13. Webserver Administration Protection Directives

SSL Security Customization for Non-SSL Mode: If you followed our recommendation and started with the supplied sample configuration file as shipped, the configuration directives shown in Figure 14 on page 31 appear in your configuration file. They are related to SSL security.

```
SSLClientAuth   off
sslmode         on
sslport         443
normalmode      on
keyfile         key.kdb
```

Figure 14. SSL Directives in IBM-Supplied Configuration File (httpd.conf)

keyfile

There is no requirement to change these parameters.

Note: However, if you leave these parameters as is, you will get the following error message, which can be seen in the httpd.errors log when starting the server:

```
+0500  SSL support initialization failed,
server will run only in non-secure mode without listening on ssl port
```

This message indicates that no setup has been done for SSL. However, the Webserver can still be used as a normal “non-secure” server.

You can avoid getting these errors by commenting out the directives shown in Figure 14.

If you want to set up SSL, refer to *Enterprise Web Serving with the Lotus Domino Go Webserver for OS/390*, SG24-2074.

4.1.2 Locating Your Web Content

This section shows you how and where to set up your own Web content without affecting or being affected by the IBM Web content that is provided with Domino Go Webserver. This is important because if you place your own Web content in the file system that IBM provides, you might find it difficult to upgrade to another version of the Webserver, or even apply maintenance.

4.1.2.1 Standard Lotus Domino Go Webserver Release 5.0 Content Setup

The Lotus Domino Go Webserver Release 5.0 server is defined to look for Web pages in /usr/lpp/internet/server_root/pub, with some exceptions for “special pages” (such as remote configuration forms). This is enforced by the definitions shown in Figure 15 on page 32.

```
#
Pass           /admin-bin/webexec/*  /usr/lpp/internet/server_root/admin-bin/webexec
Exec           /cgi-bin/*       /usr/lpp/internet/server_root/cgi-bin/*
Exec           /admin-bin/*     /usr/lpp/internet/server_root/admin-bin/*
Exec           /Docs/admin-bin/*     /usr/lpp/internet/server_root/admin-bin/*

#       These are the pass rules for server administration
#
Pass           /icons/*         /usr/lpp/internet/server_root/icons/*
Pass           /Admin/*.jpg     /usr/lpp/internet/server_root/Admin/*.jpg
Pass           /Admin/*.gif     /usr/lpp/internet/server_root/Admin/*.gif
Pass           /Admin/*.html    /usr/lpp/internet/server_root/Admin/*.html
Pass           /Docs/*          /usr/lpp/internet/server_root/Docs/*
Pass           /reports/javelin/* /usr/lpp/internet/server_root/pub/reports/javelin/*
Pass           /reports/java/*  /usr/lpp/internet/server_root/pub/reports/java/*
Pass           /reports/*       /usr/lpp/internet/server_root/pub/reports/*
Pass           /img-bin/*       /usr/lpp/internet/server_root/img-bin/*
# *** ADD NEW PASS RULES HERE ***
Pass           /*               /usr/lpp/internet/server_root/pub/*
```

Figure 15. Standard Web Content Setup Directives

These `Exec` and `Pass` statements force the Webserver to search for content at particular locations in the HFS. This approach also has the effect of hiding the structure of your file system from users of Web browsers.

The following HTML example can be used to show how the Webserver resolves the requests. Assume that a Web browser makes a request for the home page of a particular site. That home page is shown in Figure 16.

```
<html>
<head>
<TITLE>Lotus Domino Go Webserver</TITLE>
</head>
<body bgcolor="#FFFFFF">

<img src="/Admin/lgmast.gif" alt="Lotus Domino Go Webserver">   1
<hr>
<DL>
<DT>
<a href="/admin-bin/webexec/cfgstart.html">CONFIGURATION AND ADMINISTRATION
FORMS</a>
<DD>To set up, configure, and administer the Lotus Domino Go Webserver.
<br>
<font size="-1"><A HREF="tunetips.html"><b>Tune your browser first.</b></A></fon
<P>
<DT>
<A HREF="http://www.ics.raleigh.ibm.com/lotusgowebserver/">LOTUS DOMINO
GO WEBSERVER WEB SITE</a>
<DD>
To find useful information.<P>
<DT>
<A HREF="http://www.support.lotus.com/css/domgoserv.htm">LOTUS DOMINO GO WEBSERV
<DD>
To get help when you need it.
<P>
<DT><A HREF="/Docs/2tabcontents.html">HOW DO I GET STARTED?</A>
<DD>Serving pages, and other basic tasks
   .
   .
   .
   .
```

Figure 16. HTML Example - Frntpage.html

This HTML file is found by the Webserver and is sent to the client.

The client browser reads the file and finds the first reference statement in this HTML file (line **1** in Figure 16). This causes another GET request to be made to the server. The request would look like this:

```
GET /Admin/lgmast.gif HTTP/1.0
```

The server receives this request, looks for a matching configuration directive, and in this case finds the following as shown in Figure 15 on page 32:

```
Pass /Admin/*.gif     /usr/lpp/internet/server_root/Admin/*.gif
```

The Webserver therefore translates the request to:

```
/usr/lpp/internet/server_root/Admin/lgmast.gif
```

It is important that you understand how this example was mapped by the Webserver to actual file system locations. Each request is mapped according to the HTTP rules and to the configuration directives (in a top-to-bottom, first match approach).

4.1.2.2 Web Content Setup Recommendations

Our recommendations are:

- Do not modify IBM-provided default content.
- Do not mix IBM-provided default content with your own content.
- Do not use the following locators assumed by IBM default content for your own URLs:

/cgi-bin/	Used to address CGI programs.
/admin-bin/	Used to address CGI programs for remote administration purposes. WEBADM password needed to access.
/Docs/admin-bin/	Used to address CGI programs for documentation.
/icons/	Used to address server icons for tree views and other purposes.
/Admin/	Used for remote administration purposes.
/Docs/	Used for the standard documents provided with the Webserver.
/img-bin/	Used for clickable icons.
/reports/	Used to access reports generated by service tasks from the log files.
/reports/javelin/	Used to access reports generated by Web Traffic Express.
/reports/java/	Used to access reports generated by Java.

- Use a separate HFS data set for your own Web content, mounted at a different mount point than the /usr/lpp/internet/server_root path.

The following procedure shows you an easy way to start with your Webserver setup:

1. Allocate a new HFS data set for your Web content.
2. Mount this HFS to a mount point such as `/web/server1/`.
3. Create the following directories:

/web/server1/pub	Default directory containing your HTML, GIF, and other files.
/web/server1/our-cgi	Default directory containing your CGI programs.
/web/server1/reports	Directory containing the server reports.
/web/server1/logs	Directory containing the server logs.

4. We also recommend that you allocate and mount a separate HFS for each of the server logs and reports.

5. Modify the `Exec` and `Pass` statements in the httpd.conf file as shown in Figure 17.

```
#
Exec  /our-cgi/*       /web/server1/our-cgi/*
Exec  /cgi-bin/*       /usr/lpp/internet/server_root/cgi-bin/*
Exec  /admin-bin/*     /usr/lpp/internet/server_root/admin-bin/*
Exec  /Docs/admin-bin/*    /usr/lpp/internet/server_root/admin-bin/*

#       These are the pass rules for server administration
#
Pass  /icons/*         /usr/lpp/internet/server_root/icons/*
Pass  /Admin/*.jpg     /usr/lpp/internet/server_root/Admin/*.jpg
Pass  /Admin/*.gif     /usr/lpp/internet/server_root/Admin/*.gif
Pass  /Admin/*.html    /usr/lpp/internet/server_root/Admin/*.html
Pass  /Docs/*          /usr/lpp/internet/server_root/Docs/*
Pass  /img-bin/*       /usr/lpp/internet/server_root/img-bin/*
Pass  /reports/javelin/*  /usr/lpp/internet/server_root/pub/reports/javelin/*
Pass  /reports/java/*  /usr/lpp/internet/server_root/pub/reports/java/*
#Pass  /reports/*       /usr/lpp/internet/server_root/pub/reports/*
Pass  /reports/*       /web/server1/reports/*
#Pass  /*               /usr/lpp/internet/server_root/pub/*
Pass  /Server/*        /usr/lpp/internet/server_root/pub/*
Pass  /*               /web/server1/pub/*
```

Figure 17. Single Server - Modified Web Content Setup Directives

This modification will force the Webserver to search in your /web/server1/pub directory for everything but the IBM-provided default content.

The HTML example in Figure 18 shows you how to set up your own home page using the settings shown in Figure 17, and still be able to access the remote administration forms as before. This file should be called “index.html” or “Welcome.html” and should be placed in /web/server1/pub.

```
<html><head>
<title>DGW Project - Roland's Web Server</title>
</head><body bgcolor="#FFFFFF">
<h1>Welcome to my home page </h1>
This is Roland Trauner's Web server running on OS/390.
 ...
<hr>
Follow this link to access the
<a href="/Server/">Remote Server Administration</a>.
</html>
```

Figure 18. Single Server - HTML Example for a Home Page (index.html)

6. Change the `ServerRoot` statement to `/web/server1`.

7. Change the logging and reporting statements to the appropriate directories, as shown in Figure 19 on page 35.

```
AccessLog       /web/server1/logs/httpd-log
RefererLog      /web/server1/logs/referer-log
AgentLog        /web/server1/logs/agent-log
ErrorLog        /web/server1/logs/httpd-errors
CgiErrorLog     /web/server1/logs/cgi-error

AccessLogArchive           none
AccessLogExpire            15
AccessLogSizeLimit         0
ErrorLogArchive            none
ErrorLogExpire             15
ErrorLogSizeLimit          0

AccessReportRoot           /web/server1/reports
AccessReportDoDnsLookup    Off
```

Figure 19. Single Server - Logging and Reporting Changes to httpd.conf

8. There are more modifications you will do to the server configuration file, depending on the functions you wish to use. Figure 20 shows some changes we made.

```
SMF                    All
SMFRecordingInterval   00:15

SNMP                   On
SNMPCommunity          public
WebMasterEMail         trauner@de.ibm.com

service /cgi-bin/apicounter* /usr/lpp/internet/bin/htcounter.so:HTCounter*
service /cgi-bin/datetime*   /usr/lpp/internet/bin/htcounter.so:HTCounter*
service /cgi-bin/text2gif*   /usr/lpp/internet/bin/htcounter.so:HTCounter*

UseMetaFiles           Off
DirAccess              Off

CacheLocalFile   /web/server1/pub/index.html

PersistTimeout   10 seconds
```

Figure 20. Single Server - Additional Modifications to httpd.conf

Other configuration file changes depend on the purpose and content of your Webserver. You may also need to change some parameters in order to tune the server. For more detailed information on all valid configuration file directives, see *Lotus Domino Go Webserver: Webmaster's Guide Release 5.0 for OS/390*, SC31-8691.

4.2 WebSphere Application Server (WebAS) and ServletExpress (SE)

In this section we discuss the configuration of WebSphere Application Server for OS/390 V1.1 (WebAS) and ServletExpress (SE). Most of the configuration options are the same between WebSphere Application Server for OS/390 V1.1 and ServletExpress. For this reason, we will refer to WebAS throughout this chapter, unless there are differences.

Configuring WebAS can be performed by using WebAS Manager or by updating WebAS properties files. The WebAS Manager provides a graphical user interface using a Java Applet application. The interface can be used for configuring and managing servlets running on the WebSphere Application Server.

Most changes you make to configuration parameters from within the WebAS Manager take effect immediately and do not require you to restart the Webserver, eliminating server downtime. The WebAS Manager applet interface is, in fact, an example of an applet-to-servlet communication. The applet communicates with a servlet running under the WebSphere Application Server which manages the Webserver servlet environment. All this is done dynamically and “on the fly.”

Note: Some changes will require the Webserver to be restarted. These include any changes to the listening port, the parameters on the Basic Setup page, and selective updates made to the jvm.properties file to control logging. Updating the WebAS properties files directly will require a restart of the Webserver. We discuss some useful direct updates at the end of this section.

4.2.1 Configuring WebSphere Application Server for OS/390 V1.1

WebSphere Application Server for OS/390 V1.1 is shipped and packaged as part of the Lotus Domino Go Webserver Release 5.0 if you run OS/390 Release 5 or 6, and part of IBM HTTP Server Version 5.1 if you run OS/390 Release 7. The installation instructions are included in the WebAS Directory. Customers that currently have Lotus Domino Go Webserver Release 5.0 with SE can order WebSphere Application Server for OS/390 V1.1 to replace SE. See the following documents for details:

- *Program Directory for Domino Go Webserver for OS/390*, GI10-6780-00
- *Domino Go Webserver 5.0 for OS/390: Planning for Installation*, SC31-8690
- *Domino Go Webserver 5.0 for OS/390: Getting Started*, included in the package

During installation, the WebAS product, by default, is loaded in its own HFS dataset. It is mounted at /usr/lpp/WebSphere for WebAS and /usr/lpp/ServletExpress for SE.

Note: It is recommended that you first customize and start the Domino Go Webserver without WebAS to test that you can successfully serve HTML pages before you start the WebAS customization.

After you have completed the SMPE work for WebSphere Application Server, you will need to run the WebAS post-installation tool /usr/lpp/WebSphere/AppServer/config/postinstall.sh for WebAS or /usr/lpp/ServletExpress/bin/SEconfig for SE. The tool is used to add specific WebAS directives to your httpd.conf file and setup WebAS or SE properties files.

For more information, refer to the installation manual for WebSphere Application Server or ServletExpress.

The postinstall.sh for WebAS will add the directives to your httpd.conf file, as shown in Figure 21.

```
ServerInit  /usr/lpp/WebSphere/AppServer/lib/libadpter.so:AdapterInit
 1  /usr/lpp/WebSphere/AppServer/properties/server/servlet/servletservice/
 1  /servletservice/jvm.properties
 .
 .
Service   /*.jhtml     /usr/lpp/WebSphere/AppServer/lib/libadpter.so:AdapterService
Service   /*.jsp       /usr/lpp/WebSphere/AppServer/lib/libadpter.so:AdapterService
Service   /servlet/*   /usr/lpp/WebSphere/AppServer/lib/libadpter.so:AdapterService

Pass  /IBMWebAS/samples/*     /usr/lpp/WebSphere/AppServer/samples/*
Pass  /IBMWebAS/Docs/*        /usr/lpp/WebSphere/AppServer/system/admin/*
Pass  /IBMWebAS/doc/*         /usr/lpp/WebSphere/AppServer/doc/*
Pass  /IBMWebAS/system/admin/*    /usr/lpp/WebSphere/AppServer/system/admin/*
Pass  /IBMWebAS/*             /usr/lpp/WebSphere/AppServer/web/*

ServerTerm  /usr/lpp/WebSphere/AppServer/lib/libadpter.so:AdapterExit
```

Figure 21. Changes in httpd.conf after Running postinstall.sh

Notes:

1 These lines have been split for redbook printing purposes; however, in the real httpd.conf file they must be typed on one single line.

The SEconfig tool for SE will add the directives to your httpd.conf file, as shown in Figure 22.

```
ServerInit  /usr/lib/libadpter.so:AdapterInit
 1  /usr/lpp/ServletExpress/properties/server/ServletExpress/servletservice
 .
 .
Service       /*.jhtml     /usr/lib/libadpter.so:AdapterService
Service       /*.jsp       /usr/lib/libadpter.so:AdapterService
Service       /servlet/*   /usr/lib/libadpter.so:AdapterService
 .
 .
Pass    /ServletExpress/resources/*  /usr/lpp/ServletExpress/web/resources/en_US/*
Pass    /ServletExpress/Docs/*       /usr/lpp/ServletExpress/system/en_US/admin/*
Pass    /ServletExpress/*            /usr/lpp/ServletExpress/web/*
 .
ServerTerm   /usr/lib/libadpter.so:AdapterExit
```

Figure 22. Changes in httpd.conf by the SEconfig Tool

Notes:

1 These lines have been split for redbook printing purposes; however, in the real httpd.conf file they must be typed on one single line.

The postinstall and SEconfig tools also update the jvm.properties file with local path information such as JAVA_HOME and the WebAS installation path.

Note: The JAVA_HOME value is obtained from the current setting of the JAVA_HOME environment variable in your shell session. You can issue the "echo $JAVA_HOME" command in the shell to verify proper directories of the JDK. In addition, you can issue "java -fullversion" to verify that you have the desire release. At the time of writing, WebAS supported JDK 1.1.1 and 1.1.4 as well as 1.1.6.

4.2.2 WebAS Properties Files

The WebAS properties files are used to control the functionality of the WebSphere Application Server.

The properties files can be updated either by WebAS Manager or by manually updating the files directly. These files are located in directory: /usr/lpp/WebSphere/AppServer/properties/server/servlet/servletservice for WebAS and directory /usr/lpp/ServletExpress/properties/server/ServletExpress/servletservice for SE.

Because the properties files are part of the /usr/lpp/WebSphere or /usr/lpp/ServletExpress, they may be overlaid if you install maintenance or upgrade with SMPE. Since these files will be customized either through WebAS or SE Manager or manually, we suggest that you take a full copy of these libraries and leave SMPE libraries unchanged. For example, you can copy /usr/lpp/WebSphere/* to /web/WebSphere with the UNIX command:

```
cp -R /usr/lpp/WebSphere /web
```

You might want to mount a new HFS file system at the /web/WebSphere mount point. In addition, if you copy these files, you will need to update the httpd.conf file and new jvm.properties file.

If you do copy the WebAS files, you will need to update the httpd.conf file to point to the new jvm.properties file. The required change is made in Figure 23. The example assumes you copied the files to the `/web/WebSphere` directory.

```
ServerInit  /web/WebSphere/AppServer/lib/libadpter.so:AdapterInit
 1  /web/WebSphere/AppServer/properties/server/servlet/servletservice/
 1  /servletservice/jvm.properties
```

Figure 23. Example of Updated htpd.conf File

Notes:

1 These lines have been split for redbook printing purposes; however, in the real httpd.conf file they must be typed on one single line.

The jvm.properties file is used for configuration properties for the JVM and plugin DLLs at startup.

The jvm.properties file is pointed to by a ServerInit statement in the httpd.conf file for the WebSphere Application Server for OS/390 V1.1.

Figure 24 on page 39 gives an example of a jvm.properties file for WebAS where we copied /usr/lpp/WebSphere to /web/WebSphere.

```
# @(#)jvm.properties.1.81 97/12/02
#
# Configuration properties for JVM and plugin dll start-up
#

# System Properties
IBMWebASVersion=1.0.0
server.root=/web/WebSphere/AppServer
server.name=servlet
server.description=IBMWebAS
java.compiler=


# NCF Properties
ncf.service.name=servletservice
ncf.service.class=com.ibm.servlet.service.SEServlet
ncf.plugin.classname=com.ibm.servlet.ServletSystem

#
# Enable native DLL plugin logging by setting 'ncf.native.logison'
# to 'true'.  Change 'ncf.native.logfile' to the <fully-qualified >
# path of an alternate file location if desired.
#
ncf.native.logison=true
ncf.native.logfile=/web/webas/native.log

#
# Enable JVM logging by setting 'ncf.jvm.stdoutlog.enabled'
# to true. Change 'ncf.jvm.stdoutlog.file' to 'false' to write
# to a Java debugging console or 'true' for output to a log file.
# Uncomment the line for 'ncf.jvm.stdoutlog.popup', thus setting it
# to '2' to display the combined ResourceUsage/EnableTrace/Console
# popup. Otherwise, just the console popup is displayed.
# Change 'ncf.jvm.stdoutlog.filename' to the <fully-qualified>
# path of an alternative file location if desired.
#
ncf.jvm.stdoutlog.enabled=true
#ncf.jvm.stdoutlog.popup=2
ncf.jvm.stdoutlog.file=true
ncf.jvm.stdoutlog.filename=/web/webas/ncf.log

# NCF - Admin Service Properties for BasicNCFConfig Applet
ncf.jvm.classpath=/web/WebSphere/AppServer/lib/ibmwebas.jar:
   1 /web/WebSphere/AppServer/lib/jst.jar:
   1 /web/WebSphere/AppServer/lib/jsdk.jar:
   1 /web/WebSphere/AppServer/lib/x509v1.jar:
   1 /web/WebSphere/AppServer/lib:
   1 /web/WebSphere/AppServer/web/admin/classes/seadmin.jar:
   1 /web/WebSphere/AppServer/web/classes:
   1 /usr/lpp/java/J1.1/lib/classes.zip:
   1 /usr/lpp/db2/db2510/classes/db2jdbcclasses.zip:
   1 /usr/lpp/db2/db2510/classes/db2sqljclasses.zip:
   1 /usr/lpp/db2/db2510/classes/db2sqljruntime.zip
```

Figure 24 (Part 1 of 2). Example of a jvm.properties File

```
ncf.jvm.libpath=/usr/lpp/java/J1.1/lib:
   1 /usr/lpp/java/J1.1/lib/mvs/native_threads:
   1 /web/WebSphere/AppServer/lib:
   1 /usr/lib
   1 /usr/lpp/internet/bin:
   1 /usr/lpp/db2/db2510/lib
ncf.jvm.path=/usr/lpp/java/J1.1/bin
ncf.jvm.use.system.classpath=false

# Max Java Heap Size
ncf.jvm.mx=67108864

# Properties for Domino Go
ncf.native.httpd.cnf.path=/web/webas/httpd.conf

# OS/390 WebSphereAS 1.0 Only!
#
# The ncf.jvm.threads.max property is used to increase the number of threads
# that the JVM is allowed to create for threaded servlets, chaining servlets
# or filtering servlets. If any of these types of servlets are being executed
# this property will need to be set to accommodate the threading needs.
#ncf.jvm.threads.max=5

# OS/390 WebSphereAS 1.0 Only!
#
# Define the OS/390 OE native log debug level. The
# following define the current debug levels:
#
#   Level 0
#   ---------------------------------------------------
#   - OS/390 OE specific tracing off.
#
#   Level 1
#   ---------------------------------------------------
#   - Trace messages in JNI wrappers.
#     calls made during servlet processing.
#   - Trace messages in OE Data Conversion Utility
#     routines.
#
#   Level 2 (Includes Level 1.)
#   ---------------------------------------------------
#   - Trace messages in the ICS native library for JNI
#     calls made to convertgetBytes_AtoE().
#
#ncf.native.os390.debug=0
```

Figure 24 (Part 2 of 2). Example of a jvm.properties File

Notes:

> **1** These lines have been split for redbook printing purposes; however, in the real jvm.properties file they must be typed on one single line.

Figure 25 on page 41 gives an example of a jvm.properties file for SE.

```
# Configuration properties for JVM and plugin dll start-up
#

# System Properties
ServletExpressVersion=1.0.0
server.root=/web/ServletExpress
server.name=ServletExpress
java.compiler=

# NCF Properties
ncf.service.name=servletservice
ncf.service.class=com.ibm.ServletExpress.service.SEServlet
ncf.plugin.classname=com.ibm.ServletExpress.ServletSystem

#
# Enable native DLL plugin logging by setting 'ncf.native.logison'
# to 'true'.  Change 'ncf.native.logfile' to the <fully-qualified >
# path of an alternate file location if desired.
#
ncf.native.logison=true
ncf.native.logfile=/web/java14/native.log
#
# Enable JVM logging by setting 'ncf.jvm.stdoutlog.enabled'
# to true. Change 'ncf.jvm.stdoutlog.file' to 'false' to write
# to a Java debugging console or 'true' for output to a log file.
# Change 'ncf.jvm.stdoutlog.filename' to the <fully-qualified >
# path of an alternate file location if desired.
#
ncf.jvm.stdoutlog.enabled=true
ncf.jvm.stdoutlog.file=true
ncf.jvm.stdoutlog.filename=/web/java14/ncf.log

# NCF - Admin Service Properties for BasicNCFConfig Applet
ncf.jvm.classpath=/usr/lpp/Java/J1.1/lib/classes.zip:
   1 /usr/lpp/db2/db2510/classes/db2jdbcclasses.zip:
   1 /usr/lpp/ServletExpress/lib/servexp.jar:
   1 /usr/lpp/ServletExpress/lib/jsdk.jar:
   1 /usr/lpp/ServletExpress/lib/x509v1.jar:
   1 /usr/lpp/ServletExpress/lib/jst.jar:
   1 /usr/lpp/ServletExpress/lib:
   1 /usr/lpp/ServletExpress/web/admin/classes
ncf.jvm.libpath=/usr/lpp/Java/J1.1/lib:
   1 /usr/lpp/Java/J1.1/lib/mvs/native_threads:
   1 /usr/lpp/ServletExpress/lib:
   1 /usr/lib
   1 /usr/lpp/internet/bin:
   1 /usr/lpp/db2/db2510/lib
ncf.jvm.path=/usr/lpp/java14/J1.1/bin
ncf.jvm.use.system.classpath=false
```

Figure 25 (Part 1 of 2). Example of jvm.properties File in ServletExpress

```
# Max Java Heap Size
ncf.jvm.mx=67108864
#
# Properties for Netscape webserver V2.01 on AIX or SOLARIS
#
#ncf.native.outofproc.runscript=/usr/bin/servlet_eng_runner.sh
#ncf.native.outofproc.port=8090
#ncf.native.outofproc.idstring="servexp"
#ncf.native.outofproc.netscapemime=<netscape_root>/config/mime.types

#
# Properties for Apache webserver on AIX or SOLARIS
#
#ncf.native.apache.outofproc.runscript=/usr/bin/apache_servlet_eng_runne
#ncf.native.apache.outofproc.port=8082
#ncf.native.apache.outofproc.idstring="apache-servlet-engine"

# Properties for IIS
ncf.native.iis.extensionloc=/sePlugins/iis20.dll

# Properties for Domino Go
ncf.native.httpd.cnf.path=/web/java14/httpd.conf
```

Figure 25 (Part 2 of 2). Example of jvm.properties File in ServletExpress

Notes:

1 These lines have been split for redbook printing purposes; however, in the real jvm.properties file they must be typed on one single line.

ServletExpress supports native DLL logging and Java standard out logging. Native DLL logging logs messages produced by the Webserver before Java is invoked.

The Java standard out logging is used for any Java system.out and system.err print. Java standard out is very useful for adding logic in Java servlets for debugging purposes.

Enable native DLL plugin logging by:

- Setting "ncf.native.logison" to "true."
- Changing "ncf.native.logfile" to the fully-qualified path of an alternate file location if desired, like `/web/native.log`.

The changed statements are as follows:

```
ncf.native.logfile=/web/native.log
ncf.native.logison=true
```

Enable JVM logging by:

- Setting "ncf.jvm.stdoutlog.enabled" to "true"
- Changing "ncf.jvm.stdoutlog.file" to "false" (to write to a Java debugging console) or "true" (for output to a log file)
- Changing "ncf.jvm.stdoutlog.filename" to the fully-qualified path of an alternate file location if desired.

The changed statements are as follows:

```
ncf.jvm.stdoutlog.enabled=true
ncf.jvm.stdoutlog.file=true
ncf.jvm.stdoutlog.filename=/web/ncf.log
```

The WebAS ncf.server.root parameter identifies WebAS's root directory for all the system and properties files. WebAS is sensitive to the location of all WebAS files, except the ncf and native logs.

```
Ncf.server.root=/<your_new_root>/WebSphere
```

The servlets.properties file is used to configure the environment for servlets.

The servlets.classpath parameter is used to specify from which directory servlets need to be loaded. The specified list of directories is useful in providing addition to the <seroot>/servlets directory. Unlike the <seroot>/servlet directory, these specified directories will automatically reload if the class file is changed on disk.

```
Servlets.classpath=/usr1/servlets:/usr2/servlets:
```

The admin_port.properties file is used to store the port that ServletExpress is listening on. The default port is 9090.

By changing this parameter, ServletExpress manager will listen on the specified port. Unlike the other properties files, the admin_port.properties is located in /usr/lpp/WebSphere/AppServer/properties/server/servlet/adminservice for WebAS and /ServletExpress/properties/server/ServletExpress/adminservices for SE.

The entry for the portnumber looks as follows:

```
endpoint.main.port=9090
```

4.2.3 Verifying a Successful Startup of WebAS

WebAS uses the services of the Java Virtual Machine, TCP/IP and OS/390 UNIX System Services.

When you start the Webserver, check the verbose trace (-vv); you should see messages similar to what is shown in Figure 26 on page 44.

```
GWAPI:  HTTPD_extract() called
GWAPI:  HTTPD_extract() args..... name= INIT_STRING ; name size= 11
GWAPI:  HTTPD_extract() args..... buffer= 0x7c82988 ; buffer size= 1023
GWAPI:  HTTPD_extract()...  Looking up server and CGI variables
GWAPI:  HTTPD_extract()... successful with value= "/web/candy/jvm.properties"
GWAPI:  HTTPD_extract() called
GWAPI:  HTTPD_extract() args..... name= SERVER_SOFTWARE ; name size= 15
GWAPI:  HTTPD_extract() args..... buffer= 0x7aed288 ; buffer size= 255
GWAPI:  HTTPD_extract()...  Looking up server and CGI variables
GWAPI:  HTTPD_extract()... successful with value= "Lotus Domino Go Webserver
North American Edition for OS/390/V5R0M0"
GWAPI:  HTTPD_extract() called
GWAPI:  HTTPD_extract() args..... name= SERVER_NAME ; name size= 11
GWAPI:  HTTPD_extract() args..... buffer= 0x7aed388 ; buffer size= 255
GWAPI:  HTTPD_extract()...  Looking up server and CGI variables
GWAPI:  HTTPD_extract()... successful with value= ""
GWAPI:  HTTPD_extract() called
GWAPI:  HTTPD_extract() args..... name= SERVER_PORT ; name size= 11
GWAPI:  HTTPD_extract() args..... buffer= 0x7554f68 ; buffer size= 9
GWAPI:  HTTPD_extract()...  Looking up server and CGI variables
GWAPI:  HTTPD_extract()... successful with value= "80"
GWAPI:  HTTPD_log_error() called
GWAPI:  HTTPD_log_error() args..... value= ServletExpress native plugin
initalization went OK :-) ; value size= 54
```

Figure 26. Example of the Log when WebAS is Successfully Started

On a successful startup, you will see messages similar to what is shown in Figure 27 on page 45 in the /usr/lpp/Websphere/AppServer/logs/servlet/servletservice/event.log file for WebAS or /usr/lpp/ServletExpress/logs/ServletExpress/servletservice/event.log file for SE.

```
ServletManager.loadStartupServlets:
   invoker samPackages samMap samMsg samProfile samUsers hello snoop simple
ServletManager.instantiateServlet:
   Loaded local class class com.sun.server.http.InvokerServlet.
   com.sun.server.http.InvokerServlet: init
ServletManager.loadServlet invoker:
   class = com.sun.server.http.InvokerServlet class URL =<none>  arguments = <no
ServletManager.instantiateServlet:
   Loaded local class class com.ibm.ServletExpress.servlets.sam.ConnectCGIPackag
   com.ibm.ServletExpress.servlets.sam.ConnectCGIPackages: init
ServletManager.loadServlet samPackages:
   class = com.ibm.ServletExpress.servlets.sam.ConnectCGIPackages class
   URL =<none>  arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class com.ibm.ServletExpress.servlets.sam.ConnectCGI.
   com.ibm.ServletExpress.servlets.sam.ConnectCGI: init
ServletManager.loadServlet samMap:
   class = com.ibm.ServletExpress.servlets.sam.ConnectCGI class URL =<none>
   arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class com.ibm.ServletExpress.servlets.sam.ConnectCGIMsg.
   com.ibm.ServletExpress.servlets.sam.ConnectCGIMsg: init
ServletManager.loadServlet samMsg:
   class = com.ibm.ServletExpress.servlets.sam.ConnectCGIMsg class URL =<none>
   arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class com.ibm.ServletExpress.servlets.sam.ConnectCGIProfil
   com.ibm.ServletExpress.servlets.sam.ConnectCGIProfile: init
ServletManager.loadServlet samProfile:
   class = com.ibm.ServletExpress.servlets.sam.ConnectCGIProfile class URL =<non
   arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class com.ibm.ServletExpress.servlets.sam.ConnectCGIUser.
   com.ibm.ServletExpress.servlets.sam.ConnectCGIUser: init
ServletManager.loadServlet samUsers:
   class = com.ibm.ServletExpress.servlets.sam.ConnectCGIUser class URL =<none>
   arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class HelloWorldServlet.
HelloWorldServlet: init
ServletManager.loadServlet hello:
   class = HelloWorldServlet class URL =<none>  arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class SnoopServlet.
SnoopServlet: init
ServletManager.loadServlet snoop:
   class = SnoopServlet class URL =<none>  arguments = <none>
ServletManager.instantiateServlet:
   Loaded local class class SimpleServlet.
SimpleServlet: init
ServletManager.loadServlet simple:
   class = SimpleServlet class URL =<none>  arguments = <none>
Service started.
```

Figure 27. Example of the WebAS event.log File after a Successful Start

WebAS, by default, listens on port 9090. Use the `onetstat` command from OS/390 UNIX System Services to verify that WebAS is in a listening state, as shown in Figure 28 on page 46.

Note: Depending on how many servlets are being loaded, it may take several minutes (up to five minutes, in our system) after starting the Webserver before you see WebAS listening on port 9090.

```
RCONWAY:/u/rconway: >onetstat
MVS TCP/IP onetstat CS/390 V2R5       TCPIP Name: TCPIPOE
User Id  Conn  Local Socket            Foreign Socket         State
-------  ----  ------------            --------------         -----
WEBCANDY 06026 0.0.0.0..443            0.0.0.0..0             Listen
WEBCANDY 0602B 0.0.0.0..9090           0.0.0.0..0             Listen
WEBCANDY 06025 0.0.0.0..80             0.0.0.0..0             Listen
```

Figure 28. The onetstat Command

In addition to WebAS listening on port 9090, you must also watch for a message similar to the following before accessing your Webserver:

```
IMW3536I SA 1761607710 0.0.0.0:80 * * READY
```

4.2.4 If Something Goes Wrong

If you are having trouble getting WebAS to come up, here are some things to check out:

1. Installations with JDK 1.1.1 (9/97 or later) should not require additional maintenance. Installations using Lotus Domino Go Webserver Release 5.0 with JDK 1.1.4 require PTFs for both Lotus Domino Go Webserver Release 5.0 ServletExpress support and the JDK:

DGW. 5.0	APAR PQ18246
JDK 1.1.4	APARs OW34311, OW34445, OW34509, OW34447, OW34171, and OW33911.

   Installations with JDK 1.1.6 should not require additional maintenance.

2. Verify that external links for the ServletExpress DLLs have been defined (this is only for ServletExpress. WebAS does not use external links)..

   ```
   cd /<SEroot>/lib
   ls -al  lib*
   ```

   Expected results:

   ```
   erwxrwxrwx   /usr/lpp/ServletExpress/lib/libadpter.so ->  EJSADPTR
   erwxrwxrwx   /usr/lpp/ServletExpress/lib/libicsnativ.so ->  EJSICSNT
   ```

   Note: If these links are missing, you can generate them (while in the <SEroot> directory) by executing the following commands:

   ```
   ln -e EJSADPTR libadpter.so
   ln -e EJSICSNT libicsnativ.so
   ```

3. Enable native DLL plugin logging by:

   - Setting ncf.native.logison to `true`.
   - Changing ncf.native.logfile to the fully-qualified path of an alternate file location if desired. By default it is set to: `/usr/lpp/ServletExpress/logs/native.log`

   The changed statements are as follows:

   ```
   ncf.native.logfile=/usr/lpp/ServletExpress/logs/native.log
   ncf.native.logison=true
   ```

4. Enable JVM logging by:

   - Setting ncf.jvm.stdoutlog.enabled to `true`

- Changing ncf.jvm.stdoutlog.file to `false` (to write to a Java debugging console), or `true` (for output to a log file).
- Changing ncf.jvm.stdoutlog.filename to the fully-qualified path of an alternate file location if desired. By default it is set to: `/usr/lpp/ServletExpress/logs/ncf.log`

The changed statements are as follows:

```
ncf.jvm.stdoutlog.enabled=true
ncf.jvm.stdoutlog.file=true
ncf.jvm.stdoutlog.filename=/usr/lpp/ServletExpress/logs/ncf.log
```

Note: For ServletExpress, you may see the following WARNING message with traceback statements in the ncf.log file. It can be ignored:

```
IBM ServletExpress WARNING: Cannot load service IBM: No service class specified.
java.lang.IllegalArgumentException: No service class specified
.at com.sun.server.ServiceManager.createService(ServiceManager.java:923)
.at com.sun.server.ServiceManager.loadService(ServiceManager.java:878)
.at com.sun.server.ServiceManager.loadServices(ServiceManager.java:505)
.at com.sun.server.ServiceManager.startServices(ServiceManager.java:348)
.at com.sun.server.ServerProcess.main(ServerProcess.java:231)
.at com.ibm.ServletExpress.service.ServerProcessThread.run(ServerProcessThread)
.at java.lang.Thread.run(Thread.java)
```

5. When running Lotus Domino Go Webserver Release 5.0 with WebAS or SE support, and when the Webserver is configured with a user ID of %%CLIENT%% or a surrogate ID in the httpd.conf configuration file, clients may receive an error 500 message.

   Review the Domino Go Webserver trace log to determine if the following messages were issued for the particular client request:

   ```
   Failed access as Surrogate: <surrogate ID>, Errno: 139,
   Errno2: 0be802af, Error: EDC5139I Operation not permitted.
   IMW0241E Access denied - surrogateuser setup error.
   ```

   -OR-

   ```
   Failed access as Surrogate: <surrogate ID>, Errno: 139,
   Errno2: 090c02af, Error: EDC5139I Operation not permitted.
   IMW0241E Access denied - surrogateuser setup error.
   ```

   If these messages are generated for the failing client request, you must turn on program control for the Java DLLS residing in the HFS. For instructions, see APAR PQ18310 for details. Here is a summary of the APAR instructions:

   a. Give a superuser ID Read access to the BPX.FILEATTR.PROGCTL Facility to enable this superuser ID to update the HFS file attributes via the `extattr` command:

      ```
      RDEFINE FACILITY BPX.FILEATTR.PROGCTL UACC(NONE)
      PERMIT BPX.FILEATTR.PROGCTL CLASS(FACILITY) ID(superuserid)
            ACCESS(READ)
      SETROPTS RACLIST(FACILITY) REFRESH
      ```

   b. Go into the OMVS shell using the superuser ID that you just permitted to the BPX.FILEATTR.PROGCTL class.

      ```
      cd <JAVA_HOME>/lib/mvs/native_threads
      extattr +p *.*
      ls -E
      ```

      where <JAVA_HOME> is the path or the root directory for JDK.

6. When running Lotus Domino Go Webserver Release 5.0 with WebAS or SE support, and when the Webserver is configured with a user ID of %%CLIENT%% or a surrogate ID and the Keyfile directive is being used, clients may receive an error 500 message. Review the Domino Go Webserver trace log to determine if the following messages were issued for the particular client request:

   ```
   IMW0240E Access denied - unauthorized program loaded message.
   ```

   If these messages are generated for the failing client request, you must turn on program control for the C++ load library. For instructions, see APAR PQ18310 for details. Here is a summary of the APAR instructions:

   Issue the following RACF commands to turn on program control for the C++ load library:

   ```
   RALTER PROGRAM * ADDMEM('CBC.SCLBDLL'//NOPADCHK) UACC(READ)
   SETROPTS WHEN(PROGRAM) REFRESH
   ```

More hints and tips on how to configure WebAS can be found in APAR II11345 and at the following URL:

```
http://www.ibm.com/s390/nc/servlett.html
```

4.2.5 Running the Sample Servlet Code Shipped with DGW 5.0

There are a number of sample servlets (shown in Table 4) that are shipped with WebAS and are automatically loaded at WebServer startup. Try to execute them to verify that everything is configured correctly.

Note: All servlets or servlet packages migrated to your new DGW 5.0 installation should reside in the /usr/lpp/ServletExpress/servlets/ directory. Failure to use the default servlets directory will eliminate the dynamic reload capability of your migrated servlets/servlet packages.

Table 4. Sample Servlets Shipped with DGW 5.0

Servlet	**URL to open**	**Expected result**
Hello	http://*your.server.name*/servlet/HelloWorldServlet	Displays the string, “Hello World.”
Simple	http://*your.server.name*/servlet/SimpleServlet	Displays a heading and the text message, “This is output from SimpleServlet”
Snoop	http://*your.server.name*/servlet/SnoopServlet	Echos back information about the HTTP request sent by the client

4.2.6 Using WebAS Manager

Having the correct level of the JDK on your workstation is just as important as having the correct JDK level running on the OS/390 server. WebAS Manager requires that you have JDK 1.1 (or later) installed on your workstation.

Netscape Communicator 4.03 (or later) with the JDK 1.1 patch is required. To check your Netscape Java level on a workstation running Windows 95 or Windows NT, open the Java console from the **Window** option on the Netscape browser menubar. If needed, the JDK is available for downloading from:

```
http://help.netscape.com/filelib.html
```

Follow the instructions on that page to download and install the update.

Microsoft Internet Explorer 4 and Sun HotJava 1.1 can also be used to access WebAS Manager.

To start the WebAS Manager applet, enter the following URL at your Web browser:

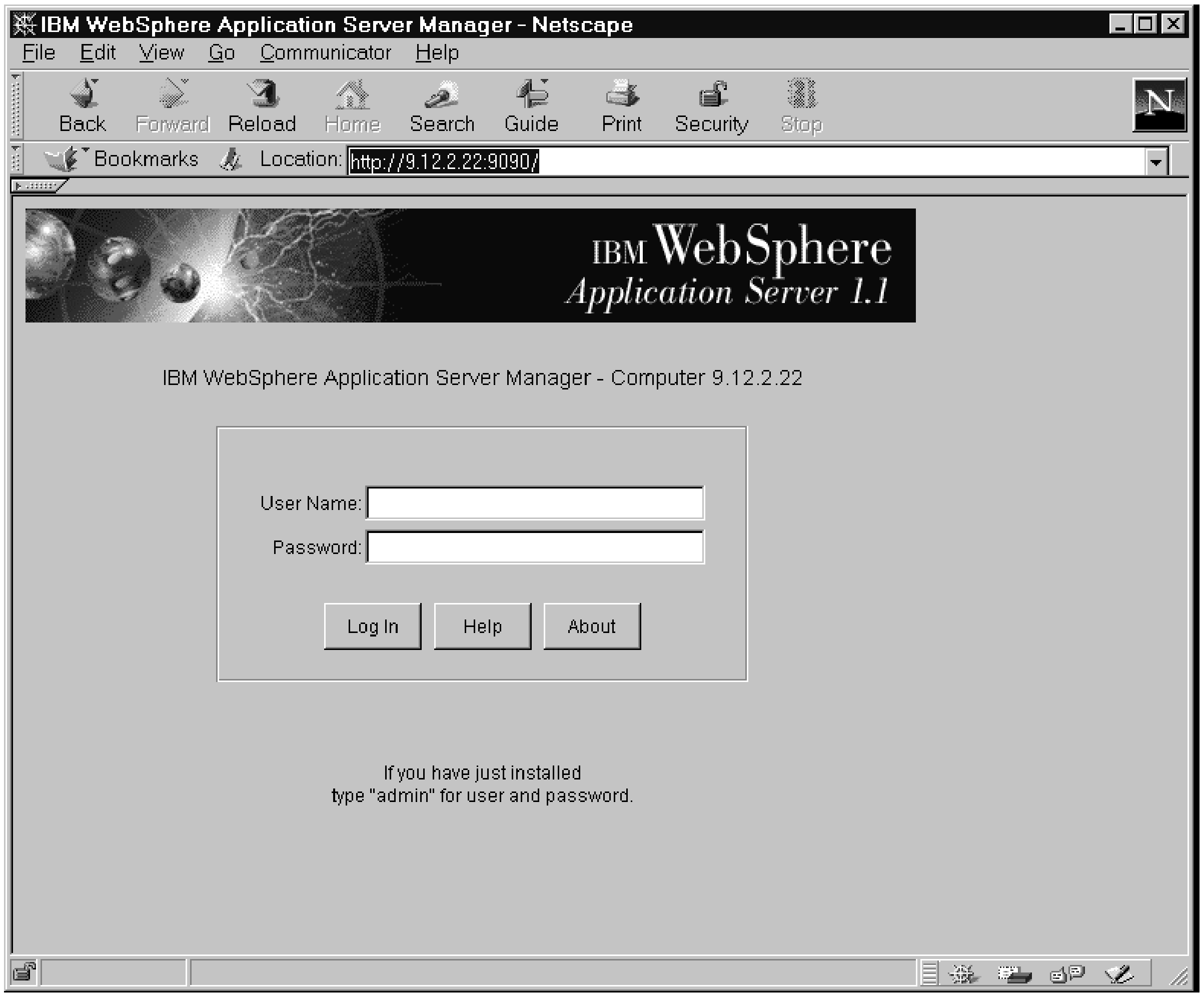
```
    http://your.web.server:9090
```

The WebAS Manager applet will start to load and you will be prompted to enter a valid user ID and password as shown in Figure 29 on page 50. The initial administrator user ID and password is admin/admin. You need to log in with this combination the first time. Once you are logged in, you can change it.

Note: When you change the admin password, it is stored in:

```
/usr/lpp/ServletExpress/realms/data/adminRealm/keyfile
```

(The password is encrypted in this file.)


IBM WebSphere Application Server Manager - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://9.12.2.22:9090/
IBM WebSphere Application Server 1.1
Log In Help About

IBM WebSphere Application Server Manager - Computer 9.12.2.22

User Name:
Password:

If you have just installed
type "admin" for user and password.

Figure 29. ServletManager Login Screen

When you log in with the admin user ID, the applet will display a window that shows you that WebAS is running. If you click on **Manage**, the WebAs Manager configuration interface will pop up and you can start to configure and control your servlets as shown in Figure 30 on page 51.

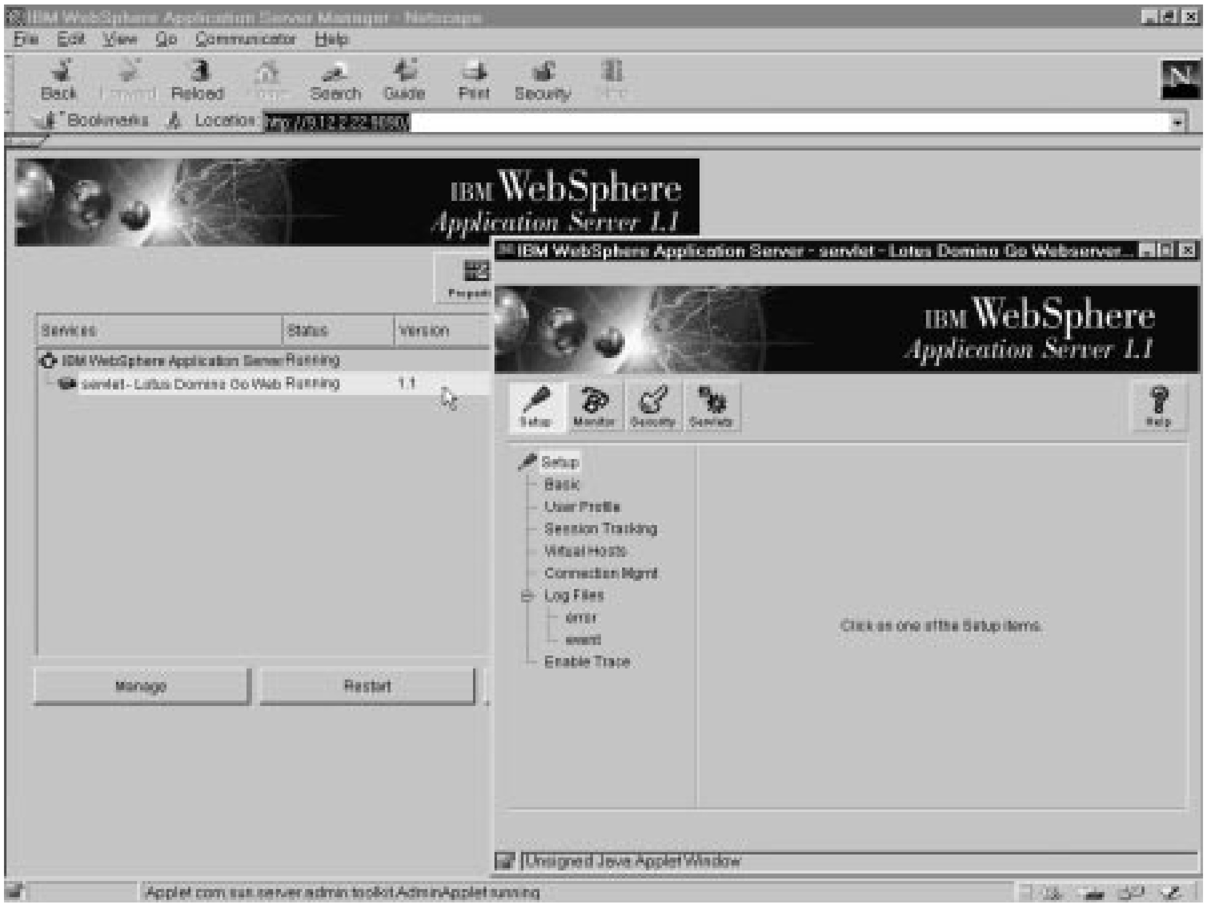

IBM WebSphere Application Server Manager - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location
IBM WebSphere Application Server 1.1
Services Status Version
IBM WebSphere Application Server Running
servlet - Lotus Domino Go Web Running 1.1
Manage Restart
IBM WebSphere Application Server - servlet - Lotus Domino Go Webserver...
Setup Monitor Security Servlets Help
Setup
Basic
User Profile
Session Tracking
Virtual Hosts
Connection Mgmt
Log Files
error
event
Enable Trace
Click on one of the Setup items.
Unsigned Java Applet Window
Applet com.sun.server.admin.toolkit.AdminApplet running

Figure 30. ServletManager Panels

See Chapter 5, “Configuring and managing Java Servlets” in *Lotus Domino Go Webserver: Webmaster's Guide Release 5.0 for OS/390*, SC31-8691. for a complete description of how to update WebAs Manager properties, as well as how to configure, load and manage servlets on your system.

Chapter 5. Configuring Java Support on OS/390

In this chapter we explain how you need to setup your environment on OS/390 in order to use Java.

In 5.1, “JDK Installation and Setup” we outline the installation of the JDK on OS/390. The JDK is a prerequisite for all products on OS/390 depending on Java.

5.2, “VisualAge for Java, Enterprise Edition for OS/390” on page 58 gives you all the details for installing the OS/390 components of VisualAge for Java Enterprise Edition for OS/390.

5.1 JDK Installation and Setup

At the time of writing, the latest generally available version of the Java Development Toolkit for OS/390 is JDK 1.1.6 and is downloadable from the following URL:

```
http://www.ibm.com/s390/java
```

In addition, this URL outlines the latest prereqs and instructions, and you will be able to download the lastest JDK available for OS/390. You can also order the latest version on tape, which includes a program directory containing the prereqs and the instructions.

Refer to URL:

```
http://www.ibm.com/s390/java/javainst.html#prer
```

for the latest information regarding prereqs for Java on OS/390 and to URL:

```
http://www.ibm.com/s390/java/javainst.html
```

for information regarding installation of Java for OS/390.

5.1.1 Directory Structure

You should decide from the beginning about the directory structure in which the JDK and other Java-related products will be installed. A possible structure is to create the `/usr/lpp/java/` directory. Then, in this directory, unpack the downloaded JDK install file as explained in 5.1.2, “Installation Method: Tar and Tarball Files.” The unpacking will create a subdirectory named J1.1 in the same `/usr/lpp/java/` directory. You may also install JDBC and other Java-related packages in the same directory structure.

5.1.2 Installation Method: Tar and Tarball Files

There are two ways of installing the JDK:

1. Download a Tar file and manually install it.
2. Download a Tarball file and install it with SMP/E.

If you choose the first solution, you will have to download a Tar file (a file made with the OS/390 UNIX System Services `tar` utility) and install it. You can either enter manual UNIX commands to untar the file, or run a utility called `ajvinst.exec`.

This is a REXX installation script that is downloadable from the same place as the Tar or Tarball files.

The second solution uses SMP/E. The Tarball file contains the SMP/E jobs and the program directory. Run the `ajvinst.exec` installation script to uncompress and install the Tarball.

5.1.3 Downloading the JDK

Go to URL for downloading the JDK onto your workstation:

```
http://www.ibm.com/s390/java/register.html
```

Then, after registration, follow the directions to transfer the JDK compressed file to your workstation.
After having stored the JDK compressed file on your workstation, transfer it to your OS/390 server using FTP. Initiate the FTP session from the workstation, using the FTP server running in the OS/390 machine.

5.1.4 Verifying the Installation

Your path should contain the binary directory where `java` is located. Type:

```
export PATH=/usr/lpp/java/J1.1/bin:$PATH
```

Now, verify that Java is correctly installed by typing:

```
java -fullversion
```

Java should reply with the correct version and build date of the JDK.

5.1.5 Using More than One Release of JDK

In order to use multiple releases of JDK under OS/390, the installation can simply install different releases in separate install libraries. In the following example, we installed JDK 1.1.4 in /usr/lpp/java14/J1.1 and we installed JDK 1.1.6 in /usr/lpp/java16/J1.1.

To point to the different desired releases, we setup the following in /HOME/.profile:

```
# ================================================================
# JAVA, JDBC, and Servlet Environment setup
# ================================================================
echo " "
echo " ***  JAVA, JDBC, and Servlet Environment is being Setup ***"
echo " "
export JAVA_HOME=/usr/lpp/java14/J1.1
  echo "JAVA_HOME path is set to ==> " $JAVA_HOME
export PATH=$JAVA_HOME/bin:/u/odonnel/SQLJ:$PATH
  echo "PATH is set to ==> " $PATH
export CLASSPATH=/usr/lpp/db2/db2510/classes/db2jdbcclasses.zip: 1
1 $JAVA_HOME/lib/classes.zip:
1 :/usr/lpp/ServletExpress/lib/jsdk.jar:
1 :/usr/lpp/SQLJ/classes.zip:
1 :.:
  echo "CLASSPATH path is set to ==> " $CLASSPATH
export LIBPATH=/usr:/usr/lib:/usr/lpp/db2/db2510/lib:
1 /usr/lpp/SQLJ/:
  echo "LIBPATH path is set to   ==> " $LIBPATH
export LD_LIBRARY_PATH=/usr/lpp/db2/db2510/lib:/usr/lpp/SQLJ:
  echo "LD_LIBRARY_PATH path is  ==> " $LD_LIBRARY_PATH
echo " "
echo " ***  JAVA Environment is now set  ***"
echo " "
export DB2SQLPLANNAME=SQLJ
export DB2SQLSSID=DB51
export DB2SQLJPLANNAME=SQLJ
export DB2SQLJSSID=DB51
```

In the example, you will need to update JAVA_HOME to the desired JDK home directory.

Notes:

> **1** These lines have been split for redbook printing purposes; however, in the real jvm.properties file they must be typed on one single line.

5.1.6 JDK Support for Websphere Application Server

In order to have Java available to Websphere Application Server, you must follow these steps:

1. Verify that the program control is turned on for Java DLLs. To verify this, go into the OMVS shell and issue the following command:

   ```
   cd <JAVA_HOME>/lib/mvs/native_threads
   ls -E
   ```

   In our example, we are using JDK 1.1.6. Note that the “p” bit is set to represent program control.

```
-r-xr-xr-x  -ps  1 BPXROOT  TSO       50160 Oct  8 23:04 jni_convert.o
-r-xr-xr-x  -ps  1 BPXROOT  TSO      139264 Oct  8 23:04 libagent.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO       10000 Oct  8 23:04 libagent.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO      430080 Oct  8 23:04 libagent_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO     6950912 Oct  8 23:04 libawt.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO       56080 Oct  8 23:04 libawt.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO     11837440 Oct  8 23:04 libawt_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO     11837440 Oct  8 23:04 libawt_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO     1982464 Oct  8 23:04 libjava.a
-r-xr-xr-x  -ps  1 BPXROOT  TSO      100080 Oct  8 23:04 libjava.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO     7794688 Oct  8 23:04 libjava_g.a
-r-xr-xr-x  -ps  1 BPXROOT  TSO      106240 Oct  8 23:04 libjava_g.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO     1089536 Oct  8 23:04 libjitc.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO     7008256 Oct  8 23:04 libjitc_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO      245760 Oct  8 23:04 libjpeg.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO        5680 Oct  8 23:04 libjpeg.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO     1015808 Oct  8 23:04 libjpeg_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO      380510 Oct  8 23:04 libm.a
-r-xr-xr-x  -ps  1 BPXROOT  TSO      893870 Oct  8 23:04 libm_g.a
-r-xr-xr-x  -ps  1 BPXROOT  TSO      217088 Oct  8 23:04 libmath.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO       11440 Oct  8 23:04 libmath.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO      507904 Oct  8 23:04 libmath_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO       40960 Oct  8 23:04 libmmedia.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO         640 Oct  8 23:04 libmmedia.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO      143360 Oct  8 23:04 libmmedia_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO      139264 Oct  8 23:04 libnet.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO        8320 Oct  8 23:04 libnet.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO      393216 Oct  8 23:04 libnet_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO       40960 Oct  8 23:04 libsysresource.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO         880 Oct  8 23:04 libsysresource.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO      147456 Oct  8 23:04 libsysresource_g.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO      196608 Oct  8 23:04 libzip.so
-r-xr-xr-x  -ps  1 BPXROOT  TSO        7840 Oct  8 23:04 libzip.x
-r-xr-xr-x  -ps  1 BPXROOT  TSO      557056 Oct  8 23:04 libzip_g.so
```

If program control is not turned on, you can use the following procedure to turn it on:

a. Give a superuser ID Read access to the BPX.FILEATTR.PROGCTL Facility to enable this superuser ID the ability to update the HFS file attributes via the `extattr` command:

```
RDEFINE FACILITY BPX.FILEATTR.PROGCTL UACC(NONE)
PERMIT BPX.FILEATTR.PROGCTL CLASS(FACILITY) ID(superuserid)
       ACCESS(READ)
SETROPTS RACLIST(FACILITY) REFRESH
```

b. Go into the OMVS shell using superuser ID that you just permitted to the BPX.FILEATTR.PROGCTL class:

```
cd <JAVA_HOME>/lib/mvs/native_threads
extattr +p *.*
ls -E
```

where <JAVA_HOME> is the path or the root directory for JDK.

Verify the classpath, libpath, and path statement in the ServletExpress jvm.properties file located in <SEroot>/ServletExpress/properties/server /ServletExpress/servletservice.

```
ncf.jvm.classpath=/web/WebSphere/AppServer/lib/ibmwebas.jar:
    1 /web/WebSphere/AppServer/lib/jst.jar:
    1 /web/WebSphere/AppServer/lib/jsdk.jar:
    1 /web/WebSphere/AppServer/lib/x509v1.jar:
    1 /web/WebSphere/AppServer/lib:
    1 /web/WebSphere/AppServer/web/admin/classes/seadmin.jar:
    1 /web/WebSphere/AppServer/web/classes:
    1 /usr/lpp/java/J1.1/lib/classes.zip:
    1 /usr/lpp/db2/db2510/classes/db2jdbcclasses.zip:
    1 /usr/lpp/db2/db2510/classes/db2sqljclasses.zip:
    1 /usr/lpp/db2/db2510/classes/db2sqljruntime.zip
ncf.jvm.libpath=/usr/lpp/java/J1.1/lib:
    1 /usr/lpp/java/J1.1/lib/mvs/native_threads:
    1 /web/WebSphere/AppServer/lib:
    1 /usr/lib
    1 /usr/lpp/internet/bin:
    1 /usr/lpp/db2/db2510/lib
ncf.jvm.path=/usr/lpp/java/J1.1/bin
```

Notes:

1 These lines have been split for redbook printing purposes; however, in the real jvm.properties file they must be typed on one single line.

5.1.7 Remote Abstract Windowing Toolkit (RAWT)

OS/390 does not support a native GUI. A Java application using AWT classes can only run on OS/390 if the GUI is “exported” to a client with a native GUI capability.

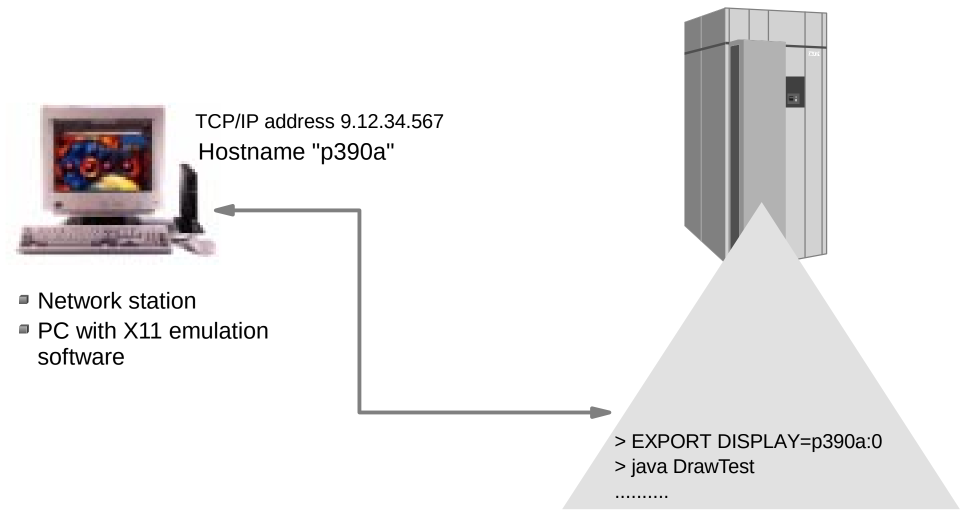
Typically, before you would run the Java application from the OS/390 command line, you would start your X11 server on the client and issue this command from the OS/390 command line, where the hostname is the hostname of the client displaying the GUI:

```
export DISPLAY=<hostname>:0
```

The result would then be an X-windows-like GUI.

Until recently, the X11 protocol was the only solution to make this happen. However, this protocol is very time-consuming to use as every send and receive is a new connection. Also, the X11 protocol requires an X11 server to be installed on the client side to display the GUI. Under Windows/95 and Windows NT, the X11 server is not a base function, so a product has to be purchased and installed on each client machine to run the X11 server.

Figure 31 shows the required infrastructure for the X11 protocol.



```
> EXPORT DISPLAY=p390a:0
> java DrawTest
..........
```

Figure 31. Infrastructure for Running AWT Applications Using the X11 Protocol

Remote AWT (RAWT) is a new solution for displaying the graphical user interface of a Java application running on OS/390, on a client supporting a native GUI. The big advantage of RAWT is that there is no longer a requirement for an X-server on the client side.

RAWT consists of two parts: one on the host where the application runs, and one on the client where the GUI will be displayed. Both components are Java-enabled, meaning that they only require a Java Runtime Environment (JRE) to be installed. Once the two components have been installed, they can communicate with each other over TCP/IP. So RAWT is really “light,” requiring only TCP/IP and Java. Figure 32 on page 58 shows the infrastructure when RAWT is used.

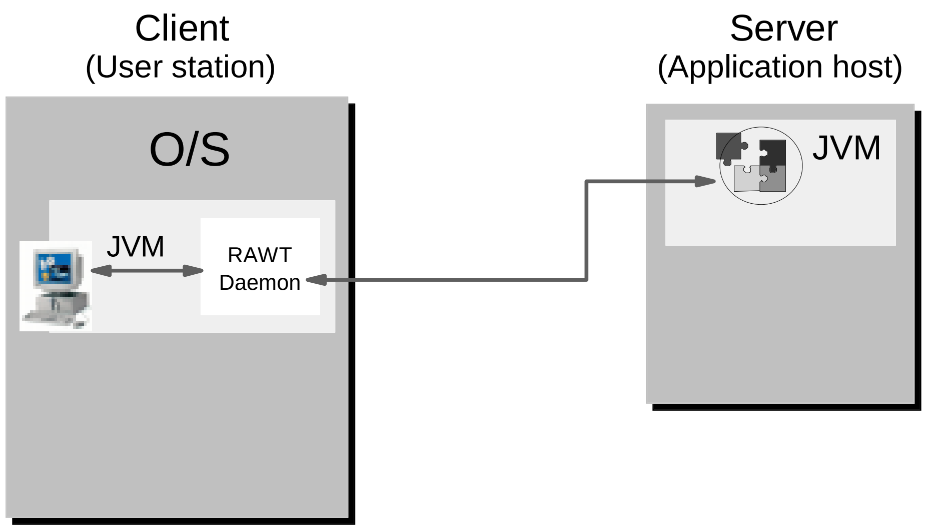


Figure 32. Infrastructure for Running AWT Applications Using RAWT

The RAWT component on OS/390 comes with the JDK 1.1.6 with the following PTF installed:

OS/390 Version Release 1,2 and 3 UW57717

OS/390 Version Release 4 and 5 UW57730

OS/390 Version Release 6 and 7 UW57731

Refer to URL: `http://www.ibm.com/s390/java/rawt.html` for details.

5.2 VisualAge for Java, Enterprise Edition for OS/390

In this section we describe the configuration of the OS/390 components of VisualAge for Java, Enterprise Edition for OS/390. The workstation-related matters are discussed in

Attention

In this redbook we use the term HPJ for the native code compiler for Java on OS/390 because HPJ was IBM's internal code name.

However, the official product name is VisualAge for Java, Enterprise Edition for OS/390.

Chapter 6, “Configuring VisualAge for Java on the Workstation” on page 71.

5.2.1 Introduction

The VisualAge for Java Enterprise Edition for OS/390 is comprised of the following:

1. VisualAge for Java Enterprise Edition IDE (Integrated Development Environment) running on a Windows NT workstation.
2. The Enterprise Toolkit for OS/390 (ET/390), which is a separately installed set of plug-ins for VisualAge for Java Enterprise Edition, running on the Windows NT workstation, which provide the facilities to:

- Establish a host session with and log onto OS/390
- Export packages to OS/390 to:
  - Run in the OS/390 JVM
  - Compile using the HPJ/390 compiler and execute on OS/390
- Remotely debug Java applications running on the OS/390 JVM or running as bound executables from the VisualAge for Java Graphical User Interface
- Remotely analyze the performance of Java applications running on OS/390
- Run the jport utility on the VisualAge for Java IDE (or from the OS/390 UNIX shell) to evaluate the portability of a Java package to OS/390

3. The Visual Age for Java OS/390 facilities:
   - The Java Run Time Library

     This feature is required by all VisualAge for Java, Enterprise Edition customers and is used to execute fully bound Java programs. The run-time library implements the Java APIs as fully compiled native objects and provides for memory management (garbage collection and other system routines). OS/390 UNIX System Services and CICS Transaction Server 1.3 are supported.
   - The HPJ/390 compiler

     The HPJ/390 compiler is an optimizing native code translator/binder that statically translates Java bytecode directly into native (object) code in the same manner as traditional compilers for C/C++, COBOL and FORTRAN. Traditional resource intensive optimization techniques (such as dataflow analysis and interprocedure optimization) are used to improve the performance of the generated code. The high performance compiler/binder fully binds the object code into an executable or dynamic link library (JLL) that can be run in the UNIX System Services Shell or under the CICS Transaction Server for OS/390.

5.2.2 The Orderable Package

The program number for VisualAge for Java, Enterprise Edition for OS/390 is 5655-JAV.

The VisualAge for Java, Enterprise Edition for OS/390 is packaged and orderable as two separate features:

1. IBM VisualAge for Java, Enterprise Edition for OS/390, Version 2.x - Run Time Feature - Required and unpriced.

   Feature number 5976 for 3480 cartridge tape; feature number 6082 for 4mm tape; and feature number 5975 for 9 track 6250 bpi tape.
2. IBM VisualAge for Java, Enterprise Edition for OS/390, Version 2.x - Compiler Feature - Optional and priced.

   Feature number 5989 for 3480 cartridge tape; feature number 6092 for 4mm tape; and feature number 5988 for 9 track 6250 bpi tape.

The Run Time Feature is required by all customers and is needed to execute fully bound Java Programs. Included with the Run-Time feature is the shrink-wrapped CD package for VisualAge for Java, Enterprise Edition containing two CDs. The two CDs contain VisualAge for Java Enterprise Edition for the Windows NT workstation and the Enterprise toolkits for all supported host environments including OS/390. The ET/390 plug-ins on the workstation will require a separate install after the VisualAge for Java, Enterprise Edition for Windows NT Workstation install.

The HPJ Run Time feature on OS/390 and optionally the HPJ Compiler feature on OS/390 are installed via an SMP/E process from the tapes described.

The HPJ Compiler feature is optional and may be used in conjunction with the run-time feature to develop fully compiled and bound Java programs.

5.2.3 Software Requirements

The following OS/390 programs are required to install, service and run VisualAge for Java, Enterprise Edition for OS/390:

1. OS/390 V2R4 (5647-A01), or later with:
   - Language Environment
   - UNIX System Services (OpenEdition) enabled
   - DFSMS enabled
   - SMP/E Rel 1.8 (Program Number 5668-949) for installation
2. Also required for the following functions are the specified products that are listed, along with the appropriate service levels, in Figure 11 on page 10 of the *Program Directory for VisualAge for Java, Enterprise Edition for OS/390*, Program Number 5655-JAV, Document Number GI10-4949 (product documentation).
   - C/C++ with Debug Tool on OS/390

     Required to use the VisualAge for Java, Enterprise Edition for OS/390 Debugging tool. This is provided in OS/390 V2R4 and later as an optional feature. It is provided as a dataset EQA.V1R2M0.SEQAMOD.
   - OS/390 Version 2 Release 4 (or later) Host Performance Analyzer.

     Required to use the VisualAge for Java, Enterprise Edition for OS/390 remote performance analysis tools. This is provided in OS/390 V2 R4 and later as part of the optional feature - C/C++ with Debug Tool. It is provided as a dataset CBC.SCTVMOD.
   - OS/390 Version 2 Release 4 C/C++ Compiler without DEBUG Tool - only if JNI support is required.

     Required for JNI support using the OS/390 JVM or the High Performance Compiler for Java on OS/390.

     Note: If the C/C++ with Debug Tool optional feature has been installed for Debug and/or Performance Analysis VisualAge for Java, Enterprise Edition for OS/390 support, that is sufficient for JNI support,

     Note: All of these C/C++ Debug products are supplied as optionally installable features with the OS/390 product. To make them part of the system, they must be enabled by an appropriate IFAPRDxx entry in the IFAPRDxx parmlib member. This is described in *OS/390 V2R6.0 Planning for Installation*, GC28-1726.

     The VisualAge for Java, Enterprise Edition for OS/390 Version2 - Product Number 5655-JAV, when ordered with OS/390 V2R7.0, is provided with an IBM-supplied IFAPRD00 member that contains the required `PRODUCT` statements to enable the VAJAVA/390-DEBUG feature. This feature provides the required Debug/Performance Analyzer functions.

- NFS and FTP Links

  An NFS server and an FTP server are required on OS/390, and an NFS client and an FTP client are required on the NT Workstation for workstation-to-OS/390 remote file access and file transfer operations.

  The FTP client is part of the base function of Windows NT and requires no special configuration for VisualAge for Java, Enterprise Edition for OS/390 to be able to use it.

  The FTP Server on OS/390 is part of the base TCP/IP facilities. These facilities are part of the OS/390 base support from OS/390 V2 R4 and later (in OS/390 V2 R7, the TCP/IP facilities will be known as eNetwork Communications Server facilities).

  See 5.2.6.3, “Setting Up FTP” on page 69 for FTP setup requirements on OS/390 that are related to VisualAge for Java, Enterprise Edition for OS/390

  The NFS Client on the IDE Workstation is part of the base functions of Windows NT. See 5.2.6.1, “Setting Up the NFS Client on Windows NT” on page 67 for setting up the NFS Client on Windows NT.

  The NFS Server on OS/390 is part of the base elements of OS/390 for OS/390 V2R4 and later releases. The NFS Server was known as DFSMS/MVS NFS until OS/390 V2 R6, at which time it became exclusive to OS/390, had function added to it, and is now known as NFS. See 5.2.6.2, “NFS Server Setup” on page 68 for configuring the NFS Server on OS/390.

5.2.4 Installation of VisualAge for Java, Enterprise Edition for OS/390

The Installation of the HPJ Run Time Library and the HPJ Compiler, Program Number 5655-JAV, is provided through two FMIDs:

FMID H0A5201 VisualAge for Java Compiler, priced

FMID H0A5202 VisualAge for Java Run Time Library, no charge

The installation of these two FMIDs is detailed in the *Program Directory for VisualAge for Java, Enterprise Edition for OS/390*, Program Number 5655-JAV, Document Number GI10-4949 (product documentation).

5.2.4.1 Installation of HPJ on OS/390

The installation of the HPJ Run Time and the HPJ Compiler are described in detail in the *Program Directory for VisualAge for Java, Enterprise Edition for OS/390*, Program Number 5655-JAV, Document Number GI10-4949 (product documentation).

As part of the SMP/E installation for both FMIDs, sample jobs are provided that assist in the installation of VisualAge for Java. These jobs and the changes that must be made to them for the specific installation configuration, as well as the steps that must be followed to perform the jobs, are described in the Program Directory product documentation.

5.2.4.2 Installation and Set Up of Directories on UNIX System Services

The HFS directories are set up as a result of the HPOISMKD and the HPJISMKD JOBS for the VisualAge for Java, Enterprise Edition for OS/390 Run Time Library and the VisualAge for Java, Enterprise Edition for OS/390 HPJ Compiler respectively.

When the installation of the Run Time Library and the HPJ Compiler is complete, it is necessary to set the environment variables for `IBMHPJ_HOME`, `CLASSPATH`, `LIBPATH AND STEPLIB` to the appropriate values.

`STEPLIB` should include `HPJ.SHPOMOD`, `HPJ.SHPJMOD and CEE.SCEERUN`. When both the VisualAge for Java Runtime Library and the VisualAge for Java Compiler have been installed, the contents of the file with default name `/usr/lpp/hpj/bin/profile.hpj` as shown in Figure 33 should be appended to `/etc/profile` for use by everyone, or copied into `$HOME.profile` for individual use and updated to reflect the values used in your environment as shown in Figure 34 on page 63.

Note: For better performance of the VisualAge for Java Run Time Library and the HPJ Compiler, your system programmer could copy the entire `HPJ.SHPOMOD` and `HPJ.SHPJMOD` data sets into the LPA. If this is done, `HPJ.SHPOMOD` and `HPJ.SHPJMOD` should be removed from the environment variable `STEPLIB`. Performance can also be improved by copying the EQA.V1R2M0.SEQAMOD (Debugger) and the CBC.SCTVMOD (Performance Analyzer) data sets into the LPA.

```
#
#   1) Change HPJ and CEE to the appropriate high-level qualifier for
#      VisualAge for Java, Enterprise Edition for OS/390 and Language
#      Environment.
#
#   2) Change IBMHPJ_HOME to your install directory.
#
#-------------------------------------------------------------------
#
export IBMHPJ_HOME="/usr/lpp/hpj"
export IBMHPJ_RTL="CEE.SCEELKED:CEE.SCEELKEX:CEE.SCEEOBJ:CEE.SCEECPP"
export CLASSPATH=$CLASSPATH:$IBMHPJ_HOME/lib
export PATH=$PATH:$IBMHPJ_HOME/bin
export LIBPATH=$LIBPATH:$IBMHPJ_HOME/lib
export STEPLIB=$STEPLIB:HPJ.SHPJMOD:HPJ.SHPOMOD:CEE.SCEERUN
#
echo '*-------------------------------------------------------------'
echo '         profile.java was executed                           '
echo '*-------------------------------------------------------------'
```

Figure 33. Example of profile.hpj File

The system programmer should append the updated profile.hpj file to `/etc/profile` for general use or to the `/u/"username"/.profile` for individual user use.

```
export IBMHPJ_HOME=/usr/lpp/hpj  1
export IBMHPJ_RTL="CEE.SCEELKED:CEE.SCEELKEX:CEE.SCEEOBJ:CEE.SCEECPP"  2
export CLASSPATH=$IBMHPJ_HOME/lib:$CLASSPATH  3
export PATH=$IBMHPJ_HOME/bin:$PATH  4
export LIBPATH=$IBMHPJ_HOME/lib:$LIBPATH  5
export STEPLIB=$STEPLIB:HPJ.SHPJMOD:HPJ.SHPOMOD:CEE.SCEERUN  6
```

Figure 34. Example of HPJ Variables in /etc/.profile

Notes:

1 Set the home variable for the HPJ root.

2 Required LE support for HPJ Compiler; CEE was used as the high level qualifier.

3 Append the CLASSPATH with the HPJ classes.

4 Append the PATH with the HPJ executables.

5 Append the LIBPATH with the HPJ executables.

6 HPJ was used as the high level qualifier.

5.2.4.3 Customization of the javaInstall.data File

This file is used for communication between the IDE on the workstation and the OS/390 Java Execution environment. It is used for:

- Providing default values for the fields in establishing a host session
- Providing fields for use in the tmp.cmd files that are sent to the host

The javaInstall.data file is in the `/usr/lpp/hpj/` directory.

```
@@HPJHostName: wtsc58oe.itso.ibm.com  1
@@HPJHome: /usr/lpp/hpj         2
@@HPJBinderExecutablesPDSE: HPJ.SHPJMOD
@@HPJBinderMessagesPDSE: HPJ.SHPJMOD
@@HPJLERuntimeBind: CEE.SCEELKED:CEE.SCEELKEX:CEE.SCEEOBJ:CEE.SCEECPP
@@HPJLERuntimeRun: CEE.SCEERUN
@@HPJRuntime: HPJ.SHPJMOD
@@HPJDebugger: EQA.V1R2M0.SEQAMOD  3
@@HPJProfiler: CBC.SCTVMOD       4
@@HPJJavaHome: /usr/lpp/java16/J1.1  5
@@HPJPICLHome: /usr/lpp/hpj/jdebug/engine/jdebug.jar  6
@@HPJCICSRegion: IYKC54:IXJD78
@@HPJCICSEXCI: CICSTS13.CICS.SDFHEXCI
```

Figure 35. Example of Settings in .javaInstall.data File

Notes:

1 Set the Host name to the IP name of the OS/390 host.

2 Set the HPJHome to the directory that contains the HPJ directories and files.

3 This data set must be included in the OS/390 module's search path.

4 This data set must be included in the OS/390 module's search path.

5 HPJJavaHome is the location of the jdk that is installed on OS/390.

6 HPJPICLHome is the home directory that contains the jdk source files for the OS/390 host debugger

Because there was no @@HPJFTPPort specification, the default port of 021 is assumed.

Note: If the PDSEs or PDSs specified in the following are added to the LPA, an * (asterisk) must be added to the specification:

- @@HPJBinderExecutablesPDSE
- @@HPJLE RuntimeRun
- @@HPJRuntime
- @@HPJDebugger
- @@HPJProfiler

5.2.4.4 Installation of the Remote Debugger for OS/390

The C/C++ with Debug Tool is an optional feature for OS/390 that can be ordered with the original OS/390 order. After installation, it can be dynamically enabled. This feature can be dynamically enabled by copying IFAPRD00 to a new IFAPRDxx SYS1.PARMLIB member that you can edit to enable the VAJAVA/390-DEBUG feature by adding the following PRODUCT statement.

```
PRODUCT OWNER('IBM CORP')
        NAME('IBM VA JAVA/390')
        ID(5655-JAV)
        VERSION(*) RELEASE(*) MOD(*)
        FEATURENAME('VAJAVA/390-DEBUG')
        STATE(ENABLED)
```

This will provide the required C/C++ compiler feature for use with JNI and will provide the base DEBUG tool required for VisualAge for Java Enterprise Edition for OS/390 using the remote debug facilities from the IDE.

Note: Refer to *OS/390 V2R6.0 Planning for Installation*, GC28-1726 or later, for instructions on dynamically enabling features including alerting your IBM representative that you are starting to use the newly enabled features.

5.2.4.5 Setting Up to Use of the Remote Debugger with OS/390

Remote debugging can be set up in the following manner:

From the IDE

- Set up the ET/390 Properties - Debugging Options
- Choose **Debug Main** from the ET/390 actions.

This will start the remote debugger on the workstation and start the application on the OS/390 with the TEST runtime option.

If the java source files are on OS/390 the IVJ-DBG_PATH environment variable must be set. A good initial value for this environment variable is the value of CLASSPATH on OS/390. You should include the statement

```
export IVJ_DBG_PATH=$CLASSPATH
```

in the appropriate profile file.

5.2.4.6 Installation for Remote Debugging of Bytecode on OS/390

Remote debugging of bytecode on OS/390 is currently in a technical preview status. To perform rmote debugging of bytecode requires an auxillary debug engine on OS/390. It is also necessary to provide a Data Information File on OS/390 that contains the location of the auxillary debug engine on the host. This can be accomplished as follows:

1. The VisualAge for Java, Enterprise Edition, ET/390 Toolkit contains an Extra directory that contains the following:
   - A readme file on this installation
   - A \extra\jdebug.jar file (The auxiliary debug engine)
   - A \extra\jdebug file (The Data Information File)
2. FTP the "\extra\jdebug.jar" file from the install CD or the file you downloaded it into on the workstation to OS/390 HFS in binary mode to a HFS directory that is pointed to by your OS/390 PATH environment variable.
3. ftp the "\extra\jdbug" file from the workstation to the HFS directory that is pointed to by your OS/390 PATH environment variable.
4. Make the jdbug file executable by performing the following OE Shell command: `chmod +x jdbug`
5. On OS/390 make the following changes to your OE shell environment (for example, put in your .profile file)
   - `export CLASSPATH=<location of the jdebug.jar file>:$CLASSPATH`
   - `export JAVA_COMPILER=off`
6. Invoke the bytecode debugger engine in the OS/390 shell by typing `jdbug [-qport=<TCP/IP port>]`.
   This option sets the TCP/IP port where the host debugger expects a connection. The default port is 8000. The host debugger is now waiting for a connection on the specified TCP/IP port.

   Note: It is necessary to set the Debugger Port number in the OS/390 HFS directory `/etc/services/`.

5.2.4.7 Installation of the Performance Analyzer for OS/390

The data set CBC.SCTVMOD that contains members such as the Performance Analyzer module CTVPFILE, is also enabled when you enable the VAJAVA/390-DEBUG feature for IBM VisualAge for Java, Enterprise Edition.

The latest service must be applied to FMIDs H24P111 J24P112.

To run the Performance Analyzer on the host system, the CBC.SCTVMOD dataset must be included in the OS/390 module's search path. It can be added to the Link Pack Area by your system programmer, or you can use the `export` command in the OS/390 shell to add it to your STEPLIB before you run your program, as shown in the following example:

```
export STEPLIB=CBC.SCTVMOD:$STEPLIB
```

Note: The sample job HPJREG, provided as part of the SMP/E install, will place a tailored IFAPRDxx member into the SYS1.PARMLIB dataset. This will enable both the Debug Tool and the Performance Analyzer.

5.2.4.8 Setting Up to Use the Performance Analyzer

Have your system programmer add the SCTVMOD load module data set to the Link Pack Area. As an alternative, add the SCTVMOD load module data set to the STEPLIB of the program. This is done by setting the STEPLIB environment variable with the `export` OS/390 shell command, as shown in the following example:

```
export STEPLIB=CBC.sctvmod:$STEPLIB
```

Prior to creating a function trace, it is necessary to bind your program using the `-g=hook` option with the `hpj` command. This option enables the bytecode binder to generate hooks in the code at various points relative to a function call, or at a function exit or exit.

When a Java application is run on the OS/390 shell, the Performance Analyzer is started as a result of the PROFILE run time option being set on by one of the methods described in the following examples. To enable the function tracing of a program during its execution, you have to set the run time option PROFILE and its suboptions. The PROFILE options can be set:

- As installation defaults at install time.
- As an `hpj` command's `-lerunopts` option which is set when you build the Java executable. The PROFILE run time option is bound with the Java executable.
- As an export command at run time that sets _CEE_RUNOPTS.

The order of application of the PROFILE run time option is:

- The `export` command set at run time; if specified:
  - `export _CEE_RUNOPTS="profile(on,'task')"`
- The `hpj` command's `-lerunopts` option; if specified:
  - `-lerunopts:PROFILE(ON,REAL)`
- An installation default for the PROFILE run time option set at install time as part of the Language Environment setup; it must be specified :
  - `PROFILE(ON,REAL)`

5.2.5 Testing the HPJ/390 Environment

The VisualAge for Java Run Time Library may be tested by editing and submitting the sample job HPOJIVP to verify the correct installation of the Run Time Library.

Edit and submit a sample job HPJJIVP to verify the correct installation of the HPJ/390 Compiler.

Note: Refer to *OS/390 V2R6.0 Planning for Installation*, GC28-1726 or later (depending on the OS/390 Release that you are installing VisualAge for Java, Enterprise Edition for OS/390 on), and the service Web sites referenced in this redbook as well as to the *Program Directory for VisualAge for Java, Enterprise Edition for OS/390*, GI10-4949. to be sure that the proper service has been applied to all programs that are part of this install.

5.2.6 Related Network Installations on OS/390

OS/390 must be configured with a full TCP/IP stack with the following functions enabled:

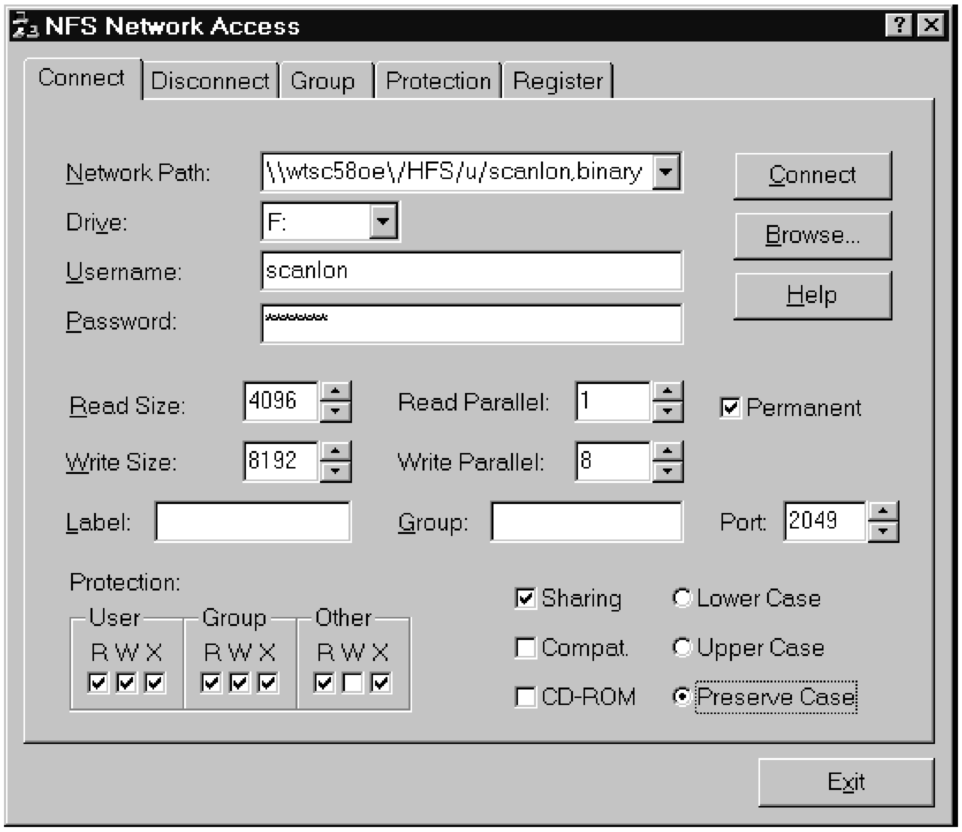
- NFS Server - This comes as part of the base elements of OS/390 and must be installed and enabled for OS/390 Release 4 and later.
- FTP Server - This comes as part of the TCP/IP stack and must be installed and enabled.

5.2.6.1 Setting Up the NFS Client on Windows NT

The Windows NT NFS Client can be set up using the base NT `nfs link` facility for both a binary link and a text link (required if the source file is to be sent to the OS/390 host for debugging). It is necessary to specify:

- The local drive letter
- The hostname
- The mount point on the host
- The username and the password that you are known by on the host
- The protection properties you want on the host
- `preserve case`
- The port number, which should be 2049 unless your system has a customized port address

Alternatively, if a third-party NFS facility such as Hummingbird's Maestro is available on your workstation, you can use a graphical interface such as that shown in Figure 36 on page 68 to set up your NFS connection.



Network Path: \\wtsc58oe\/HFS/u/scanlon.binary
Drive: F:
Username: scanlon
Password: *******
Read Size: 4096
Read Parallel: 1
[✓] Permanent
Write Size: 8192
Write Parallel: 8
Label:
Group:
Port: 2049
Protection:

User R	User W	User X	Group R	Group W	Group X	Other R	Other W	Other X
✓	✓	✓	✓	✓	✓	✓	☐	✓

[✓] Sharing
☐ Compat.
☐ CD-ROM
Lower Case / Upper Case / [✓ Preserve Case]

Figure 36. NFS Setup Using Hummingbird NFS Maestro - Binary Connection

5.2.6.2 NFS Server Setup

The NFS Server on OS/390 has to be started as part of the OS/390 IPL of the OS/390 system. The NFS Server can be started with SYS1.PROCLIB or using the OE shell.

The NFS server should have the `logout(n)` parameter in the Site Attributes in the `NFS.ATTRIB` data set set with the value of `n` equal to a value in seconds that is consistent with a typical work session. The default NFS session time is quite small (30 minutes). See *OS/390 V2R6.0 NFS Customization and Operation*, SC26-7253.

Note: It is imperative that the `inetd.conf` properties be set up such that the `rexecd` daemon is started. This is required for remote VisualAge for Java, Enterprise Edition for OS/390 to OS/390 operations to work. Figure 37 on page 69 shows an example of configuring this facility.

```
# /etc/inetd.conf
#
#          Internet server configuration database
#
# Services can be added and deleted by deleting or inserting a
# comment character (ie. #) at the beginning of a line
#
#================================================================
# service | socket | protocol | wait/ | user | server  | server program
# name    | type   |          | nowait|      | program |   arguments
#================================================================
#
otelnet  stream tcp nowait OMVSKERN /usr/sbin/otelnetd otelnetd -l
shell    stream tcp nowait OMVSKERN /usr/sbin/orshd rshd -LV
login    stream tcp nowait OMVSKERN /usr/sbin/rlogind rlogind -m
exec     stream tcp nowait OMVSKERN /usr/sbin/orexecd rexecd -LV
#finger   stream tcp nowait OMVSKERN /usr/sbin/fingerd fingerd
```

Figure 37. Example of inetd.conf File

5.2.6.3 Setting Up FTP

The FTP Server must be started as part of the IPL of the OS/390 system. The FTP server can be started either with SYS1.PROCLIB, or by using the OE shell.

To perform ET/390 Run Executable, Debug Executable, Run Main and Debug Main actions from the VisualAge for Java IDE, the OS/390 Communications Server (TCP/IP) `Sbdataconn` parameter in the /etc/ftp.data data set must be correctly configured on the host to define the conversion between EBCDIC and ASCII code pages. See *OS/390 TCP/IP OpenEdition Implementation Guide*, SG24-2141, for more information on how to define this parameter.

Figure 38 shows the values we set in the /etc/ftp.data data set.

```
StartDir        HFS        ; Set initial FTP working directory in HFS
Primary         50         ; Primary allocation is 50 tracks
Secondary       20         ; Secondary allocation is 20 tracks
Directory       15         ; PDS allocated with 15 directory blocks
Lrecl           80         ; Logical Record Length of 80
BlockSize       28960      ; Block Size of 28960 (1/2 trk for 3390)
AutoRecall      true       ; migrated HSM files recalled automatically
AutoMount       false      ; non-mounted volumes mounted automatically
DirectoryMode   false      ; use all qualifiers (Datasetmode)
Volume          TARTS2     ; Volume serial number for allocation
SpaceType       TRACK      ; datasets allocated in tracks
Recfm           FB         ; Fixed Blocked record format
Filetype        SEQ        ; File Type = SEQ (default)
Ctrlconn        IBM-850    ; ASCII codepage
Sbdataconn      (IBM-1047,IBM-850) ; EBCDIC, ASCII codepage
Umask           022        ; Make new HFS files rw- r-- r-- 644
Inactive        300        ; 5 minutes
```

Figure 38. Example of /etc/ftp.data File

Chapter 6. Configuring VisualAge for Java on the Workstation

In this chapter we explain how to set up VisualAge for Java Enterprise Edition for OS/390 on the workstation in order to make use of the Enterprise Toolkit/390 (ET/390) facilities.

The configuration of the OS/390 components has been discussed in 5.2, “VisualAge for Java, Enterprise Edition for OS/390” on page 58.

Attention

In this redbook we use the term HPJ for the native code compiler for Java on OS/390 because HPJ was IBM's internal code name.

However, the official product name is VisualAge for Java, Enterprise Edition for OS/390.

6.1 VisualAge for Java Setup for OS/390 Operation

In 5.2, “VisualAge for Java, Enterprise Edition for OS/390” on page 58 we describe the setup required on the OS/390 to install the OS/390 facilities required for:

- Standalone VisualAge for Java operations on OS/390
- Remote execution and debugging on OS/390 under control of the VisualAge for Java, Enterprise Edition for OS/390 IDE on the workstation:
  - Code Java programs at the workstation and export your bytecode to run in a remote OS/390 Java Virtual Machine (JVM).
  - For improved performance on OS/390, use the ET/390 Toolkit to bind Java bytecode into optimized object code and run it under OS/390 UNIX System Services as an executable or a DLL in the OS/390 shell.
  - Write CICS Applications in Java and bind the Java bytecode into object code to run under CICS/ESA.

In this chapter we describe the setup required of the IDE to establish a remote session on OS/390 and to perform export, bind, execution and debug/performance analysis operations on OS/390.

In 6.1.1, “FTP Connections” and 6.1.2, “NFS Connections” on page 72 we describe the two connection methods you must implement to successfully use the ET/390 facilities.

6.1.1 FTP Connections

Prior to using the ET/390 facilities, the workstation user must insure that FTP is available as a client on the workstation, and as both a client and a server on OS/390.

FTP is used by VisualAge for Java Enterprise Edition for OS/390 for the following operations:

- To download the javaInstall.data file from OS/390 to the workstation, where the data in this file is used to build a command file that is stored locally in the IDE on the workstation for later communication to OS/390.
- To send to the OS/390 host a temporary command, built using the command file created, associated with a bind and/or execute operation initiated on the workstation.
- To send the output from compiler operations performed on OS/390 to the IDE workstation, where it is displayed in the IDE log.

OS/390 comes with a full TCP/IP stack which includes an FTP server. See *TCP/IP Open Edition: User's Guide*, GC31-8305, for information on using FTP in the OE environment.

The VisualAge for Java ET/390 facilities use the OS/390 FTP server transparently in performing the three functions just described.

Windows NT provides an FTP-enabled FTP client as part of TCP/IP. The VisualAge for Java IDE uses the FTP facilities of the client in a manner that is transparent to the developer on the workstation.

6.1.2 NFS Connections

VisualAge for Java, Enterprise Edition for OS/390 requires that NFS connections must be established between the NFS client on the workstation and the NFS server on OS/390.

The OS/390 NFS Server (NFSS) must be installed on the OS/390 system. DFSMS/MVS Network File System Version 1 Release 3 comes as part of the base with OS/390 V2 Release 4 and higher. The installation process is described in *OS/390 V2R6.0 NFS User's Guide*, SC26-7254.

The NFS client on the workstation can be set up by using the base Windows NT `nfs link` facility as described in 5.2.6.1, “Setting Up the NFS Client on Windows NT” on page 67. Or a graphical product such as Hummingbird's Maestro NFS client can be used for the set up as shown in the NFS configuration dialog box in Figure 36 on page 68

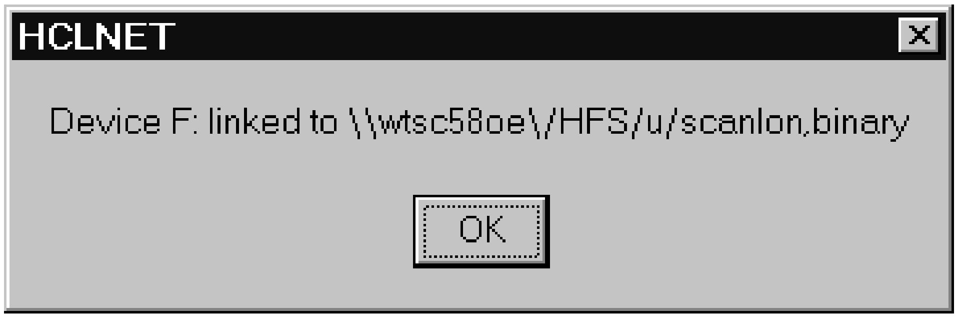
In the example shown in Figure 36 on page 68, the workstation user has mounted the F: drive on the /u/scanlon/ HFS directory with a binary connection.

Attention

Both a binary connection *and* a text NFS connection are required, because text files must be translated between ASCII and EBCDIC, while binary files are not translated.

Note: Sharing and Preserve Case must be checked on.

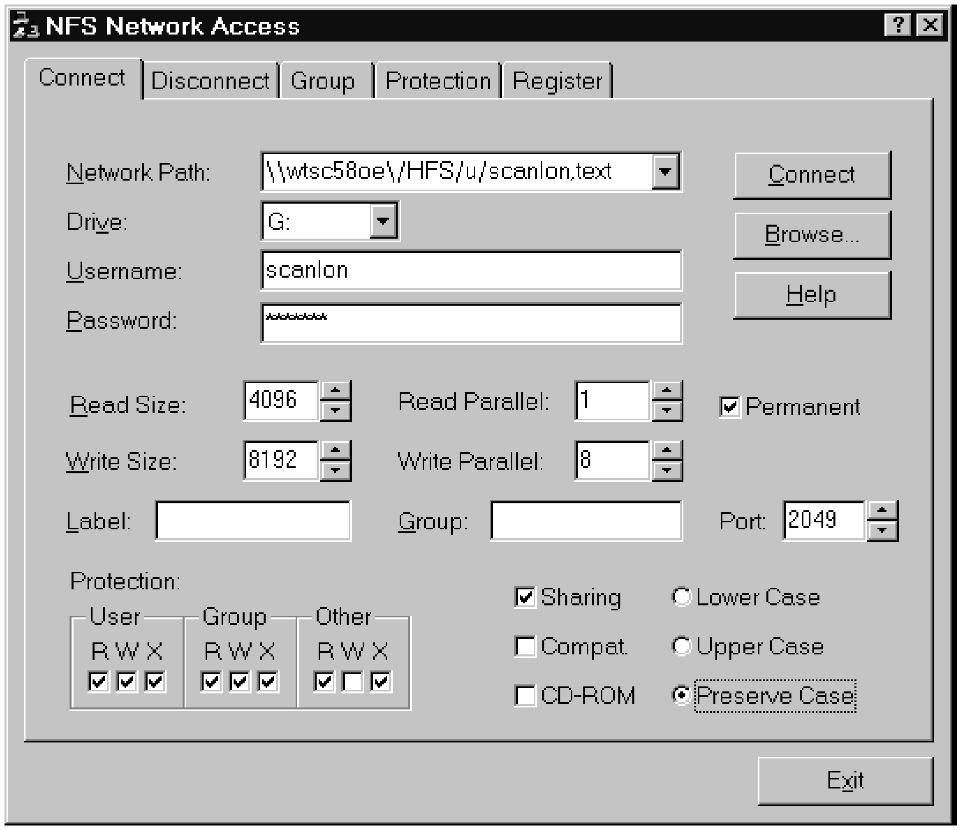
When a connection has been made successfully, the box shown in Figure 39 on page 73 is displayed.



Device F: linked to \\wtsc58oe\/HFS/u/scanlon,binary

Figure 39. Successful NFS Mount - Binary Connection

In the example shown in Figure 40, an NFS connection is created with a text connection.

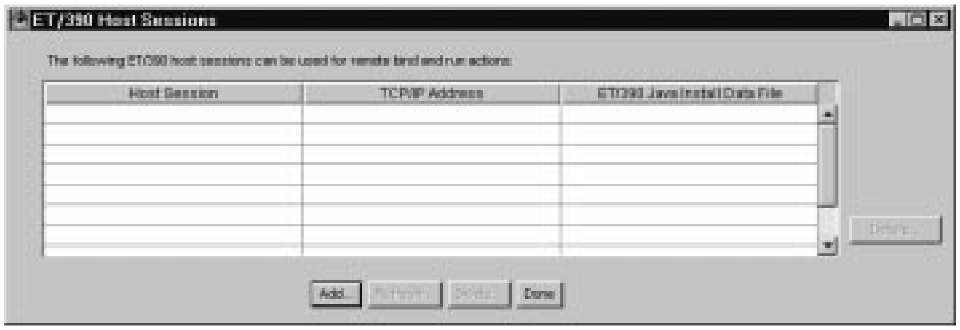


Network Path: \\wtsc58oe\/HFS/u/scanlon,text
Drive: G:
Username: scanlon
Password: *******
Read Size: 4096
Write Size: 8192
Read Parallel: 1
Write Parallel: 8
[✓ Permanent]
Label:
Group:
Port: 2049
Protection:
User R W X [✓] [✓] [✓]
Group R W X [✓] [✓] [✓]
Other R W X [✓] [ ] [✓]
[✓ Sharing]
[ ] Compat.
[ ] CD-ROM
Lower Case / Upper Case / [✓ Preserve Case]

Figure 40. NFS Set Up Using Hummingbird NFS Maestro - Text Connection

6.2 Establishing a Host Session

To establish a connection from VisualAge for Java Enterprise Edition Version 2.0 to an ET/390 host session on OS/390, you should open, or be in, a **Workbench** session and then choose **Workspace** from the menu bar and then **Tools**, **ET/390**, **Host Session**. This will show the screen in Figure 41 on page 74.

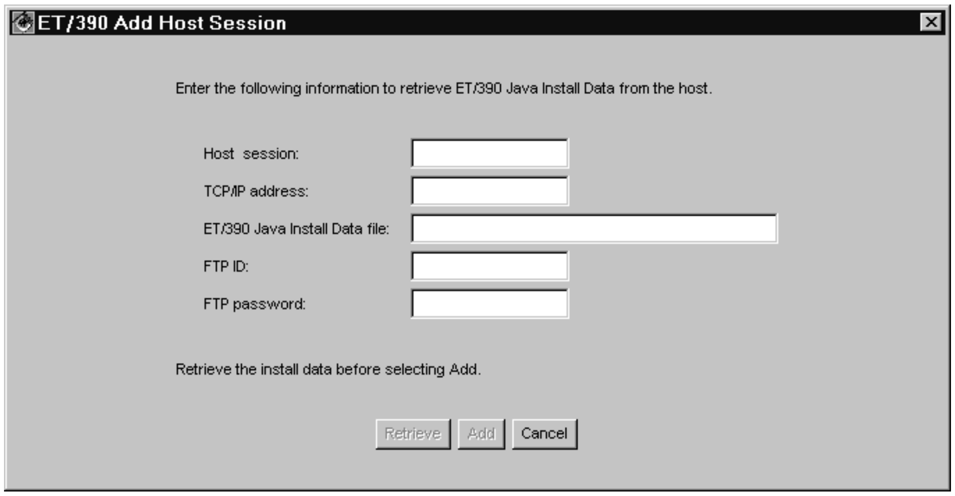


The following ET/390 host sessions can be used for remote bind and run actions:

Host Session	TCP/IP Address	ET/390 Java Install Data File

Figure 41. Setting Up a Host Session

If it is the first time a host session is set up, there will be no entries in the table. The **Add** button should be clicked, at which point you will see the frame shown in Figure 42.



Enter the following information to retrieve ET/390 Java Install Data from the host.

Host session:	
TCP/IP address:	
ET/390 Java Install Data file:	
FTP ID:	
FTP password:	

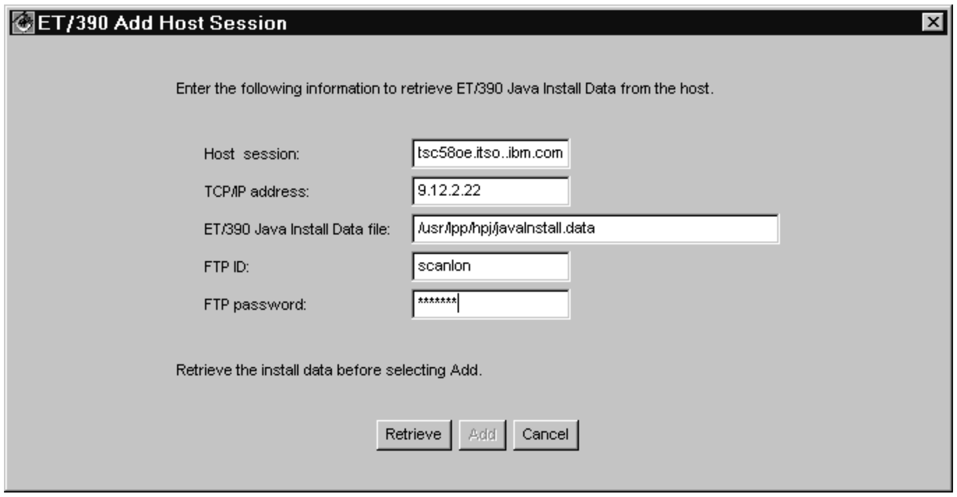
Retrieve the install data before selecting Add.

Figure 42. Adding a Host Session

You must fill in the five text fields.

- The host session is the full IP name (computer name and domain) of the OS/390 system you want to connect to.
- The TCP/IP address is the IP address of the host defined in the host session.
- The ET/390 Java Install Data file is the fully qualified file name of the javaInstall.data file that was set up by your systems programmer on the OS/390. It is likely that it will be in the HFS /usr/lpp/hpj/ directory with a filename of javaInstall.data, but you should get this path information from your systems programmer.
- Your FTP ID and FTP password will most likely be your user ID and password on the OS/390 System, but you should get this information from your OS/390 systems administrator or systems programmer.

Figure 43 on page 75 shows the screen after you fill in the values.


ET/390 Add Host Session

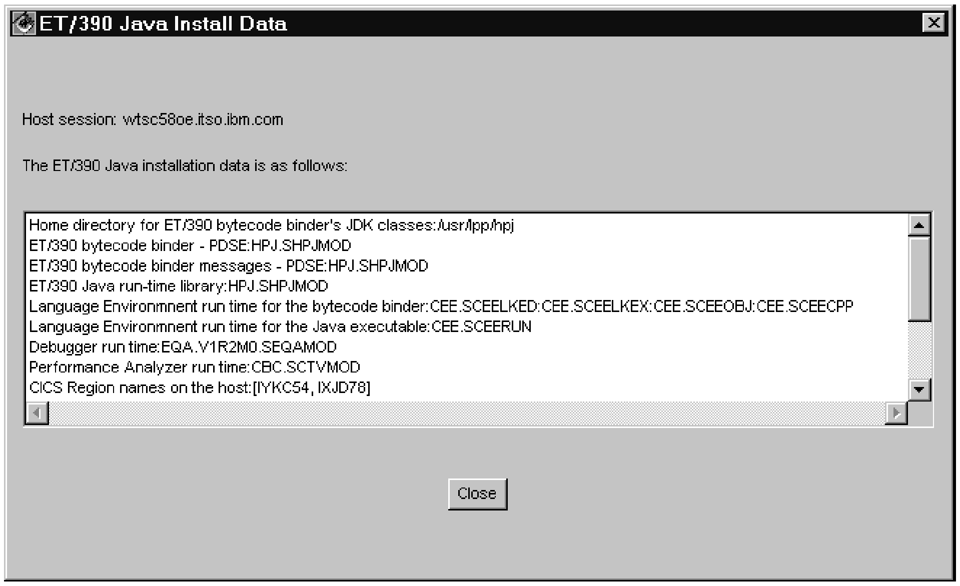
Enter the following information to retrieve ET/390 Java Install Data from the host.

Host session:	tsc58oe.itso..ibm.com
TCP/IP address:	9.12.2.22
ET/390 Java Install Data file:	/usr/lpp/hpj/javaInstall.data
FTP ID:	scanlon
FTP password:	*******

Retrieve the install data before selecting Add.

Figure 43. Adding a Host Session with Values

Prior to adding the host session to the list of ET/390 host sessions that you can connect to, it is necessary to retrieve the javaInstall.data file from the host. Click **Retrieve** on this panel to retrieve the javaInstall.data file parameters from the host and display them as shown in Figure 44 and Figure 45 on page 76.

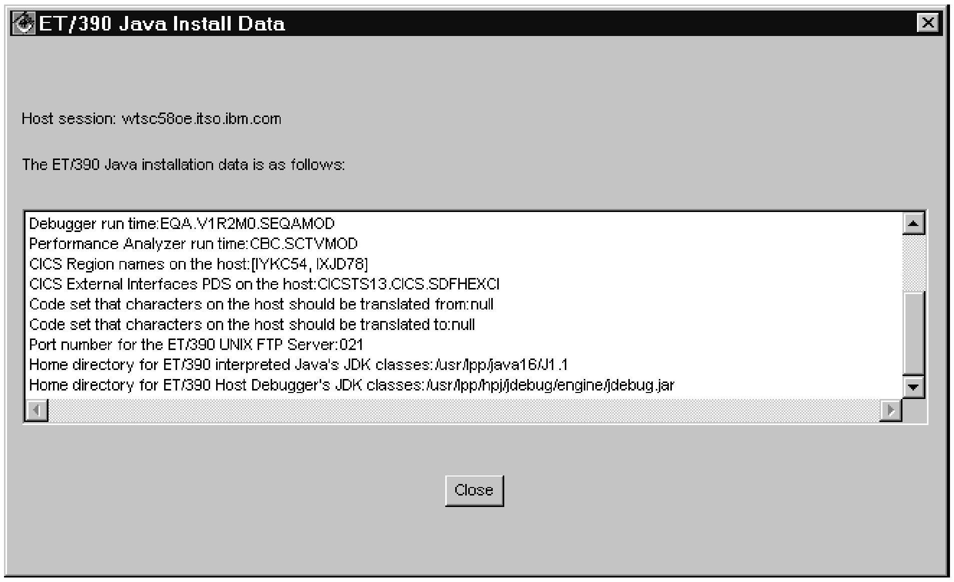

ET/390 Java Install Data

Host session: wtsc58oe.itso.ibm.com

The ET/390 Java installation data is as follows:

```
Home directory for ET/390 bytecode binder's JDK classes:/usr/lpp/hpj
ET/390 bytecode binder - PDSE:HPJ.SHPJMOD
ET/390 bytecode binder messages - PDSE:HPJ.SHPJMOD
ET/390 Java run-time library:HPJ.SHPJMOD
Language Environmnent run time for the bytecode binder:CEE.SCEELKED:CEE.SCEELKEX:CEE.SCEEOBJ:CEE.SCEECPP
Language Environmnent run time for the Java executable:CEE.SCEERUN
Debugger run time:EQA.V1R2M0.SEQAMOD
Performance Analyzer run time:CBC.SCTVMOD
CICS Region names on the host:[IYKC54, IXJD78]
```

Figure 44. javaInstall.data File Part 1


ET/390 Java Install Data

Host session: wtsc58oe.itso.ibm.com

The ET/390 Java installation data is as follows:

```
Debugger run time:EQA.V1R2M0.SEQAMOD
Performance Analyzer run time:CBC.SCTVMOD
CICS Region names on the host:[IYKC54, IXJD78]
CICS External Interfaces PDS on the host:CICSTS13.CICS.SDFHEXCI
Code set that characters on the host should be translated from:null
Code set that characters on the host should be translated to:null
Port number for the ET/390 UNIX FTP Server:021
Home directory for ET/390 interpreted Java's JDK classes:/usr/lpp/java16/J1.1
Home directory for ET/390 Host Debugger's JDK classes:/usr/lpp/hpj/jdebug/engine/jdebug.jar
```

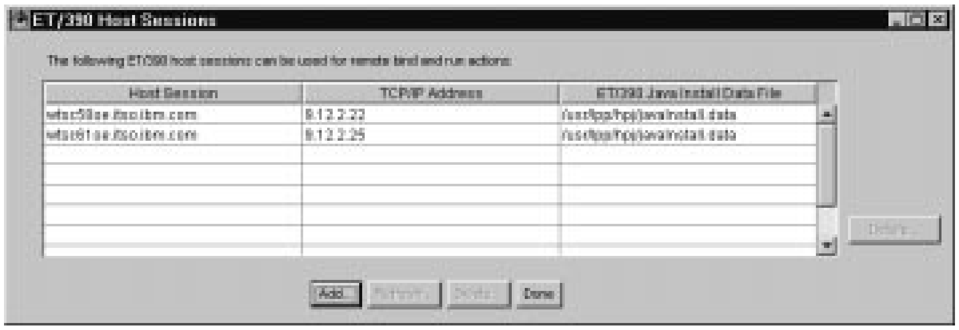
Figure 45. javaInstall.data File Part 2

Check the settings in this file. If any are incorrect, have your systems programmer change them. The javaInstall.data file settings will be downloaded and stored locally at the IDE workstation. Parts of the javaInstall.data file will be used to build the *.cmd* file that will be sent to the /tmp/ directory on OS/390 for an Export and Bind, Run Executable, or Run Main (Run interpreted bytecode on OS/390 JVM) operation.

Parts of the javaInstall.data file will be used to provide default values in the Export and Bind properties table, such as: the CICS Region field and the host session name.

When you are satisfied that all the specifications in the javaInstall.data file are correct, click **Close**. You will then be able to click **Add** on the **ET/390 Add Host Session** dialog box.

Figure 46 shows the ET/390 Host Sessions Dialog box after two host sessions were added. Click **Done** to complete the Add a Host Session task.


ET/390 Host Sessions

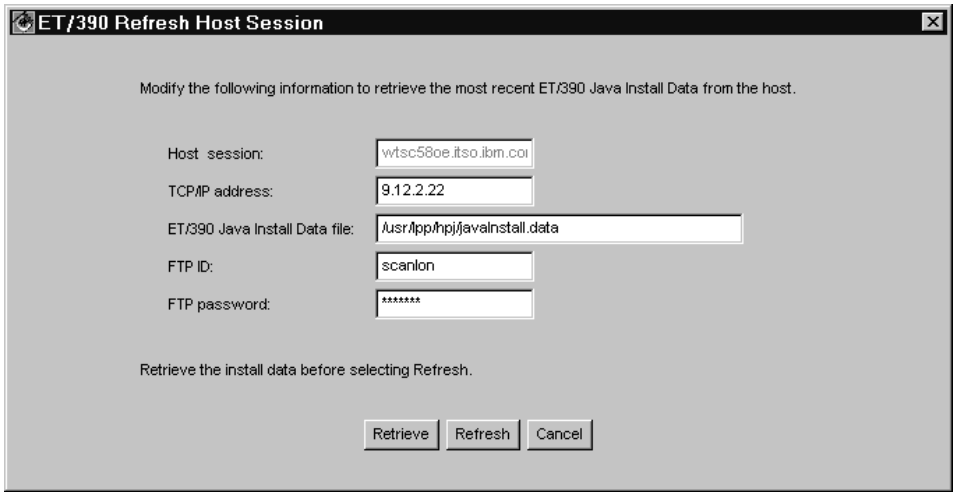
The following ET/390 host sessions can be used for remote bind and run actions:

Host Session	TCP/IP Address	ET/390 JavaInstallData File
wtsc58oe.itso.ibm.com	9.12.2.22	/usr/lpp/hpj/javaInstall.data
wtsc61oe.itso.ibm.com	9.12.2.25	/usr/lpp/hpj/javaInstall.data

Figure 46. ET/390 Host Sessions Dialog Box

If the javaInstall.data file is updated on the OS/390, it will be necessary to refresh the Host Session by entering the Host Session dialog box shown in Figure 46 once again and highlighting the host session you wish to refresh, and then clicking **Refresh**.

You will then see the ET/390 Refresh Host Sessions dialog box shown in Figure 47.



Modify the following information to retrieve the most recent ET/390 Java Install Data from the host.

Host session: wtsc58oe.itso.ibm.coi
TCP/IP address: 9.12.2.22
ET/390 Java Install Data file: /usr/lpp/hpj/javaInstall.data
FTP ID: scanlon
FTP password: *******

Retrieve the install data before selecting Refresh.

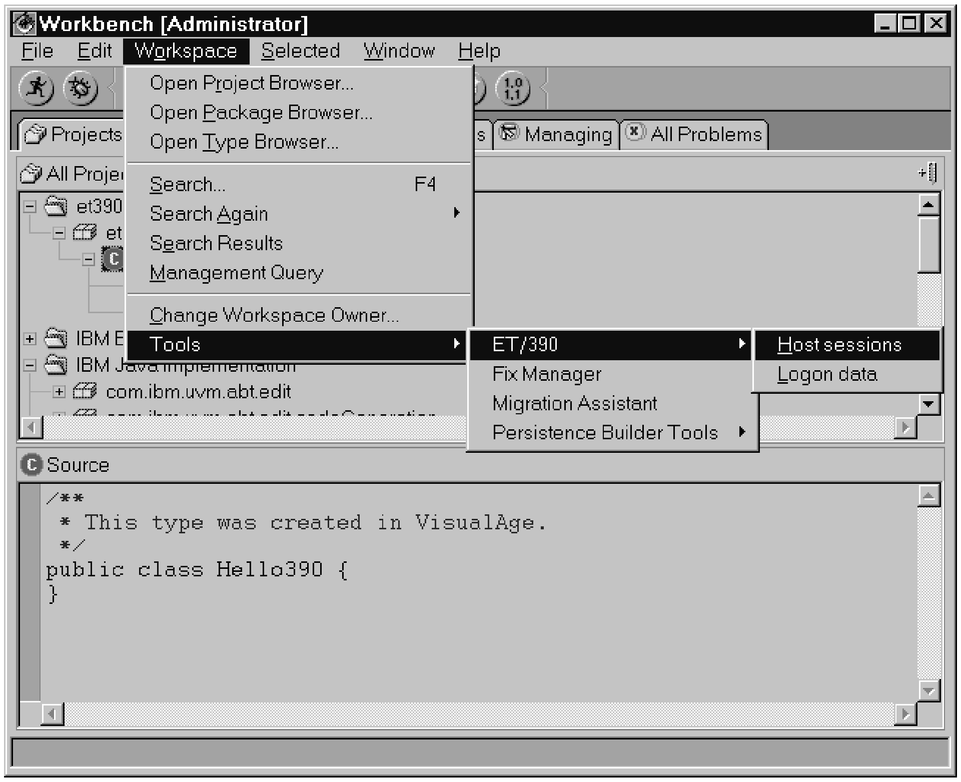
Figure 47. Refreshing a Host Session

You can click **Retrieve** to verify the changes to be sure they are correct, and click **Refresh** and then click **Done** in the ET/390 Host Sessions Dialog Box to complete the Refresh task.

6.3 Logging on to OS/390

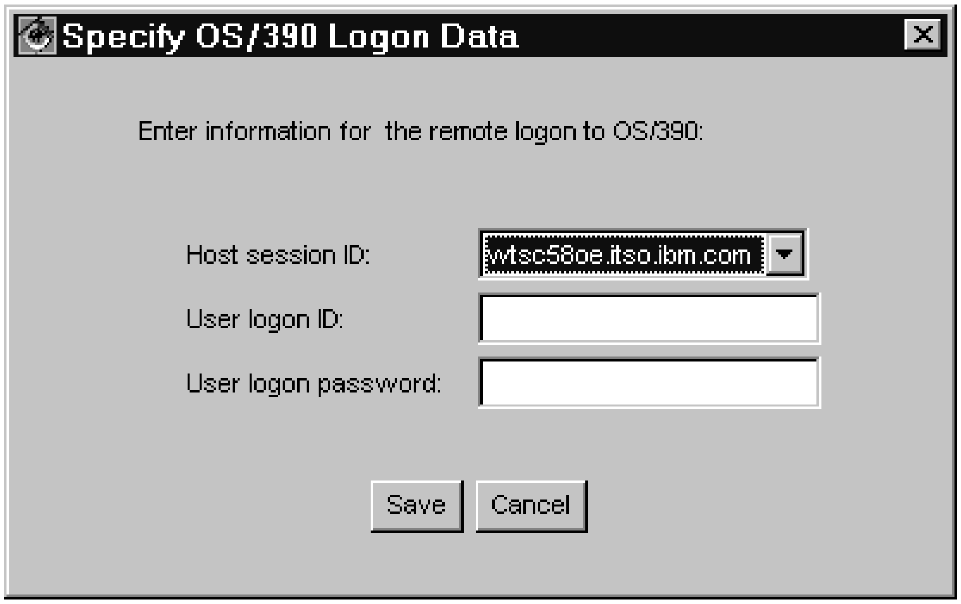
You must be logged on to one of the OS/390 sessions that you have established a host session for. This will allow you to export and bind to, or execute a Java program on, that host session.

You can log on to a host session by choosing **Workspace** from the Menu bar, then choosing **Tools**, **ET/390**, **Logon Data** as shown in Figure 48 on page 78. This will present you with the Specify OS/390 Logon Data Dialog Box as shown in Figure 49 on page 78.



```
/**
 * This type was created in VisualAge.
 */
public class Hello390 {
}
```

Figure 48. Logon to OS/390 Menu Option


Specify OS/390 Logon Data

Enter information for the remote logon to OS/390:

Host session ID: wtsc58oe.itso.ibm.com

User logon ID:

User logon password:

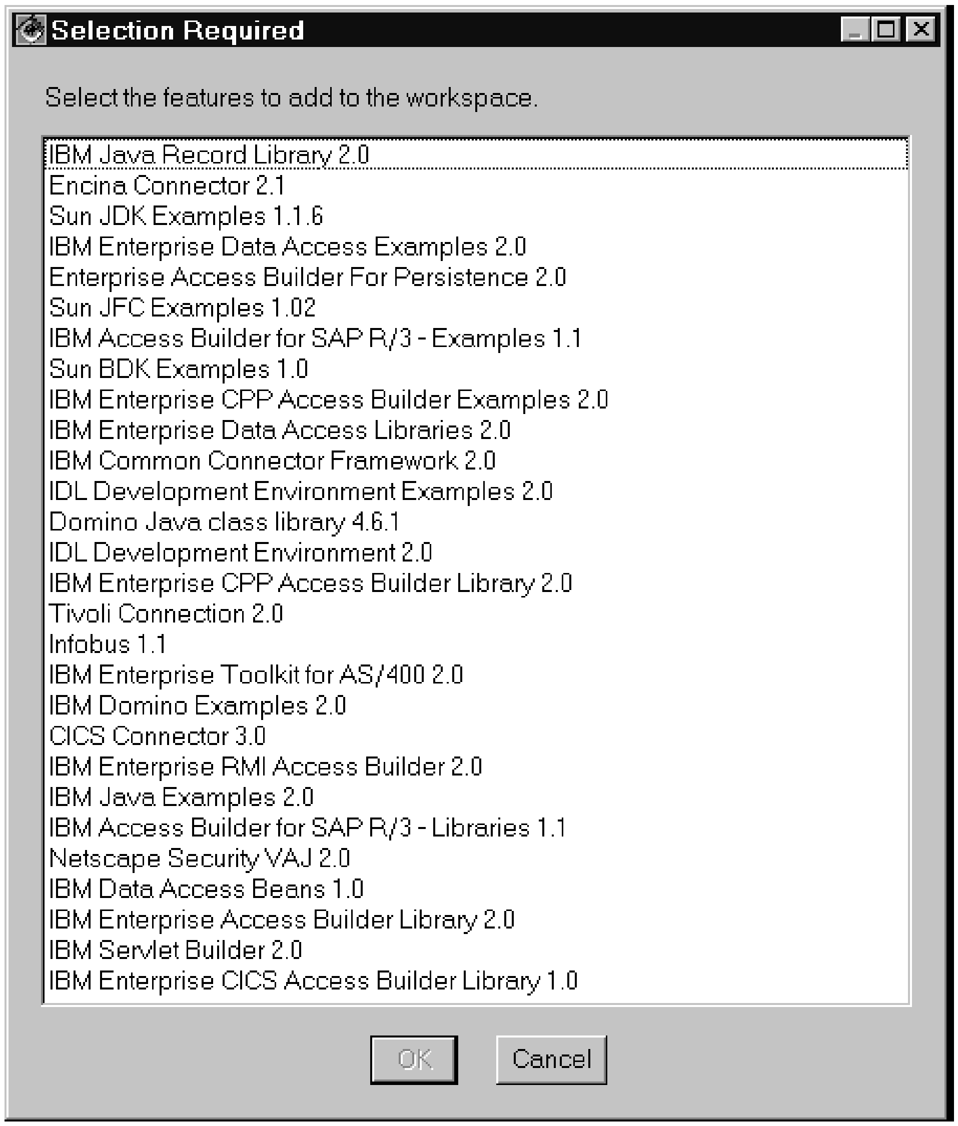
Figure 49. Logon to OS/390 Window

You will have to enter your OS/390 User logon ID and your User logon password and then click **Save** to complete the logon to the OS/390 task.

6.4 Installing VisualAge for Java Features

To install VisualAge for Java Features from the Workbench environment, select **File**, then **Quick Start**, then **Features**.

You will see the list box with several features as shown in Figure 50.



Select the features to add to the workspace.

IBM Java Record Library 2.0
Encina Connector 2.1
Sun JDK Examples 1.1.6
IBM Enterprise Data Access Examples 2.0
Enterprise Access Builder For Persistence 2.0
Sun JFC Examples 1.02
IBM Access Builder for SAP R/3 - Examples 1.1
Sun BDK Examples 1.0
IBM Enterprise CPP Access Builder Examples 2.0
IBM Enterprise Data Access Libraries 2.0
IBM Common Connector Framework 2.0
IDL Development Environment Examples 2.0
Domino Java class library 4.6.1
IDL Development Environment 2.0
IBM Enterprise CPP Access Builder Library 2.0
Tivoli Connection 2.0
Infobus 1.1
IBM Enterprise Toolkit for AS/400 2.0
IBM Domino Examples 2.0
CICS Connector 3.0
IBM Enterprise RMI Access Builder 2.0
IBM Java Examples 2.0
IBM Access Builder for SAP R/3 - Libraries 1.1
Netscape Security VAJ 2.0
IBM Data Access Beans 1.0
IBM Enterprise Access Builder Library 2.0
IBM Servlet Builder 2.0
IBM Enterprise CICS Access Builder Library 1.0

Figure 50. Features Window

Choose the feature you need and select **Add Feature**.

One of the optional features is Servlet Builder. This feature was heavily used in the development of the sample applications for this book. The Servlet Builder adds two projects to the workbench and adds the `jsdk` class libraries.

6.5 Setting Up ET/390 Properties Tables

Java Applications can be developed on the VisualAge for Java Workstation IDE and run on OS/390 UNIX or CICS/ESA environments. Every project, package and class has a set of ET/390 properties associated with it to allow the developer to control the actions that are to be performed on OS/390. The ET/390 properties can be set in panels associated with the following Sessions:

- Export and Bind Session
  - Bind Options
  - Advanced Bind Options
- Run Executable Session
  - Run Time Options
  - Advanced Run Time Options
  - Debugging Options
  - Tracing Options
- Run Main Session
  - Run Time Options

Property tables are required on a per project basis, or on a per package or per class basis, if desired.

Note: The VisualAge for Java, Enterprise Edition for OS/390 IDE provides excellent help on all of the panels required for setting up the properties. With the panel displayed, click **F1** and you will receive detailed help on that panel, with links to associated topics.

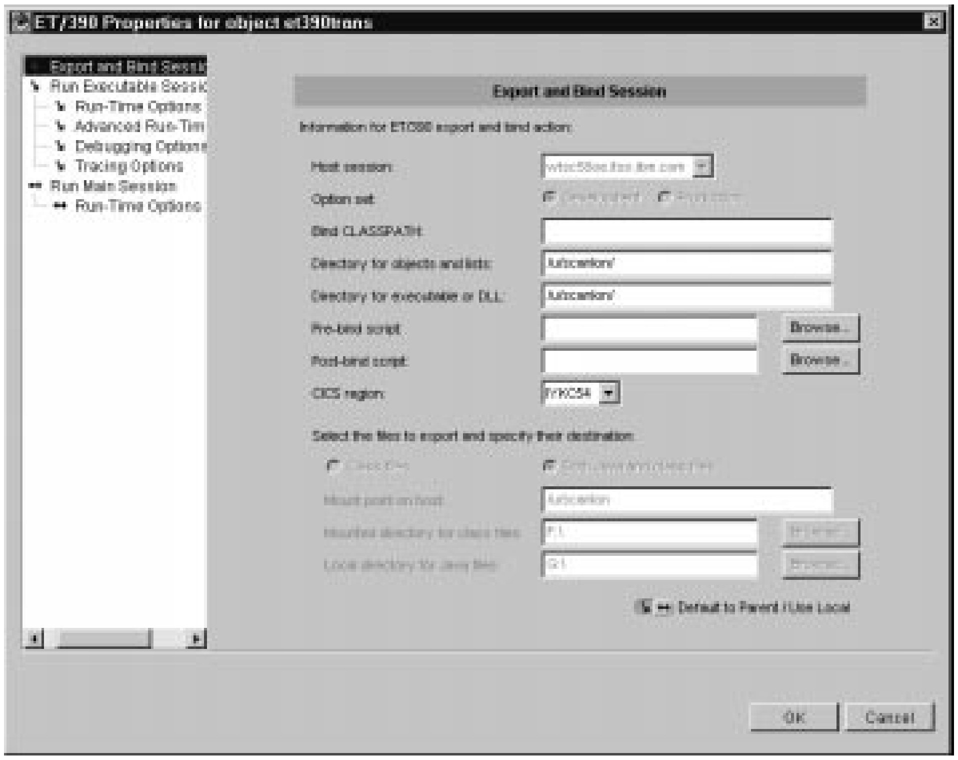
6.5.1 Setting Up Properties for an Export and Bind Session

VisualAge for Java, Enterprise Edition for OS/390 allows the developer to build OS/390 Java Executables from the workstation IDE and export and bind the executable to OS/390.

Before performing the Export and Bind operation, the developer must first set the ET/390 properties in the Export and Bind Session panel. This is done as follows:

- Select the package you want to export and either click the right mouse button or click **selected** from the menu bar. From the pop-up window, select **Tools...ET/390...Properties**. In the ET/390 Properties window, select **Export and Bind Session**.
- You can either accept the inherited values if settings were previously done for the project that this package belongs to, or you can set the fields for this package - this choice is established by choosing **Default to parent** or **Use Local**. Not all fields can be set at the package level; those that cannot be selected will be grayed out.
- In the panel, set **Host Session, Option set, Mount point on host, and Mounted directory for class files**. Also set the other options that you need.

  See Figure 51 on page 81 for the Export and Bind session.


ET/390 Properties for object et390trans

Export and Bind Session

Information for ET/390 export and bind action:

Host session:	wtsc58oe.itso.ibm.com
Option set:	
Bind CLASSPATH:	
Directory for objects and lists:	/u/tcarlon/
Directory for executable or DLL:	/u/tcarlon/
Pre-bind script:	
Post-bind script:	
CICS region:	IYKC54

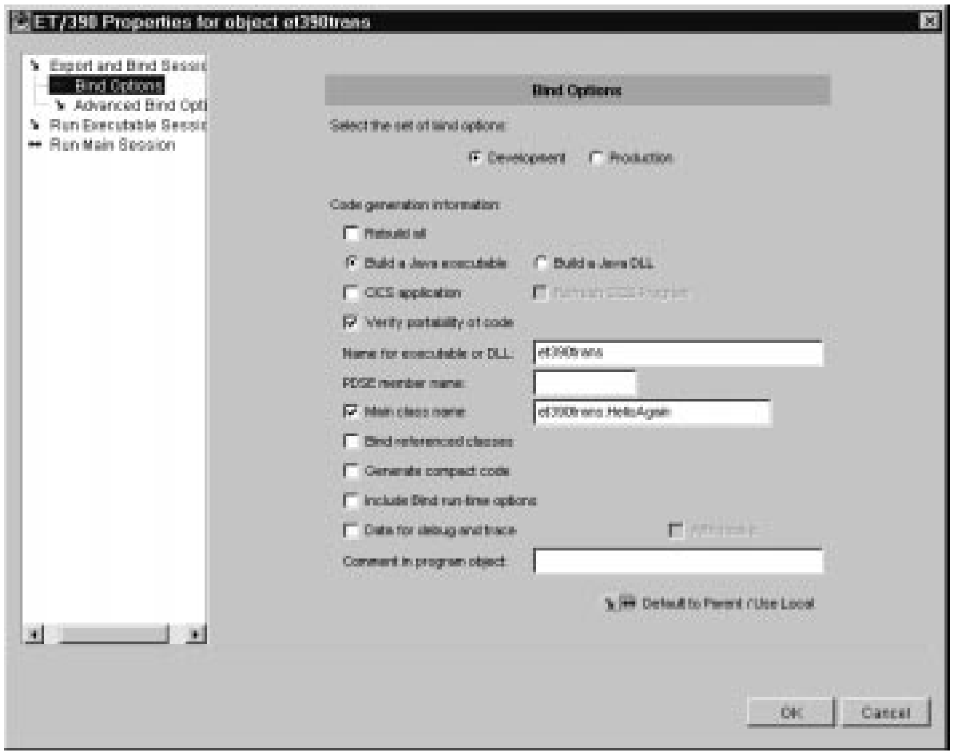
Select the files to export and specify their destination:

Mount point on host:	/u/tcarlon
Mounted directory for class files:	F:\
Local directory for Java files:	G:\

Figure 51. Export and Bind Session

- Select the Bind Options panel and select the **Build a Java Executable** option. Specify **Main class name** if there is more than one main class in the package. Set any other options that you require.

  See Figure 52 on page 82 for the Bind Options.


ET/390 Properties for object et390trans

Bind Options

Select the set of bind options: [✓ Development] Production

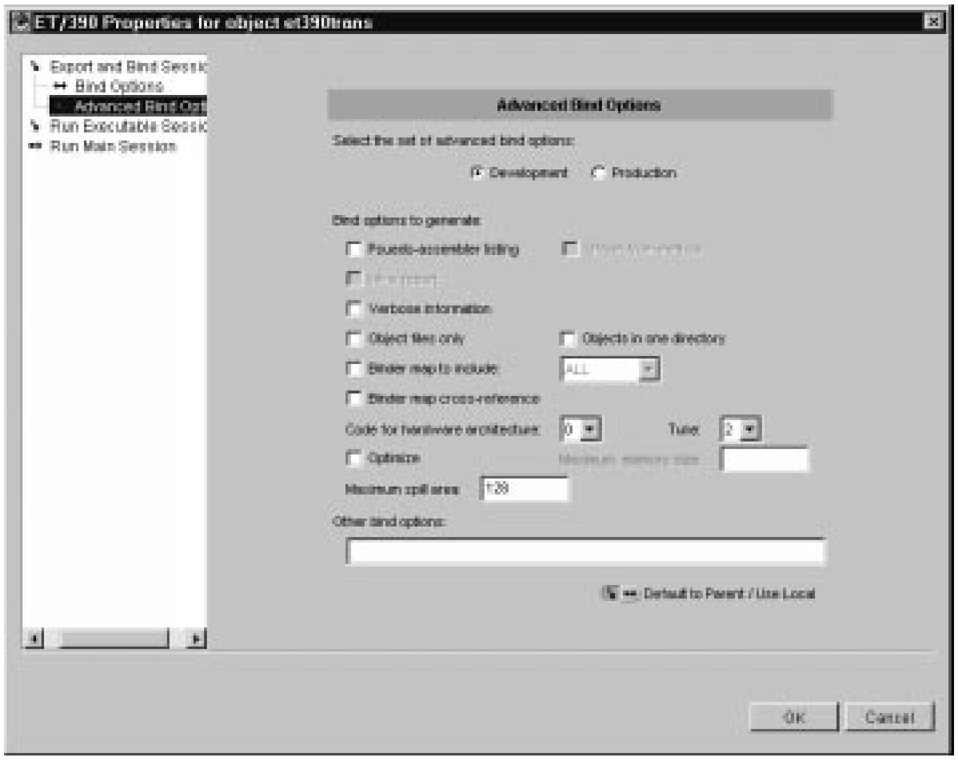
Code generation information:

- [ ] Rebuild all
- [✓ Build a Java executable] Build a Java DLL
- [ ] CICS application
- [✓] Verify portability of code
- Name for executable or DLL: et390trans
- PDSE member name:
- [✓] Main class name: et390trans.HelloAgain
- [ ] Bind referenced classes
- [ ] Generate compact code
- [ ] Include Bind run-time options
- [ ] Data for debug and trace
- Comment in program object:

Figure 52. Bind Options

- Select the **Advanced Bind Options** panel and then select any options that you require.

  See Figure 53 on page 83 for the Advanced Bind Options.



Advanced Bind Options

Select the set of advanced bind options:

[✓ Development] Production

Bind options to generate:

- [ ] Pseudo-assembler listing
- [ ] Verbose information
- [ ] Object files only
- [ ] Objects in one directory
- [ ] Binder map to include: ALL
- [ ] Binder map cross-reference

Code for hardware architecture: 0 Tune: 2

- [ ] Optimize

Maximum spill area: 128

Other bind options:

Default to Parent / Use Local

Figure 53. Advanced Bind Options

6.5.2 Setting Up the Properties for a Run Executable Session

The ET/390 Run Executable Session Properties panel provides configuration information for the Run Executable and Debug Executable on OS/390.

The **CLASSPATH** is used as the search path for Java DLLs residing on HFS and in PDSE members.

See the online help for the other fields and their contents.

6.5.3 Run Time Options

Taken together, the run time options, the advanced runtime options, the debugging options and the tracing options list the complete set of Language Environment run-time options that can be used when ET/390 Run Executable and Debug Executable actions on objects are performed.

See the online help for detailed explanations of each option field. The following discussion offers additional detail on setting up the debug options.

6.5.4 Setting Up Properties for Debugging Options

The debugging options are used when you perform a Debug Executable action on objects. The following options can be set:

- Start the debugger at program initiation

  Starts the debugger on the host when the program is initialized and passes the conditions (All conditions, On error, or No conditions) to the debugger.

This is equivalent to the Language Environment runtime option TEST.

Note: Use the ET/390 debug executable action to run the debugger with the executable on OS/390.

- Pass Conditions to the Debugger

  Specifies the condition for which the debugger runs.

- Content of dump

  Sets the level of information that is produced when the Language Environment percolates a condition of severity 2 or greater beyond the first routine's stack frame.

- Storage information

  Specifies the initialized value of heap storage when allocated, the value with which to overwrite freed heap storage, DSA's initial value, and the amount of storage for the LE storage manager to reserve in the event of an out-of-storage condition.

- LOCAL TCP/IP port of the daemon

  Specifies the port number (4 digits) for the debugger daemon's port on the workstation.

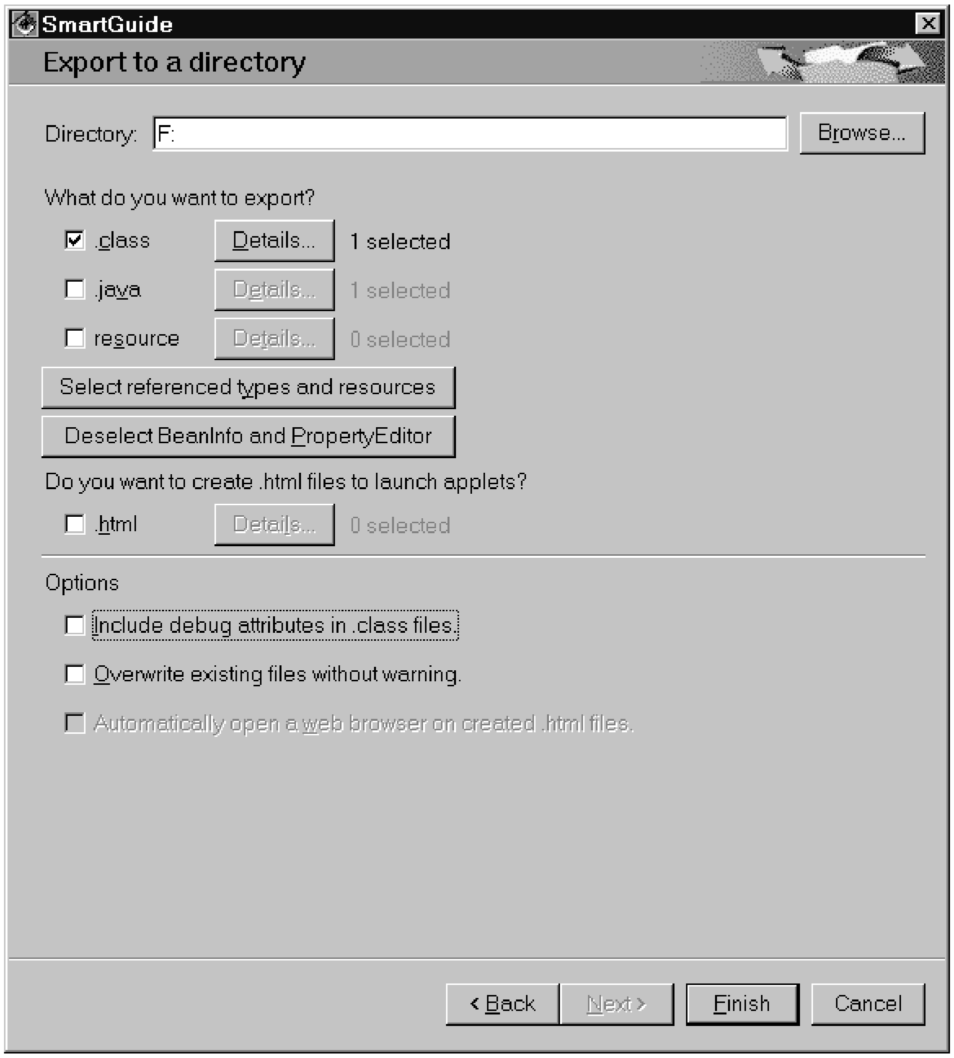
Note: See the online help available via F1 for details on all of the above settings.

6.5.5 Setting Up the Properties for a Run Main Session

You can run your project's bytecode that has main methods from the VisualAge for Java IDE. This bytecode is is set up to run on the OS/390 JVM in the following manner:

1. Generate the source code on the IDE and compile it (performed on SAVE at the IDE).
2. Make sure you have an NFS binary connection to the OS/390 host.
3. Select the package or class that you want to run on OS/390. Select **File** from the menu bar and then **Export**.
4. In the Export SmartGuide, select the class files you want to export via a binary connection.
5. Specify a drive that is mounted as binary on an HFS directory.
6. Click **Finish**.

See Figure 54 on page 85.


SmartGuide

Export to a directory

Directory: F:

What do you want to export?

[✓] .class — 1 selected

[ ] .java — 1 selected

[ ] resource — 0 selected

Do you want to create .html files to launch applets?

[ ] .html — 0 selected

Options

[ ] Include debug attributes in .class files.

[ ] Overwrite existing files without warning.

[ ] Automatically open a web browser on created .html files.

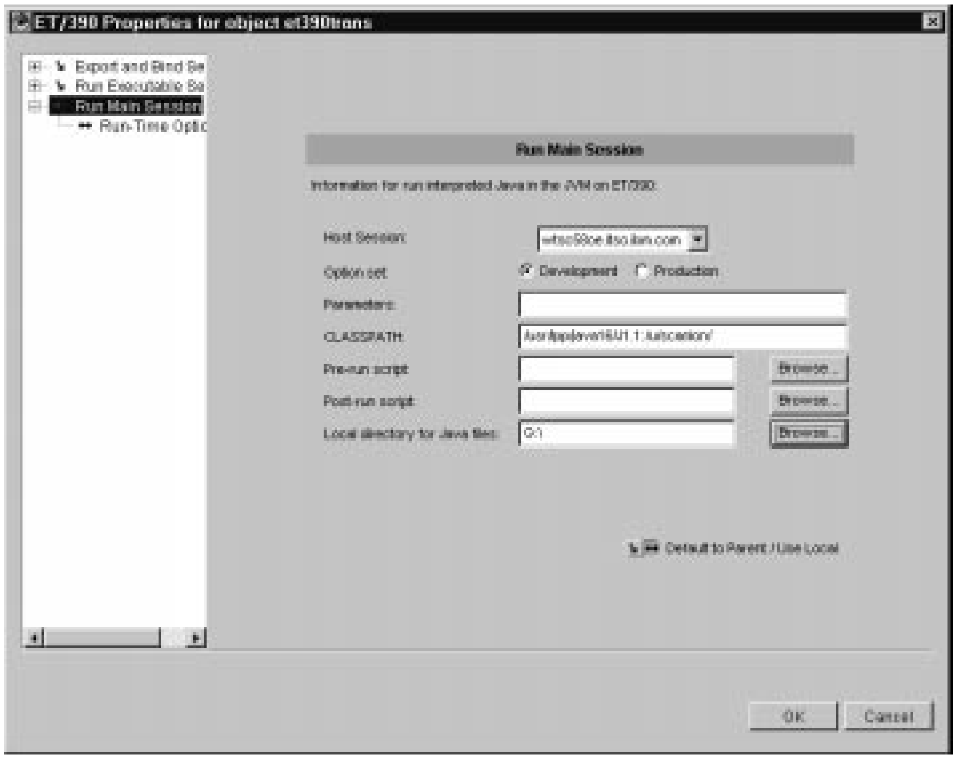
Browse...
Details...
Select referenced types and resources
Deselect BeanInfo and PropertyEditor
< Back
Next >
Finish
Cancel

Figure 54. Export a Class File

Now set the ET/390 properties in the Run Main Session panel in the following manner:

- Host Session - Specify the host session you want to run the bytecode on. Note that this specification is for the *entire project*.
- CLASSPATH - Specify the CLASSPATH that should be used by the OS/390 JVM to search for the required Java classes when your bytecode is running.
- Local directory for Java files - Enter the location of the source files on OS/390 for the bytecode you are going to run on OS/390 if you wish to run the bytecode with Debugging.

See Figure 55 on page 86.


ET/390 Properties for object et390trans

Run Main Session

Information for run interpreted Java in the JVM on ET/390:

Host Session: wtsc59oe.itso.ibm.com

Option set: [✓ Development] Production

Parameters:

CLASSPATH: /usr/lpp/java18/J1.1:/u/scenion/

Pre-run script:

Post-run script:

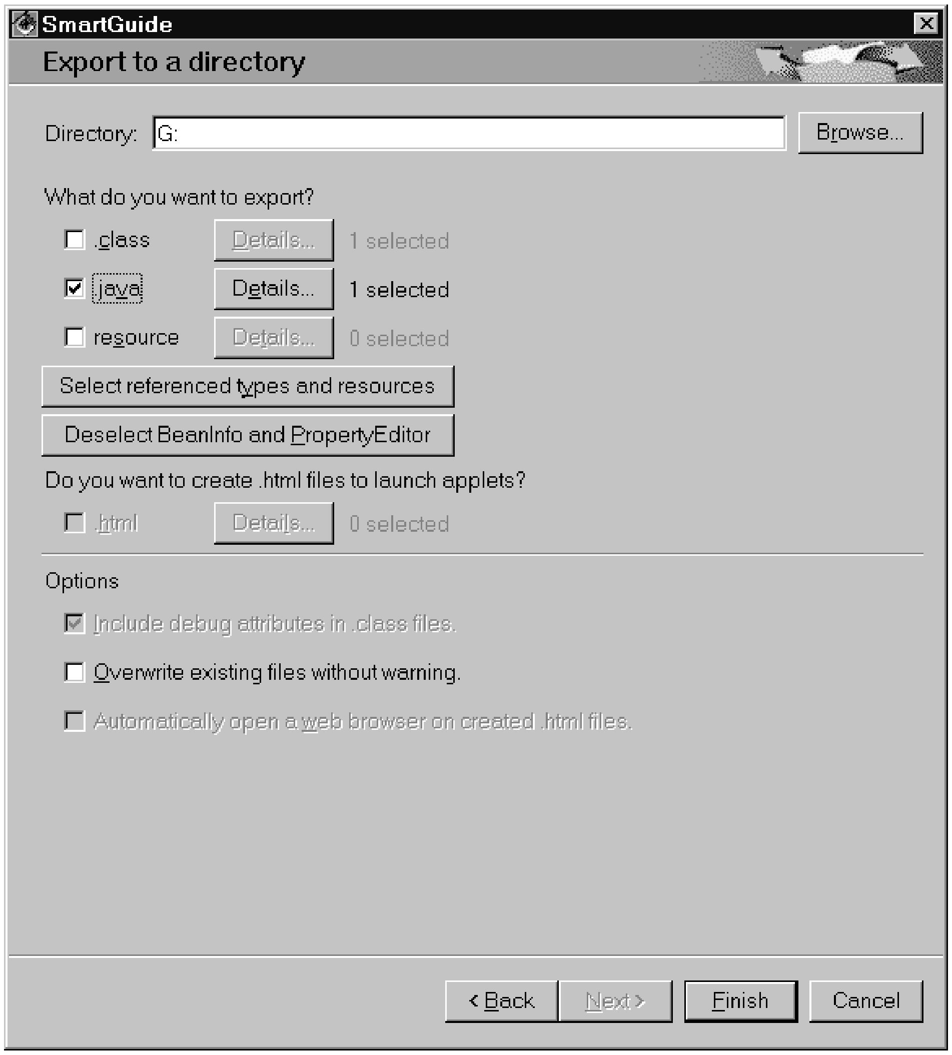
Local directory for Java files: G:\

Figure 55. Run Main Properties

To export the source files, which is required if you want to debug the bytecode on the OS/390, perform the following:

1. In the Export SmartGuide, select the source files you want to export via a text connection.
2. Specify a drive that is mounted as text on an HFS directory. (see Figure 56 on page 87)
3. Click **Finish** to export the code to the directory on the OS/390 HFS.

See Figure 56 on page 87.


SmartGuide

Export to a directory

Directory: G:

What do you want to export?

- [ ] .class — 1 selected
- [x] .java — 1 selected
- [ ] resource — 0 selected

Do you want to create .html files to launch applets?

- [ ] .html — 0 selected

Options

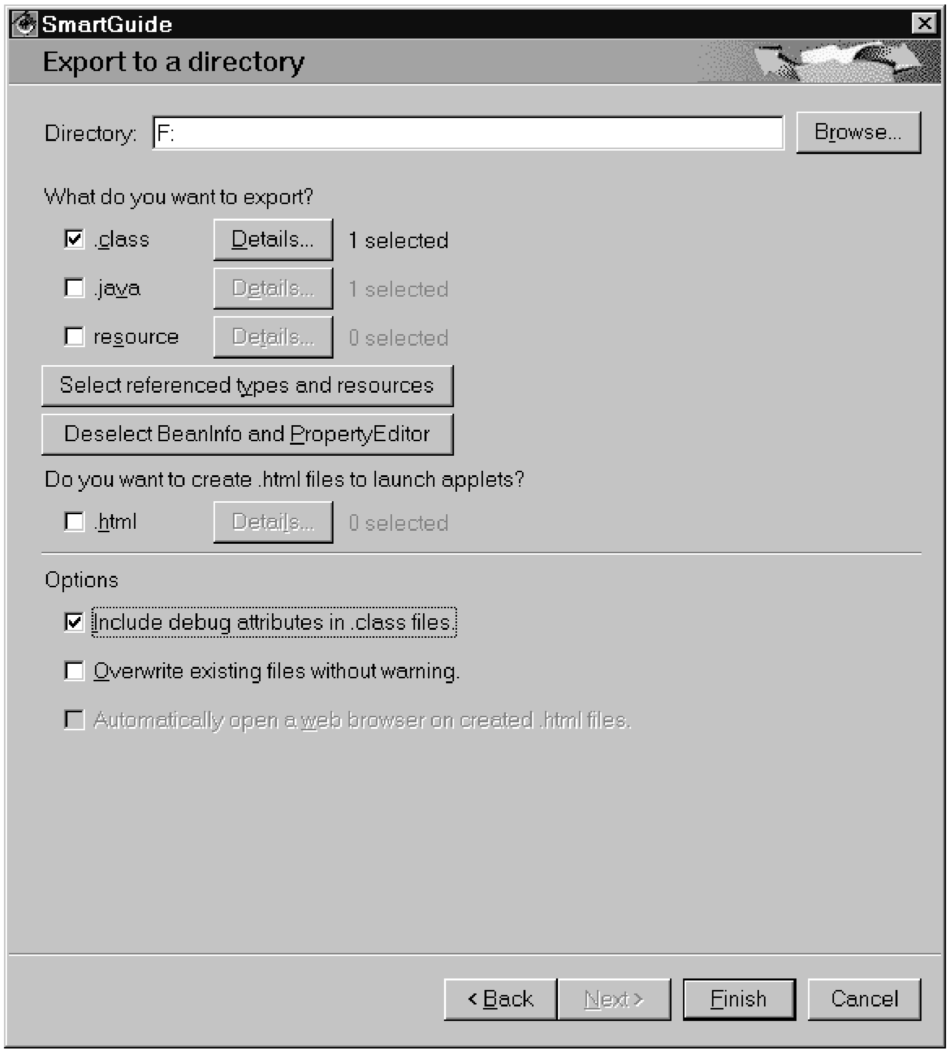
- [x] Include debug attributes in .class files.
- [ ] Overwrite existing files without warning.
- [ ] Automatically open a web browser on created .html files.

Browse...
Details...
Select referenced types and resources
Deselect BeanInfo and PropertyEditor
< Back
Next >
Finish
Cancel

Figure 56. Export a Source File

To export the class files such that they can be debugged when running the bytecode on OS/390, it is necessary to select the **Include debug attributes in .class files** box in the Export to a directory panel.

See Figure 57 on page 88.


SmartGuide

Export to a directory

Directory: F:

What do you want to export?

- [✓] .class — 1 selected
- [ ] .java — 1 selected
- [ ] resource — 0 selected

Select referenced types and resources

Deselect BeanInfo and PropertyEditor

Do you want to create .html files to launch applets?

- [ ] .html — 0 selected

Options

- [✓] Include debug attributes in .class files.
- [ ] Overwrite existing files without warning.
- [ ] Automatically open a web browser on created .html files.

Figure 57. Export a Class File for Debugging

6.5.6 Workstation Setup for the Debugger

6.5.6.1 Setting Up the Debugger Front End on the Workstation

The following steps should be performed to start the debugger daemon on the workstation:

- Make sure the IVJ_DBG_LANG environment variable is set to JAVA on the workstation.
- Start the remote debugger daemon on the workstation by issuing the following command:

  ```
  jdebugd -qport=nnnn -v
  ```

  – Where:

    -v means display connection information.

  – nnnn is the TCP/IP port number that the remote debugger daemon will listen for an incoming requests.

Note: This is the same port number that must be specified when the OS/390 executes the TEST runtime option.

Note: This port number should be specified as `jdebug   8000/tcp` in the `C:\WINNT\system32\drivers\etc\services` file on the NT workstation to insure that the proper port is obtained even if the port number is not specified in Local TCP/IP port for the daemon in the Debugging Properties, or a `jdebugd -qport=nnnn` command.

6.5.6.2 Starting the Debug Daemon

Java source files are required for debugging. They can reside either on the host or on the workstation. The `debug` daemon must be started on the workstation if the source files are kept there.

The debugger searches for Java source files for an OS/390 program that is being debugged in the following locations:

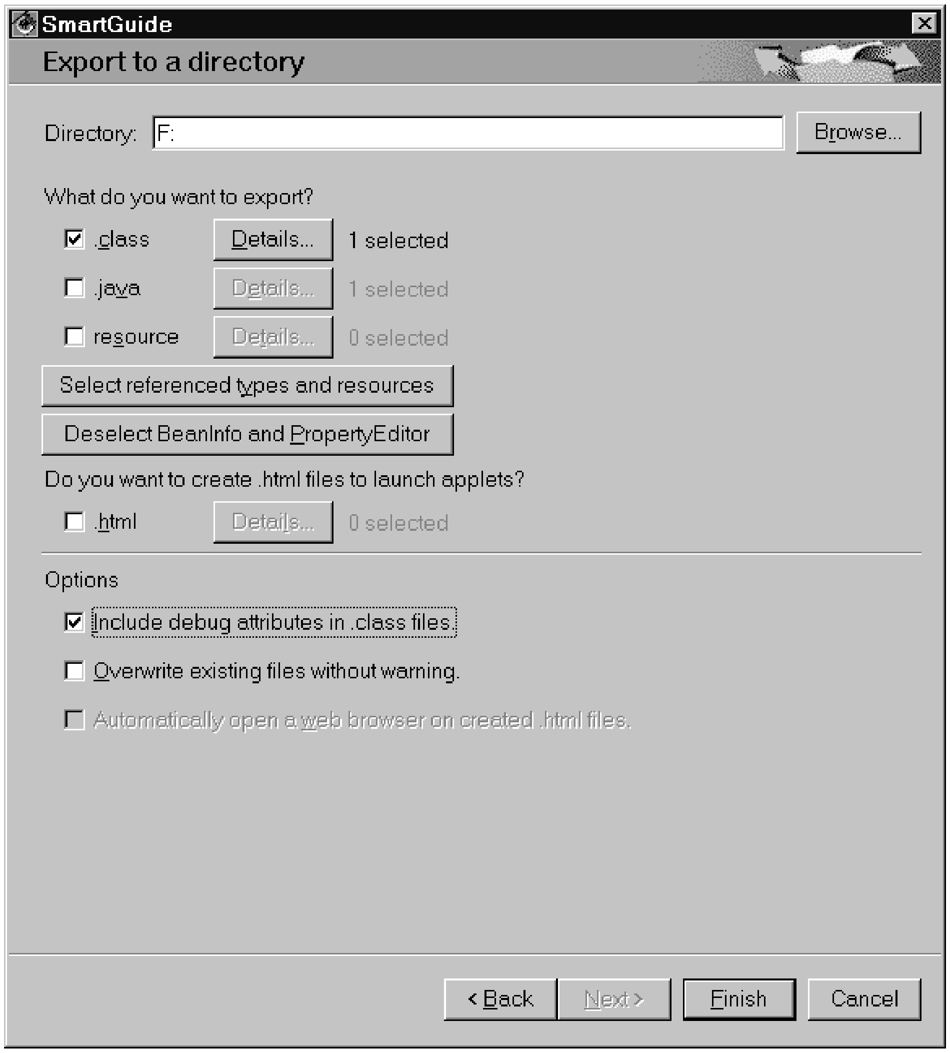
- The host file name specified at compile time
- Paths in the `export IVJ_DBG_PATH` environment variable on the machine where the application is running
- Paths in the `export IVJ_DBG_PATH` environment variable on the workstation where the debugger display is running

If you want to use the IDE for developing bytecode and then debug the bytecode from the IDE while it is running on OS/390 JVM, you have to export the bytecode to a local drive on OS/390.

This can be done as follows:

1. Be sure you have an NFS connection to OS/390 that will receive the bytecode file.
2. On the IDE, select the Java package or class that you want to run.
3. On the IDE, select **Files...Export** from the menu bar.
4. In the Export SmartGuide, choose **directory**, then in the Export to a directory window select the class files you want to export via a binary connection.
5. Specify a drive that is mounted as binary on an HFS directory.
6. Select **Include debug attributes in .class files**
7. Click **Finish** to export the .class files.

See Figure 58 on page 90.



Directory: F:

What do you want to export?

- [✓] .class — 1 selected
- [ ] .java — 1 selected
- [ ] resource — 0 selected

Do you want to create .html files to launch applets?

- [ ] .html — 0 selected

Options

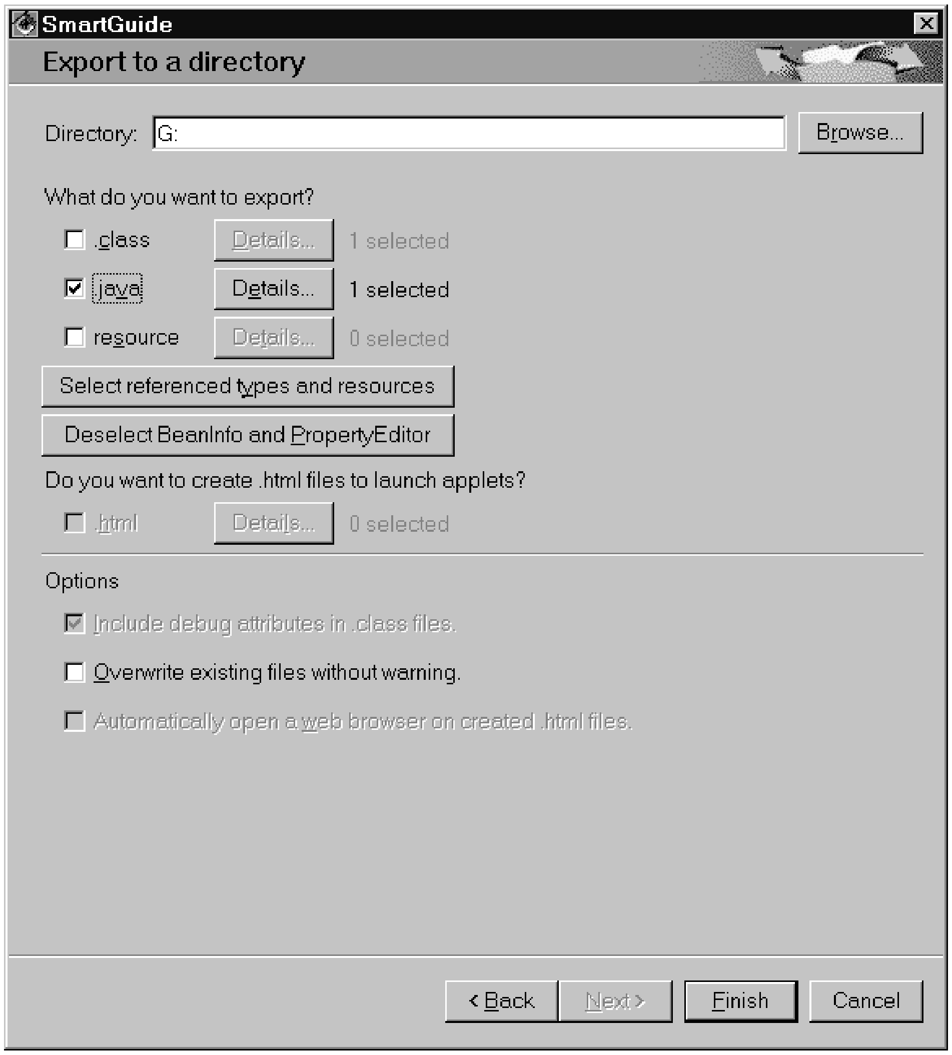
- [✓] Include debug attributes in .class files.
- [ ] Overwrite existing files without warning.
- [ ] Automatically open a web browser on created .html files.

Figure 58. Export a Class File for Debugging

To export the source files, which are required if you want to debug on OS/390, do the following:

1. In the Export SmartGuide, select the source files you want to export via a text connection.
2. Specify a drive that is mounted as text on an HFS directory.
3. Click **Finish** to export the code to the directory on the OS/390 HFS.

See Figure 59 on page 91


SmartGuide

Export to a directory

Directory: G:

What do you want to export?

- [ ] .class — 1 selected
- [x] java — 1 selected
- [ ] resource — 0 selected

Do you want to create .html files to launch applets?

- [ ] .html — 0 selected

Options

- [x] Include debug attributes in .class files.
- [ ] Overwrite existing files without warning.
- [ ] Automatically open a web browser on created .html files.

Browse...
Details...
Select referenced types and resources
Deselect BeanInfo and PropertyEditor
< Back
Next >
Finish
Cancel

Figure 59. Export a Source File

For remote debugging of Java bytecode running on OS/390 from the IDE workstation, it is necessary to connect a UI to the debugger engine. This is done on the workstation using the following steps:

1. Set up the environment using C:\IBMVJava\eab\bin\setdbg.bat (make sure you use the actual drive\directory where you installed VisualAge Java 2.0).

   Note: You must have JDK1.1.6 installed on your NT workstation for this bat file to execute correctly.

2. Start the debugger UI from the command line:

   ```
   jdebug -qhost=<hostname> [-qport=<TCP/IP port>]
   ```

   where:

   <hostname> specifies the OS/390 IP name where the host debugger is running. For example, wtsc58oe.itso.ibm.com.

   `-qport=<TCP/IP port>` sets the TCP/IP port on which the Workstation IDE UI will attempt to connect to the host debugger. If this option is omitted, the UI defaults to port 8000.

Chapter 7. NetObjects Fusion (NOF) Version 3

NetObjects Fusion (NOF) is a single-user oriented product designed to create and maintain Web sites. The HTML pages that make up the sample site and the pages on the CD included with this redbook were created using NetObjects Fusion. The CD also contains a 30-day trial version of NetObjects Fusion. To help you get started, the CD and sample sites were exported from NOF and can be imported using the trial version. Check the CD for additional information.

As a single user product, NOF is installed on a single machine, where the site definition is created and maintained. The site is then published to the target server. This process is the same no matter what type of system the target server is.

NOF uses FTP to perform the publishing, so it is important that you have FTP configured as part of TCP/IP on your OS/390 system. The FTP configuration should be set up to target the HFS.

The publishing section of NOF is easy to set up by doing the following:

1. Select the **Publish** icon. This will change the action bar specific to publishing.
2. Select the Publish action bar item and **Publish Setup** from the pull-down. This will cause a dialog box to appear.
3. Select the **HTML Output** tab. The Quote type at the bottom should be straight quotes. This will allow for the correct mapping of quotes on OS/390.
4. Select the **Server Locations** tab and then the **Add** button to create a new definition. Specify a name for your target OS/390 system and select the remote radio button. You need to fill in the TCP/IP name of the target system, the subdirectory where you want the HTML created, and the user ID and password you want to use for publishing.
5. Select the **Advanced** button. This will bring up an additional dialog box. Enter the correct permission bit settings (normally 775) so the Web server can read the pages and graphics.
6. Press **Ok** on all the dialog boxes, and you now have the server defined.
7. Press the **Publish Site** action pull-down.
8. Select the server you defined above and press **Ok**. The rest is automatic.

As projects grow larger and involve more people working on the Web site content, a single user product will quickly become a problem. NetObjects has a groupware product that will help in this area, NetObjects Authoring Server. NetObjects Authoring Server is a groupware version of NetObjects Fusion. The basic configuration is an NT server running the NetObjects Authoring Server code. Every client then installs the NetObjects Authoring Client code, and one (or more) person installs the Administration code. The administrator configures the server creating user IDs and granting authorizations.

The administrator then creates the base site, site style and structure, and configures the information about the target server where the Web site will be published to. (This can be any valid server that has FTP running and a Web server just like NetObjects Fusion). The client interface is similar to NOF except that before you can modify a page, you need to check it out and when finished modifying the page, you need to check it back in. If the initial site is built using NOF, you can also export the site and import it into NetObjects Authoring Server, giving you an easy migration path from a single user environment to a groupware environment.

In the rest of this chapter, we describe step by step the creation of a Web site using NetObjects Fusion.

7.1 Design of a Site

To develop enterprise Web applications, begin with Web site design, concentrating on the user interface. Decide on the following matters:

- Concepts through entire Web site
- Arrangement of static content
- Multimedia content
- Dynamic content (in other words, applications)

All Web interfaces consist of multiple static pages (although these pages are potentially dynamically generated. Think of all content, including the applications, as "pages."

NetObjects Fusion can help you in the Web site design step. Using the "style" function, you can maintain design consistency. Using the "site view" function, you can compose and view the entire Web site layout. Using the "page view," you can construct each page from various components.

The application part of the Web site may consist of form tags and the elements. These pages will be replaced with servlets or JSPs. But these are used as presentation logic of application in tact, wired with business logic.

You can create a simple Web site. When you execute NetObjects Fusion, the "Welcome to NetObjects Fusion" window is shown; see Figure 60.

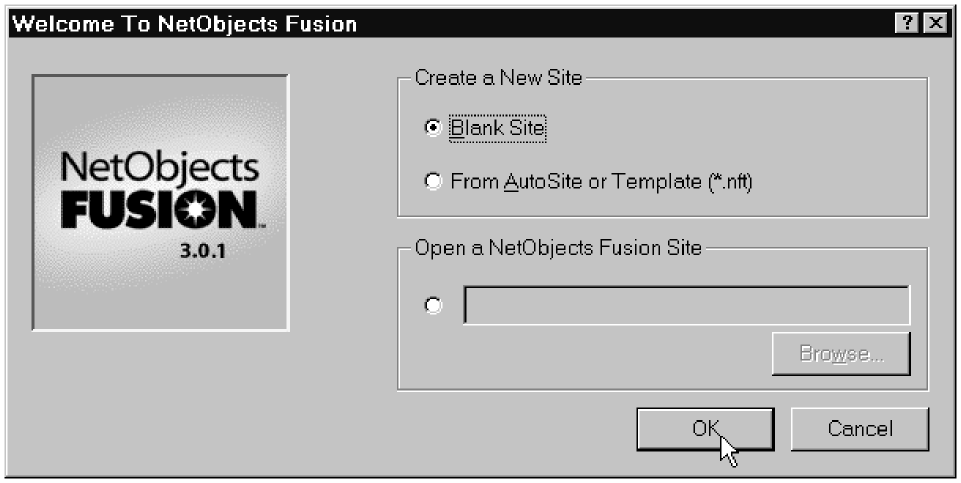
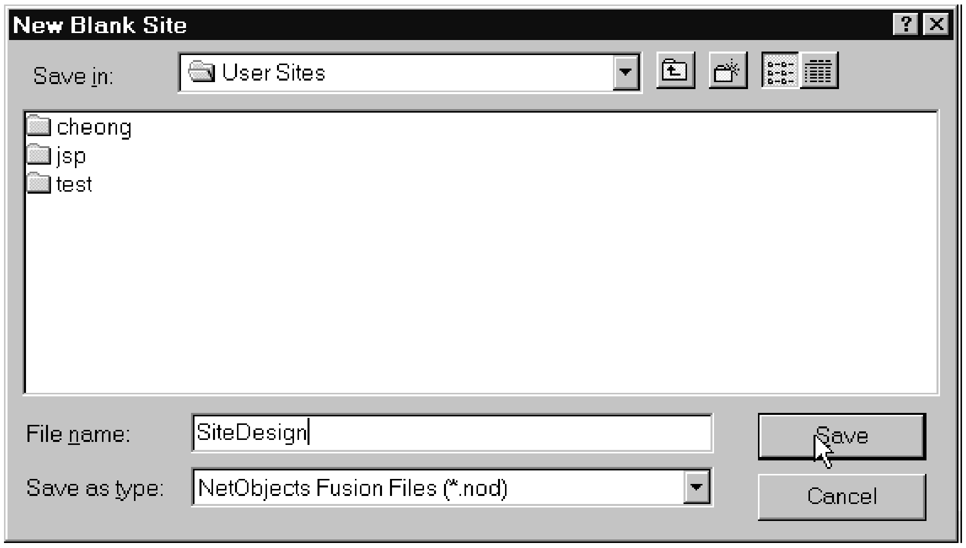

Welcome To NetObjects Fusion
NetObjects FUSION 3.0.1
Create a New Site
Blank Site
From AutoSite or Template (*.nft)
Open a NetObjects Fusion Site
Browse...
OK
Cancel

Figure 60. Welcome to NetObject Fusion Window

Select **Blank site** and press **OK**; the window shown in Figure 61 on page 95 appears. After you enter the appropriate site name, and you are ready to build a new site.


New Blank Site

Save in: User Sites

cheong
jsp
test

File name: SiteDesign

Save as type: NetObjects Fusion Files (*.nod)

Figure 61. New Blank Site of NetObject Fusion

Look at the Site View window as shown in Figure 62. This view shows the pages of the site and the relationship between them (however, a relationship does not necessarily mean a link exists yet).

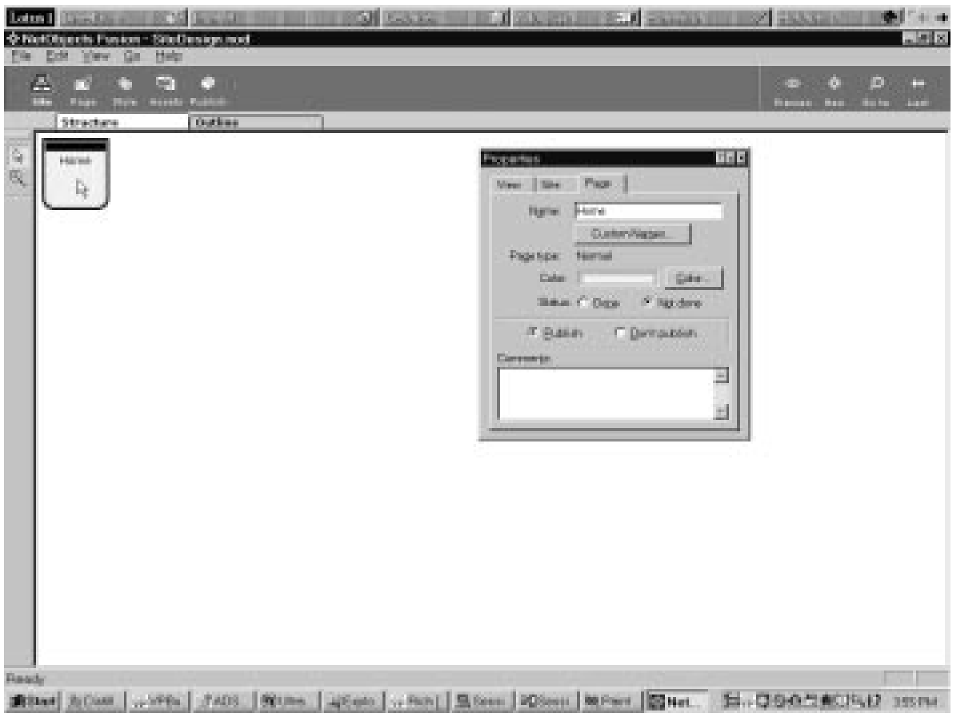


Figure 62. Site View of NetObject Fusion

Now you can set the style of the site. You find the style button at the left side of the cobalt-color toolbar; see Figure 63 on page 96.

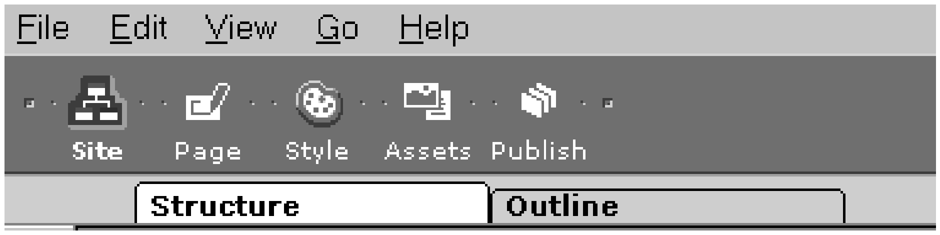


Figure 63. Left Part of the Toolbar in NetObjects Fusion

Press **Style** and we move to Style View, as shown in Figure 64.

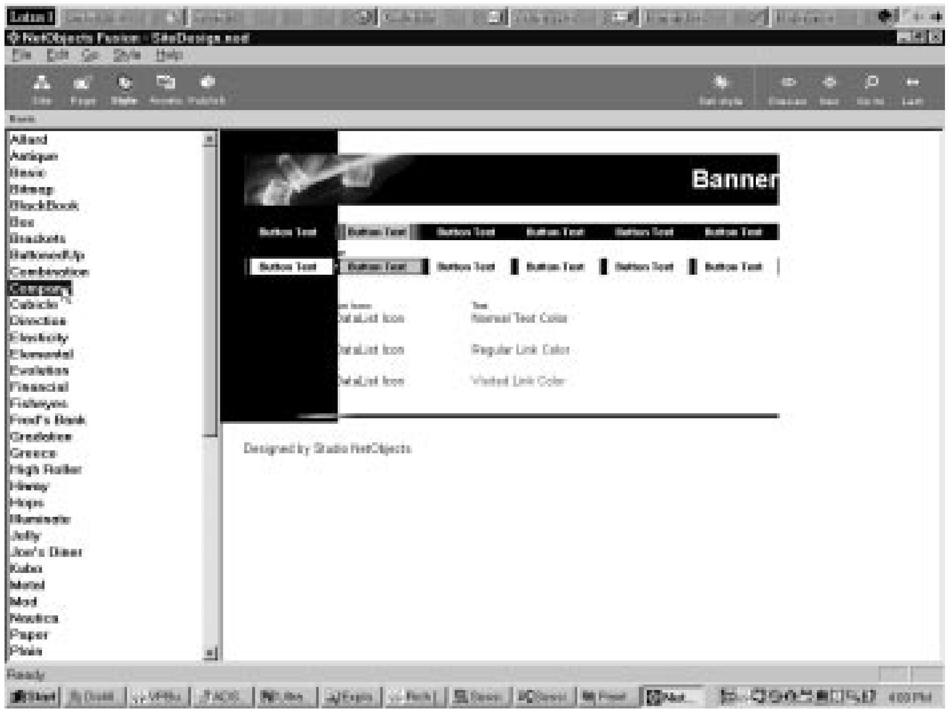


Figure 64. Style View of NetObjects Fusion

After selecting the style you want, press **Set style** at the right side of the
cobalt-color toolbar; see Figure 65. From now on, NetObjects Fusion will maintain

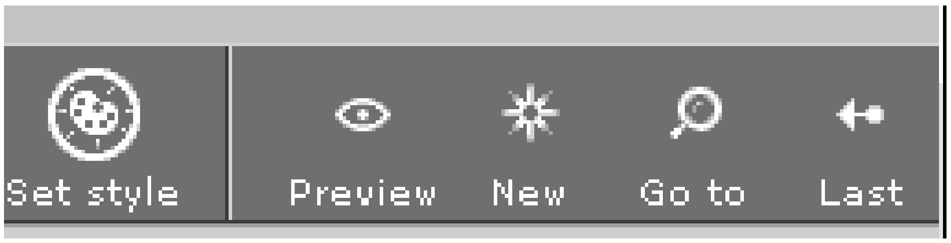


Figure 65. Right Side of Toolbar in NetObjects Fusion

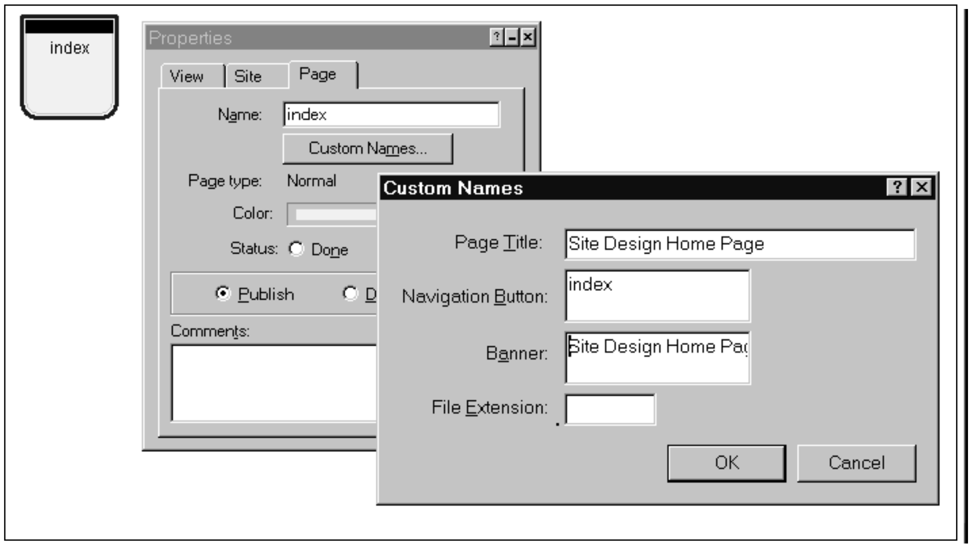
Press **Site** to return to the Site View and change the name of the first page of the site. Press the **page** tab on the properties palette, and enter the name of the page.

The name is the file name excluding the extension when publishing. As an exercise, change this name to “index.” Now, the file name of this page is index.html.

Attention

If you cannot see the properties window, you can turn on the window by selecting **View**, **properties palette** or **View**, **palettes**, **properties palette**.

Again, if you press **Custom**, you can change some other names displayed on the Web page and the file extension; see Figure 66.



Name: index

Page Title: Site Design Home Page

Navigation Button: index

Banner: Site Design Home Pag

File Extension:

Figure 66. Page Properties and the Custom Name Window in NetObjects Fusion

As an exercise, edit the page. Double-click the **page** icon to move to page view for the page. In page view, you can create your own page visually; see Figure 67 on page 98.

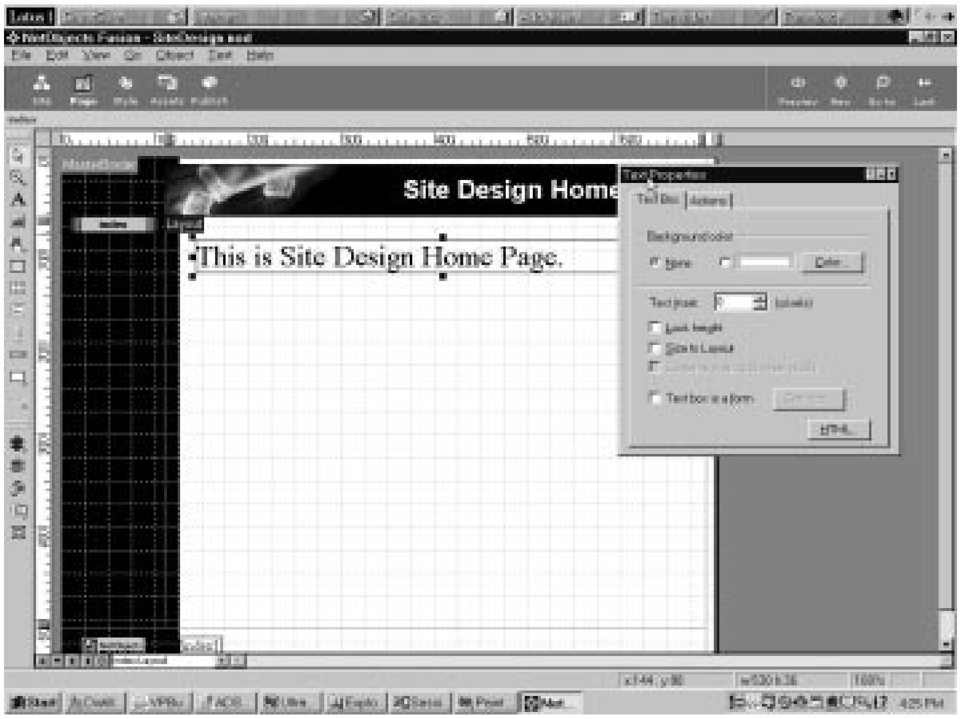


Figure 67. Page View of NetObjects Fusion

Expand the site by right-clicking the **page** icon, and selecting **New page** in the pop-up menu; see Figure 68. In this way, you can construct the tree structure of the site. In addition, you can move a page to another position by a drag-and-drop technique..

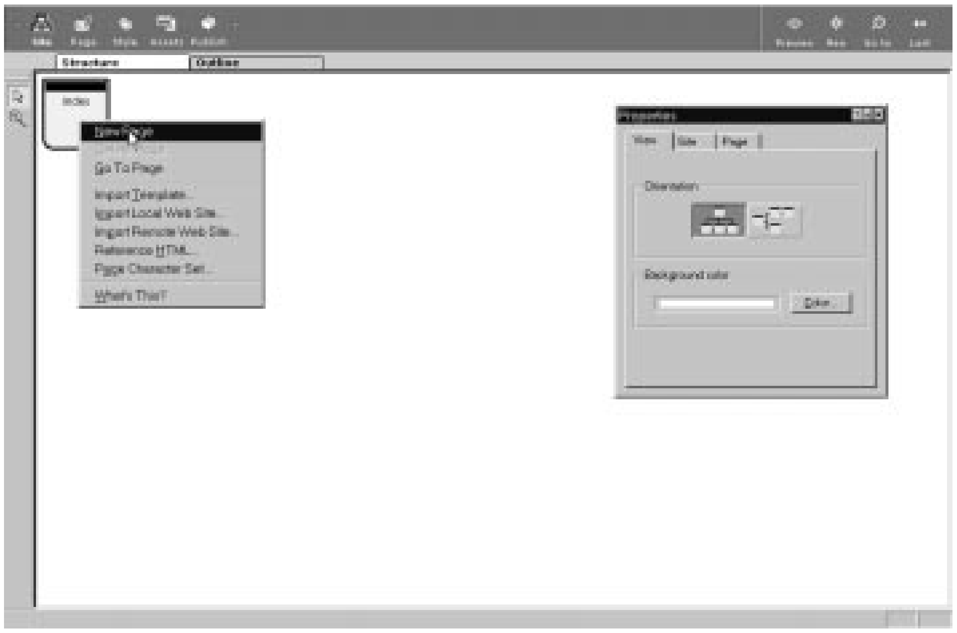


Figure 68. Expansion of Site in NetObjects Fusion

7.2 NetObjects: Static vs. Dynamic Pages

The preceding example describes how you create *static* HTML pages. However, there is another type of HTML page that needs to be addressed: a dynamic Web page. Any HTML page that is composed or modified as a result of an application running or server side include is called a *dynamic* Web page. This book describes several ways to create dynamic Web pages, including using Java servlets and Java server pages.

Chapter 8. DB2 Connectivity

In this chapter we describe the setup of DB2 for the following two situations:

- Connecting to DB2 from a DB2 client

  In this case you would use the Distributed Relational Database Architecture (DRDA) to connect from a Java client application or applet to a DB2 server.

- Connecting to DB2 from a Java servlet

  In this case you need to set up your DB2 environment for the usage of JDBC or SQLJ as the bridge between the Java servlet and the DB2 server on OS/390.

8.1 Configuration of DB2 Connect

In this section we describe how DB2 Connect can be used in combination with a DB2 server installed on OS/390. The usage of DB2 Connect can be twofold:

1. During development of an application

   A connection with the DB2 server on OS/390 can be established from the workstation using DB2 Connect, making it possible to access directly the table descriptions and database layout from your development tool. In this case, you would work with your tables on OS/390 as if they were in your local DB2 database on the workstation.

2. During runtime of an application

   A client can communicate with a DB2 server on OS/390 using DB2 Connect. In this case, you are only using DB2 features and you will not need anything else to communicate, such as a Webserver. Your client can be a Java client using the JDBC API.

8.1.1 Introduction and Comments on DB2 Connect

While most critical data of the world's largest organizations is stored on mainframe environments such as OS/390, there is a steadily growing number of applications running on PCs, UNIX and Apple workstations. The demand for integration of both platforms is continuously increasing.

As this redbook describes, there are many possible ways to accomplish this integration. One way is by using the DB2 Connect either the Enterprise edition or the Personal edition. Both editions are also known as DDCS Multi-User Gateway. DB2 Connect provides a managed gateway for remote clients to access databases stored on one of the following systems:

- DRDA server: DB2 for MVS/ESA, DB2 for AS/400, DB2 for OS/390, DB2 for VSE and VM systems
- DB2 Universal Database servers running on OS/2, Windows NT, AIX and some other UNIX systems

The DB2 Connect products implement a DRDA Application Requester that can access DRDA Application Servers running on MVS, AS/400, OS/390, VM and VSE systems. Additionally DB2 Connect provides a run-time environment for database applications written in Java and other programming languages. Java Database

Connectivity (JDBC) and Microsoft's Open Database Connectivity (ODBC) are part of the product.

DB2 Connect products provide several ways of connecting to OS/390 database servers. One of the following network configurations could be used:

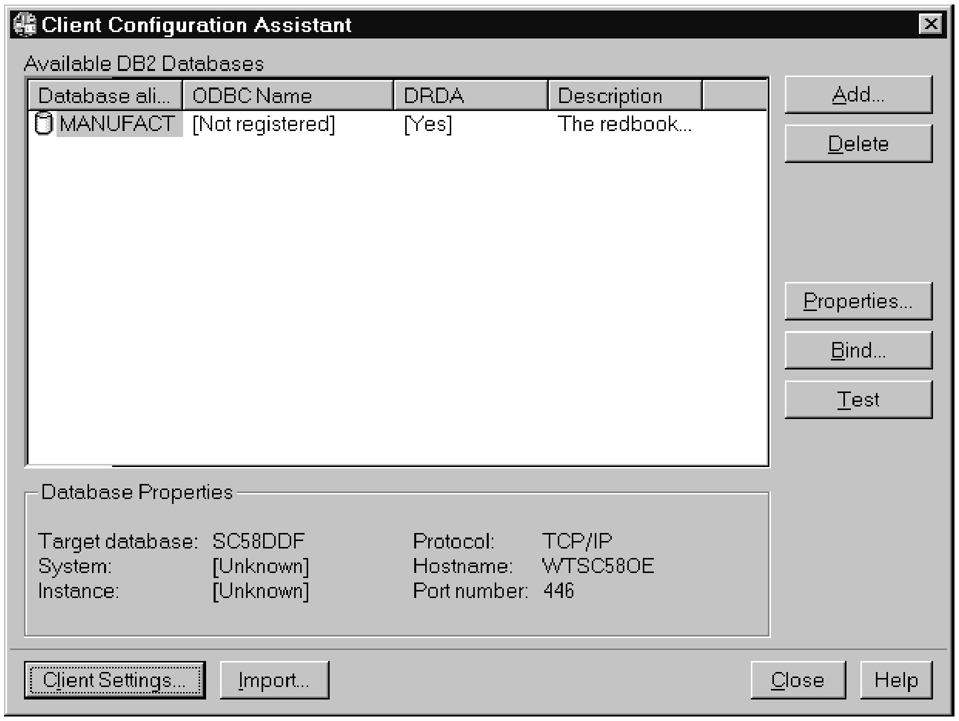
- Direct SNA or APPC connection from DB2 Connect to a DB2 DRDA host
- Direct TCP/IP connection from DB2 Connect to a DB2 DRDA host
- Indirect connection from DB2 Connect to a DRDA host via a communications gateway

TCP/IP is core of the Java programming language. The direct TCP/IP connection to the DRDA host is very convenient and desirable. All the following explanations are based on a TCP/IP connection to the host.

8.1.2 Installing and Configuring the Client

For setting up a DB2 Connect connection to the mainframe, you must have TCP/IP installed on both platforms, client and server, and running properly. On top you can now install DB2 Connect. This is very straightforward. It does not matter whether you install the Personal edition or the Enterprise edition.

Having the product successfully installed, the installation asks for a reboot. To start the client configuration, you must start the Client Configuration program. Depending on the installation, this is done automatically. If this is the *first* start of the configuration, it will welcome you and ask you about adding a database. Otherwise you will see a screen similar to Figure 69:



Database ali...	ODBC Name	DRDA	Description
MANUFACT	[Not registered]	[Yes]	The redbook...

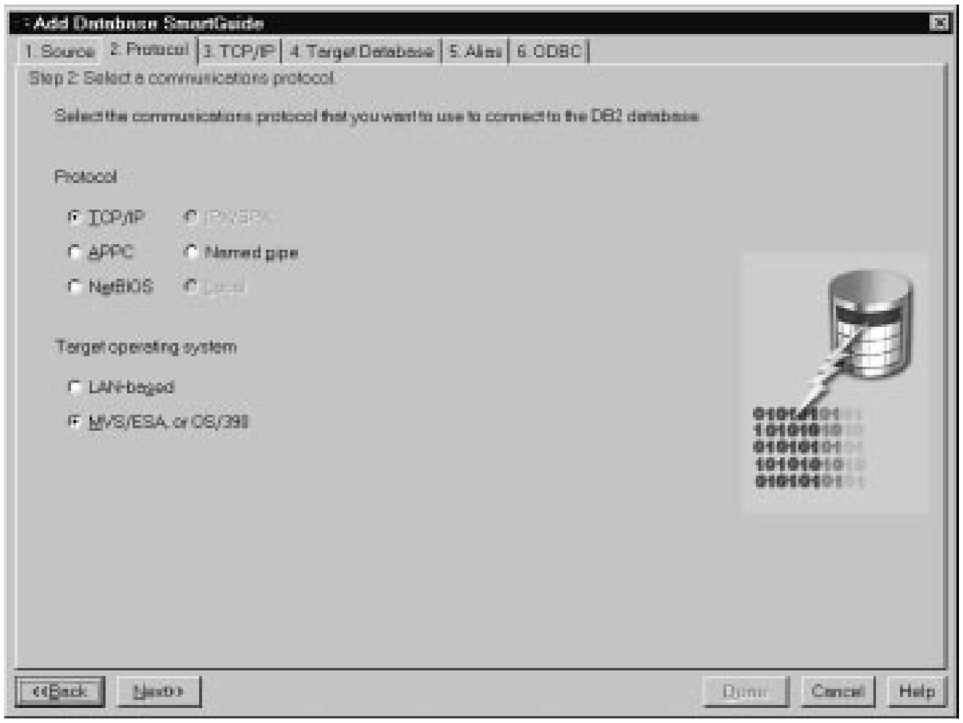
Target database:	SC58DDF
System:	[Unknown]
Instance:	[Unknown]
Protocol:	TCP/IP
Hostname:	WTSC58OE
Port number:	446

Figure 69. The Client Configuration Assistant

Every database you want to access via DB2 Connect must be added by using the Client Configuration Assistant. You do this by pressing the **Add** button. You can choose among three options:

- You can use a profile for configuration.
- You can search the network for the desired computer and the database. (This does not work if you are configuring a database located on a mainframe.)
- You can enter the information manually. (This is the way shown in this example.)

Choose the third option and press **Next**. Since we decided to use the TCP/IP connection, enter the information as shown in Figure 70. Having completed this, go to the next tab.



Step 2: Select a communications protocol.

Select the communications protocol that you want to use to connect to the DB2 database.

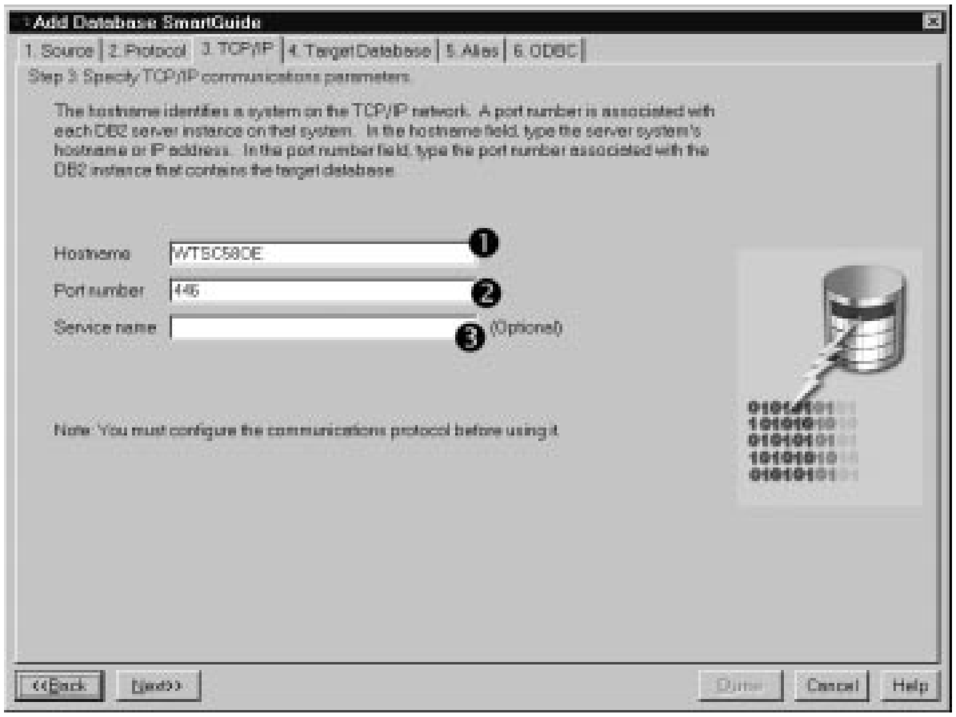
Protocol

[✓ TCP/IP] IPX/SPX APPC Named pipe NetBIOS Local

Target operating system

LAN-based [✓ MVS/ESA, or OS/390]

Figure 70. Select a Communication Protocol



Step 3: Specify TCP/IP communications parameters

The hostname identifies a system on the TCP/IP network. A port number is associated with each DB2 server instance on that system. In the hostname field, type the server system's hostname or IP address. In the port number field, type the port number associated with the DB2 instance that contains the target database.

Hostname: WTSC58OE ❶
Port number: 446 ❷
Service name: ❸ (Optional)

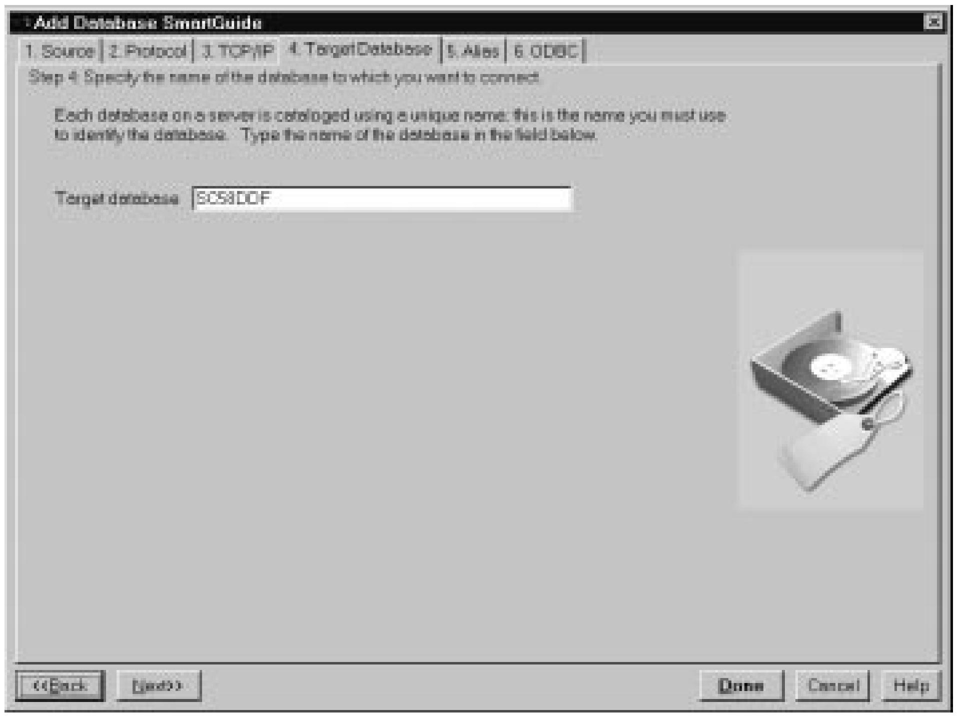
Note: You must configure the communications protocol before using it.

Figure 71. Specify the TCP/IP Parameters

Notes: Figure 71 needs to be filled out as follows:

> **1** Type in the TCP/IP name or the dotted ASCII IP address of the server DB2 is running on.
>
> **2** Type in the port number on which the DB2 instance that contains the target database is listening. It is defined in the TCP/IP services file on the server.
>
> **3** If you want an entry made in the TCP/IP services file on your system for the database server, you can optionally enter a service name.

Go to the next tab (shown in Figure 72 on page 105) and enter the name of the database you want to connect to:



Step 4: Specify the name of the database to which you want to connect.

Each database on a server is cataloged using a unique name; this is the name you must use to identify the database. Type the name of the database in the field below.

Target database: SC58DDF

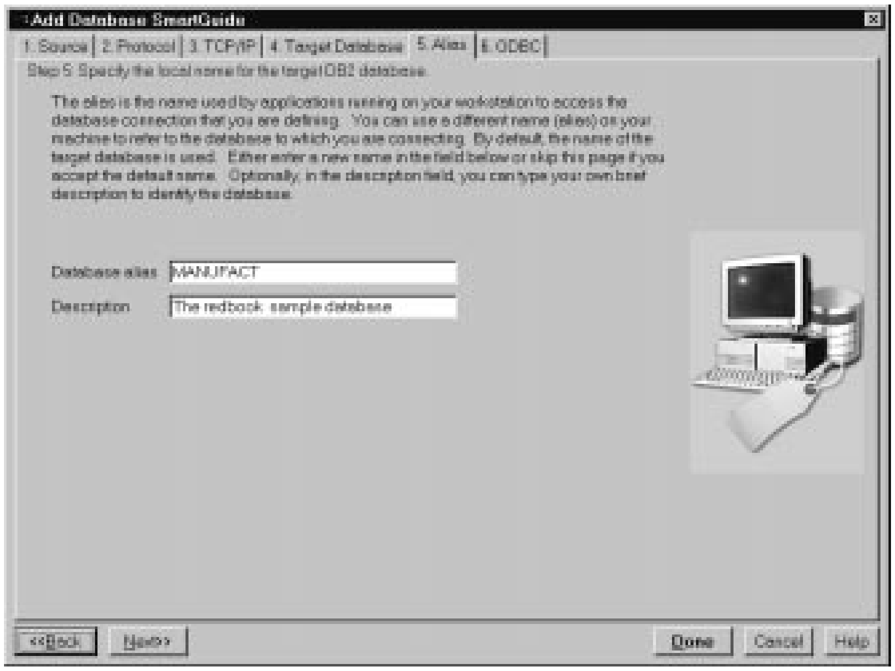
Figure 72. Specify the Name of the Database

Important

The name of the database depends on the system you want to connect to:

- The “name” or alias of the database, if you connect to DB2 Universal Database (UDB) or DB2 For Common Servers
- The name of the “location,” if you connect to DB2 on OS/390 (MVS)
- The “RDB” name, if you are connecting to DB2 for AS/400
- The “DBNAME,” if your DB2 is on VM/VSE

Go to the next page and specify the database alias. Enter a database description. The alias, by default, is the same as the target database of the previous tab. Keep in mind that the *alias is the name you use in your application*. It is limited to eight characters, as shown in Figure 73 on page 106.



Step 5: Specify the local name for the target DB2 database.

The alias is the name used by applications running on your workstation to access the database connection that you are defining. You can use a different name (alias) on your machine to refer to the database to which you are connecting. By default, the name of the target database is used. Either enter a new name in the field below or skip this page if you accept the default name. Optionally, in the description field, you can type your own brief description to identify the database.

Database alias: MANUFACT

Description: The redbook sample database

Figure 73. Specify the Local Name of the Target Database

If there is no JDBC Driver available, then you can register the database you are defining as an ODBC-source on the current computer. However, DB2 has JDBC support. No ODBC data source has to be defined. By default the Smart Guide will define a ODBC source. Just uncheck the checkbox if you do not want the ODBC source to be defined.

8.1.3 The Architectural Possibilities

While testing the possible access method to the data on OS/390, DB2 Connect is very easy to use. Its installation is straightforward, the configuration quick and the access to the database is fast.

There are basically two ways for a Java program to use DB2 Connect:

8.1.3.1 Setting Up the Fat Client Architecture

In this approach, the Java program is a Java application. All class files exist on the local hard disk of each client workstation. Also, DB2 Connect is installed on each workstation. The workstation is part of the local area network (LAN) and TCP/IP is installed. Figure 74 on page 107 illustrates the architecture:

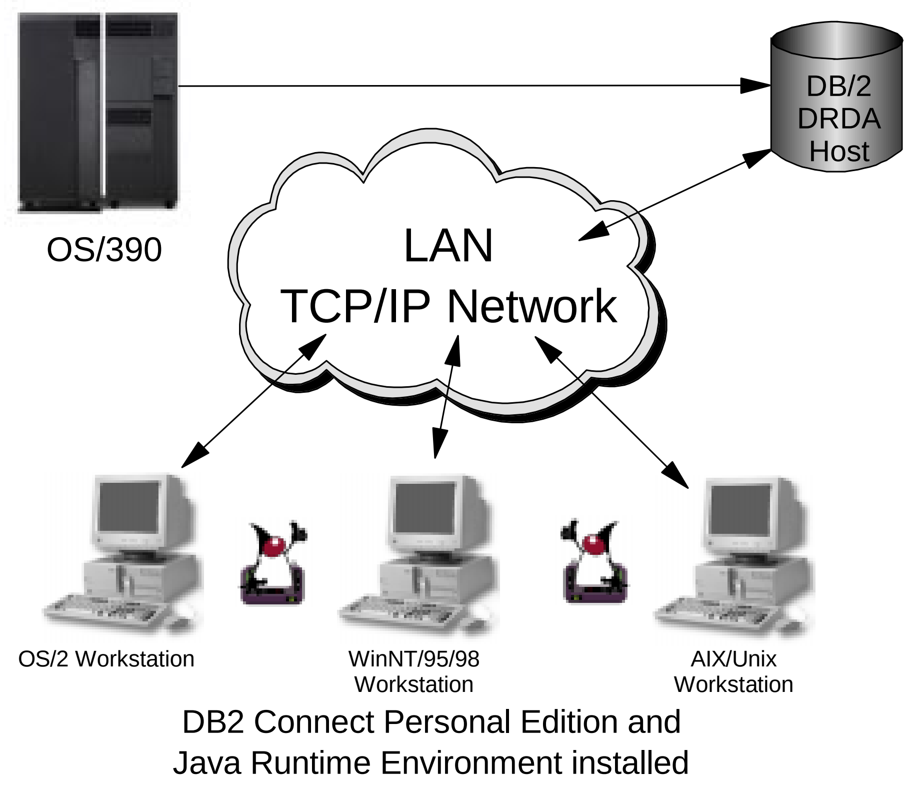


Figure 74. Two-Tier Client/Server Configuration (Fat Client)

When using a two-tier solution, you do not have to worry about Java security issues. You need to provide the application with three parameters:

- The JDBC driver name for DB2: `COM.ibm.db2.jdbc.`**`app`**`.DB2Driver`
- The URL of the database: `jdbc:db2:<DatabaseName>`
- The name of the database table

8.1.3.2 Setting Up the Thin Client Architecture

The setup for the three-tier solution is totally different from that of the two-tier. Keep in mind that this configuration does not refer to a Webserver on OS/390, but rather to a Webserver on a workstation connected to the OS/390 DB2.

In this approach, you first have to install and set up a Webserver and the Web page which includes the desired applet.

Beyond that, applets being loaded remotely are part of the Java security model, the so-called “sandbox.” That means that every applet being loaded from a remote computer is treated as untrusted.

Depending on the browser you are using, the security restrictions implemented in the Java Virtual Machine (JVM) of your browser do not allow you to establish a connection to a database that resides on a computer other than the Webserver from which the applet has been loaded.

However, this problem can be solved, since the browser treats a digitally signed applet as trusted. For that reason you must sign the applet and provide the client browser with a so-called “Certificate.”

Figure 75 illustrates the three-tier architecture:

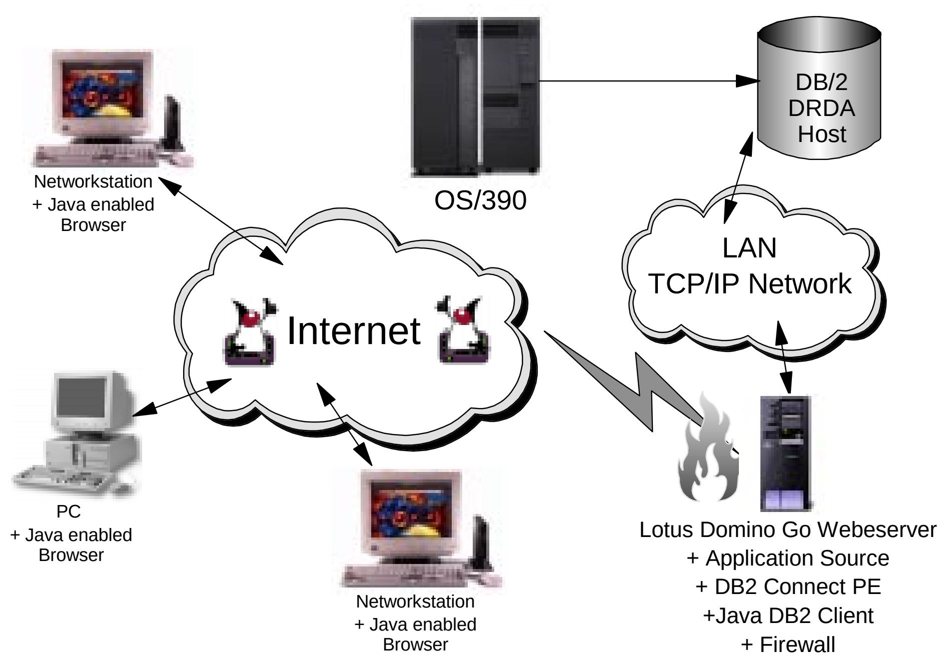


Figure 75. Three-Tier Client/Server Configuration (Thin Client)

Since this is a three-tier environment, and the physical JDBC driver is not installed on the client machine, the setup is totally different.

Assuming that the Webserver is installed and configured and that the HTML pages for loading the applet are accessible, the following software must be installed and set up correctly:

- DB2 Connect is installed and the database has been added as described in 8.1.2, “Installing and Configuring the Client” on page 102.
- The DB2 JDBC Server (db2jstrt) is running and listening on a specified unused port.

  Note: At the command line you type: `db2jstrt ####`, where #### is the free port number.
- The JDBC Driver classes are accessible through the CLASSPATH environment variable.
- The applet is packed into a Java Archive (JAR) file. The jar file has been signed and is accessible through the CLASSPATH environment variable.

Setting up the three-tier solution, you must provide the applet with three parameters:

- The JDBC driver: `COM.ibm.db2.jdbc.net.DB2Driver`

- The URL: `jdbc:db2://YourComputer:####/TheDatabase`, where `YourComputer` is the IP address or the domain name of the actual workstation and `####` is the portnumber that the DB2 JDBC server is listening on (`db2jstrt`)
- The correct name of the table

The Web client must have a Java-enabled browser.

Note: There might be security problems because of the different Java implementations on the different browsers and browser versions. We tested the sample application with Netscape Communicator version 4.5 and Microsoft Internet Explorer version 4.0

8.2 Configuring the Webserver to Use DB2

This section discusses the configuration changes needed to the Webserver in order to support DB2 access for Java applications running on Webserver.

8.2.1 Overview

In order for the Webserver and its Java applications to have access to DB2, you must first enable JDBC and/or SQLJ support on OS/390. In order to support JDBC or SQLJ on OS/390, you will need to install DB2 for OS/390 version 5.1 or above. In addition, you will need to install DB2 for OS/390 JDBC and SQLJ Driver support and DB2 for OS/390 Call Level Interface (CLI) support. Refer to *Program Directory for DB2 for OS/390*, GI10-6973-02, for those components.

You may refer to the *DB2 for OS/390 V5 Application Programming Guide and Reference for Java*, SC26-9547-00. URL:

```
http://www.ibm.com/software/data/db2/os390/sqlj.html
```

Additional information can be found in *DB2 for OS/390 V5 Call Level Interface Guide and Reference*, SC26-8959.

Once all of the required DB2 components are installed, we recommend you run the sample JDBC program which is provided in the install instructions for JDBC.

8.2.2 Lotus Domino Go Webserver Release 5.0 DB2 Support

The following steps need to be followed in order to enable the OS/390 Webserver for JDBC and SQLJ ACCESS. We assume that you are running Lotus Domino Go Webserver Release 5.0 with either ServletExpress or WebSphere Application Server for OS/390 V1.1.

1. In the Webserver startup proc, you will need to add the following STEPLIBS:

   ```
   //STEPLIB  DD DSN=IMW.SIMWMOD1,DISP=SHR
   //         DD DSN=DB2V510.SDSNLOAD,DISP=SHR
   //         DD DSN=DB2V510.SDSNEXIT,DISP=SHR
   ```

2. For JDBC support, you will need to add a DD card containing the CLI initialization dataset to the Webserver startup PROC; following is an example. In your case, you may have another data set naming convention, in which case you need to use your own specific CLI INI file.

   ```
   //DSNAOINI DD DSN=DB2V510U.DB2CLI.CLIINI,DISP=SHR
   ```

The CLI INI file is used by JDBC to determine the subsystem name and some other information.

3. You will need to update the jvm.properties as follows:
   - For `ncf.jvm.classpath` you must add:

     ```
     /usr/lpp/db2/db2510/classes/db2jdbcclasses.zip
     ```

     and

     ```
     /usr/lpp/db2/db2510/classes/db2sqljruntime.zip
     ```

   - For `ncf.jvm.libpath` you must add:

     ```
     /usr/lpp/db2/db2510/lib
     ```

4. The following is an example CLI INI we used for our examples.

```
[COMMON]
; DB51 is our DB2 SSID
MVSDEFAULTSSID=DB51
; TRACE is for IBM DEBUGGING
; CLITRACE is for application debugging
;be sure trace is not started before the ini is read, else this is
;  missed
;be sure that you put this ini out just before the job you want to
; trace, that is, if you have a setup insert job. run that before.
; turn diagnosis trace on and increase trace buffer size
TRACE=0
TRACE_NO_WRAP=1
TRACE_BUFFER_SIZE=2000000
; turn user application trace on and direct to DD name CLITRACE
CLITRACE=0
TRACEFILENAME=DD:CLITRACE

; Example SUBSYSTEM stanza for DSGC subsystem
[DB51]
; the MVSATTACHTYPE can be either CAF or RRSAF
MVSATTACHTYPE=RRSAF
PLANNAME=DSNACLI
; SC58DDF is our DB2 location name.
[SC58DDF]
AUTOCOMMIT=1
CONNECTTYPE=1
```

8.2.3 Installation of RRS

As already explained briefly in 3.3.1.3, “Resource Management” on page 24, RRSAF is a call attach option to connect to DB2. In our testing of DB2 servlets, we elected to use RRSAF. Information about RRS can be found in *OS/390 V2R4.0 MVS Programming: Resource Recovery*, GC28-1739. We followed the next steps to implement RRS:

1. We allocated Sysplex LOGR using the following JCL:

```
//DEFLOGR   JOB (999,POK),'DEFINE LOGR CDS',CLASS=A,REGION=4M,
//             MSGCLASS=T,TIME=10,MSGLEVEL=(1,1),NOTIFY=&SYSUID
//STEP1    EXEC PGM=IXCL1DSU
//SYSPRINT DD   SYSOUT=*
//SYSIN    DD   *
     DEFINEDS SYSPLEX(PLEX58)
              MAXSYSTEM(2)
```

```
                  DSN(SYS1.PLEX58.LOGR00) VOLSER(TARPLX)
                  CATALOG
              DATA TYPE(LOGR)
                   ITEM NAME(LSR) NUMBER(100)
                   ITEM NAME(LSTRR) NUMBER(60)
                   ITEM NAME(DSEXTENT) NUMBER(10)
    /*
```

2. We updated the COUPLExx member to add the LOGR:

```
COUPLE SYSPLEX(PLEX58)
       PCOUPLE(SYS1.PLEX58.CDS02)
DATA   TYPE(LOGR)
       PCOUPLE(SYS1.PLEX58.LOGR00)
DATA   TYPE(WLM)
       PCOUPLE(SYS1.PLEX58.WLM04)
```

3. We defined LOGR policy with the following JCL:

```
//DEFLOGRP  JOB (999,POK),'LOGR POLICY',CLASS=A,REGION=4M,
//             MSGCLASS=X,TIME=10,MSGLEVEL=(1,1),NOTIFY=&SYSUID
//STEP1    EXEC PGM=IXCMIAPU
//SYSPRINT DD   SYSOUT=*
//SYSABEND DD   SYSOUT=*
//SYSIN    DD   *
     DATA TYPE(LOGR) REPORT(YES)

     DEFINE LOGSTREAM
     NAME(ATR.PLEX58.ARCHIVE)
     DASDONLY(YES)
     HLQ(LOGR) MODEL(NO)
     LS_SIZE(1024)
     STG_SIZE(1024)
     LOWOFFLOAD(0) HIGHOFFLOAD(80)
     RETPD(15) AUTODELETE(YES)

     DEFINE LOGSTREAM
     NAME(ATR.PLEX58.RM.DATA)
     DASDONLY(YES)
     HLQ(LOGR) MODEL(NO)
     LS_SIZE(1024)
     STG_SIZE(1024)
     LOWOFFLOAD(0) HIGHOFFLOAD(80)
     RETPD(15) AUTODELETE(YES)

     DEFINE LOGSTREAM
     NAME(ATR.PLEX58.MAIN.UR)
     DASDONLY(YES)
     HLQ(LOGR) MODEL(NO)
     LS_SIZE(1024)
     STG_SIZE(1024)
     LOWOFFLOAD(0) HIGHOFFLOAD(80)
     RETPD(15) AUTODELETE(YES)

     DEFINE LOGSTREAM
     NAME(ATR.PLEX58.DELAYED.UR)
     DASDONLY(YES)
     HLQ(LOGR) MODEL(NO)
     LS_SIZE(1024)
     STG_SIZE(1024)
```

```
LOWOFFLOAD(0) HIGHOFFLOAD(80)
RETPD(15) AUTODELETE(YES)

DEFINE LOGSTREAM
NAME(ATR.PLEX58.RESTART)
DASDONLY(YES)
HLQ(LOGR) MODEL(NO)
LS_SIZE(1024)
STG_SIZE(1024)
LOWOFFLOAD(0) HIGHOFFLOAD(80)
RETPD(15) AUTODELETE(YES)
```

4. We updated the IEFSSNxx member to add RRS, as follows:

   ```
   SUBSYS SUBNAME(RRS)         /* RRS         */
   ```

5. We copied SYS1.SAMPLIB(ATRRRS) to SYS1.PROCLIB(RRS)

To run RRS, simply perform an `S RRS` command. We suggest adding the startup for RRS into the COMMNDxx member in SYS1.PARMLIB. To take RRS down, you will need to issue `SETRRS CANCEL` command.

8.3 Configuring JDBC and SQLJ in OS/390 UNIX Shell

In order to run JDBC or SQLJ programs directly from the OS/390 UNIX shell or use the SQLJ translator code, you must configure your profile after JDBC and SQLJ code has been installed. The following steps outline the environmental variables in the profile that needs to be set:

1. Modify STEPLIB to include the SDSNEXIT and SDSNLOAD data sets:

   ```
   export STEPLIB=DSN510.SDSNEXIT:DSN510.SDSNLOAD
   ```

2. Modify LIBPATH and LD_LIBRARY_PATH to include the DLL library for DB2:

   ```
   export LIBPATH=/usr:/usr/lib:/usr/lpp/db2/db2510/lib
   export LD_LIBRARY_PATH=/usr/lpp/db2/db2510/lib
   ```

3. Modify CLASSPATH to include the following library:

   ```
   export CLASSPATH=/usr/lpp/db2/db2510/classes/db2classes.zip:
   export CLASSPATH=/usr/lpp/db2/db2510/classes/db2sqljclasses.zip:
   ```

4. Add DB2SQLJDBRMLIB to point to a MVS partitioned data set into which DBRMs are placed. The default is prefix.DBRMLIB.DATA.

   ```
   export DB2SQLJDBRMLIB=USER.DBRMLIB.DATA
   ```

5. Add DB2SQLJPLANNAME to point to the name of the plan that is associated with an SQLJ application. The plan is created by DB2 for OS/390 bind process. This is only for SQLJ. (JDBC uses CLI parms.)

   ```
   export DB2SQLJPLANNAME=SQLJPLAN
   ```

6. Add DB2SQLJSSID to point to the name of the DB2 subsystem to which an SQLJ application connects. This is only for SQLJ. (JDBC uses CLI parms.)

   ```
   export DB2SQLJSSID=DB51
   ```

7. Add DB2SQLJATTACHTYPE to specify the attachment facility that an SQLJ application program uses to connect to DB2. The value can be CAF or RRSAF. This is only for SQLJ. (JDBC uses CLI parms.)

   ```
   export DB2SQLJATTACHTYPE=RRSAF
   ```

8.4 Possible Pitfalls When Using DB2 from Java

We encountered the following potential causes for problems when configuring the JDBC environment for DB2 on OS/390:

- No DB2 location name has been defined.

  You need to have a DB2 location name in order to use JDBC. The location name is usually obtained by configuring DDF. Without this location name, you cannot use JDBC.

- CLI members not bound.

  You may need to re-bind the CLI members. Otherwise you might get an error from your JDBC application indicating that something is wrong with your driver.

  You can re-bind the CLI members by running job DSNTIJCL in your DB2 JCL library.

- Insufficient authorization given to use CLI.

  You may need to explicitly set the authorization for using the CLI members. You can do this by executing the following command from SPUFI or from a job:

  ```
  GRANT EXECUTE ON PLAN DSNACLI TO PUBLIC
  ```

- JDBC classes are not in CLASSPATH.

  You need to have the JDBC classes added to your CLASSPATH environment variable, in `/etc/profile` for Java applications and in `jvm.properties` for Java servlets executed in Lotus Domino Go Webserver Release 5.0.

- JDBC native drivers are not in PATH.

  You need to have the JDBC drivers added to your PATH environment variable, in `/etc/profile` for Java applications and in `jvm.properties` for Java servlets executed in Lotus Domino Go Webserver Release 5.0.

Part 3. Develop Application Solutions for OS/390 Using Java

In this part we describe various application solutions using Java and a specific subsystem on OS/390. All solutions make use of a Webserver, Lotus Domino Go Webserver Release 5.0 with either ServletExpress or WebSphere Application Server for OS/390 V1.1. For each subsystem we focus on a different element:

In Chapter 9, “DB2 Access” on page 117 we focus on programming servlets using either JDBC or SQLJ to access DB2.

In Chapter 10, “Develop Java Solutions for CICS on OS/390” on page 155 our starting point was a Windows NT workstation with a development environment including a CICS server (TxSeries) and a Webserver. We developed an application entirely on Windows NT, ran it there, and then moved the application up to OS/390.

In Chapter 11, “Accessing IMS Transactions from the Web” on page 207 we focus on a more complicated scenario to access IMS: using the session concept.
Also, we developed a solution using the MQSeries Bindings for Java on OS/390 to access IMS from a Java servlet or JSP.

Chapter 9. DB2 Access

In this chapter we make a technical comparison between two methods for accessing DB2 on OS/390 from Java: Java Database Connectivity (JDBC) and SQLJ.

At the time of writing, JDBC support for DB2 on OS/390 has been available for almost a year and SQLJ has just been made available.

9.1, “JDBC Implementation for DB2 on OS/390” gives you details of the JDBC support on OS/390 along with some sample Java code. 9.2, “SQLJ Implementation for DB2 on OS/390” on page 128 explains how to code SQLJ statements, and 9.2.13, “Steps in the SQLJ Program Preparation Process” on page 149 outlines the procedure to prepare your program and make it ready for use.

9.1 JDBC Implementation for DB2 on OS/390

DB2 for OS/390 supports the JDBC specification. You can download the specification from the JDBC Website at URL:

```
http://splash.javasoft.com/jdbc
```

You should familiarize yourself with the specification to understand how to use the JDBC APIs. Documentation that includes detailed information about each of the JDBC API interfaces, classes, and exceptions is also available at this Website.

DB2 for OS/390 requires the JDK for OS/390 (version 1.1.1 or higher). The contents of the JDK include a Java compiler, a Java Virtual Machine (JVM), and a Java Debugger. You can find out more about the JDK from the Java for OS/390 Website at URL:

```
http://www.ibm.com/s390/java
```

A Java application executes under the JVM. The Java application first loads the JDBC driver, in this case the DB2 for OS/390 JDBC driver, and subsequently connects to a DRDA server (by invoking the `DriverManager.getConnection` method).

The Java application identifies the target data source it wants to connect to by passing a database Uniform Resource Locator (URL) to the `DriverManager`.

The basic structure for the URL is:

```
jdbc:<subprotocol>:<subname>
```

The URL values for a DB2 for OS/390 data source are specified as follows:

```
jdbc:db2os390:<location>
```

The location value is the DB2 LOCATION name defined in the DB2 catalog table, SYSIBM.LOCATIONS.

When the application attempts a connection to a data source, it requests a `java.sql.Connection` implementation from the `DriverManager` (part of the `java.sql` package). The `DriverManager` searches all of the known `java.sql.Driver` implementations for a driver that is capable of accepting the database URL. It then invokes the first JDBC driver that supports the subprotocol specified in the URL and is registered with the `DriverManager`.

In this case, the DB2 for OS/390 driver (which is registered with the `DriverManager`) accepts the URL, and returns a `java.sql.Connection` implementation that represents the database connection.

The DB2 for OS/390 JDBC driver is implemented as a “Type 1” driver. It provides Java application support only. The Type 1 driver is one of four types of JDBC drivers. It is implemented as a JDBC-ODBC bridge which enables the use of existing JDBC drivers in Java programs. It translates all of the JDBC method calls into ODBC function calls which process database requests.

Included with the driver is the `ibm.sql` package. This package is the DB2 for OS/390 implementation of the `java.sql` JDBC API package. The package includes all of the JDBC classes, interfaces, and exceptions.

The JDBC API consists of the abstract Java interfaces that an application program uses to access databases, execute SQL statements, and process the results. There are five main interfaces that perform these functions:

1. The `DriverManager` class loads drivers and creates database connections.
2. The `connection` interface handles the connection to a specific database.
3. The `statement` interface handles the SQL statements on a connection.

   This interface has two underlying interfaces:

   - The `PreparedStatement` interface supports any SQL statement containing input parameter markers.
   - The `CallableStatement` interface supports the invocation of a stored procedure and allows the application to retrieve output parameters.
4. The `ResultSet` interface provides access to the results that an executed statement generates.

Like ODBC, JDBC is a dynamic SQL interface. Writing a JDBC application is similar to writing a C application using ODBC to access a database. When you create a Java application that uses the JDBC interfaces, you import the `java.sql` package and invoke methods according to the JDBC specification.

To help you begin coding your program, the following sample servlet will display the SYSIBM.SYSTABLES table:

```
//*****************************************************************
//* sample01.java
//*
//* This simple sample will demonstrate servlet using JDBC to
//* DB2 for OS390.
//*****************************************************************
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
public
class sample01 extends HttpServlet {
   static {
   // * ### 1 ###
   // *
   // * You must register the DB2 for OS390 driver before attempting
   // * to connect. This next statement will load the driver and
   // * creates an instance of the class.

      try {
         Class.forName("ibm.sql.DB2Driver");
      }
      catch (ClassNotFoundException e) {
         e.printStackTrace();
      }
   }
    public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws  ServletException, IOException
    {
 Connection con =null ;
 Statement stmt = null;
 PrintWriter  out;
 res.setContentType("text/html");
 out = res.getWriter();
 out.println("<html>");
 out.println("<head><title>Hello DB2 for OS390</title></head>");
 out.println("<body>");
 out.println("<h1>Hello DB2 for OS390</h1>");
 try {
      out.println("<p>**** Lets try JDBC ****</p>");
      // Create the connection
         out.println("<p> Lets trying to connect to db2.</p>");
   // **************************************************************
   // *
   // * ### 2 ###
   // *
   // * In order to for JDBC to connect to DB2, you will need to     *
   // * know the following:                                          *
   // * DbLocationName will be the DB2 Location name. This can be
   // *                optain from DB2 admin.
   // * Dbsubprotocol  is the JDBC subprotocol. DB2OS390 is the
   // *                sub protocol for OS390 DB2.
   // * jdbcDriver     for OS390 DB2, ibm.sql.DB2Driver is required.
   // * url            is required for a database connection. It
   // *                consist of protocal + subprotocol + subname.
   // *                For our OS390 DB2 example, protocol is "jdbc",
   // *                subprotocol must be "DB2OS390, and subname is
   // *                the DB2 location name.
   // *
```

Figure 76 (Part 1 of 2). Example of a Simple Servlet Accessing DB2 Via JDBC

```
   // * jdbc also allows for user/password option. When running a
   // * servlet under the Go Domino Web Server, the user/password
   // * is ignored. The Go Domino Web Server will pass user to DB2.
   // * Go Dominto Web Server will pass either Surrogate ID, Client
   // * ID, or Server ID depending on the configuration options used.
   // *
         String url = "jdbc:db2os390:SC58DDF";
   // * ### 3 ###
   // *
   // * Ready to estabish connection to DB2 for OS390 URL.
   // * Note: The url is defined in the Global Static definitions above.
         con = DriverManager.getConnection (url);
         out.println("<p>**** We are connected to DB2 for OS/390.</p>");
   // * ### 4 ###
   // *
   // * The statement interface needs to be established in order
   // * to execute SQL statements. SQL statements will be executed
   // * as part of the doManUpdateScreen class.
        stmt = con.createStatement();
        out.println("<p>**** Statement Created.</P>");
   // * ### 5 ###
   // *
   // * Lets do some DB2 queries.
   // * Execute a Query and generate a ResultSet instance

        ResultSet rs = stmt.executeQuery("SELECT * FROM SYSIBM.SYSTABLES");
        out.println("<p> Query now successful. Here is Results:</p>");
   // * Print all of the table names to sysout
        while (rs.next()) {
          String s = rs.getString(1);
          out.println(" NAME = " + s);
        }
        out.println("<p> Result completed</p>");
   // * Close the statement
        stmt.close();
        out.println("<p> Statement Closed </p>");
   // * Close the connection
        con.close();
        out.println("<p>Now we are Disconnect from DB2 for OS/390.</p>"
      }
      catch( SQLException e ) {
      out.println("Error:");
      out.println(e.toString());
      }
      catch( Exception e ) {
      out.println("Error:");
      out.println(e.toString());
      }
      out.println("</body></html>");
  }
}
```

Figure 76 (Part 2 of 2). Example of a Simple Servlet Accessing DB2 Via JDBC

Our next example demonstrates a JDBC SELECT, INSERT, and UPDATE.

```
// *****************************************************************
// * sample02.java -                                               *
// *
// *  This sample servlet will demonstrate JDBC to OS390 for DB2. It
// *  will perform a SELECT, UPDATE, and INSERT operations.  In
// *  addition, this servlet will pass data between this servlet
// *  and the client.
// *
// *  It will use JDBC 1.0 sun standard
// *
// *  The DB2 connect was done in the doget instead of the
// *  init() since we were using CAF and did not have RRSAF running
// *  until the end of our lab work for this book. RRSAF is
// *  required if you plan on performing the connect in the init()
// *  method.
// *
// *****************************************************************
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;           // You must import to use JDBC
import java.util.*;

public class sample02  extends HttpServlet
{
   // ***************************************************************
   // * Define Global Static Variables to this class
   // * Note: An alternative to defining these variables as Static is
   // * to pass these using properties files.
   // ***************************************************************

   // ***************************************************************
   // *
   // * ### 1 ###
   // *
   // * In order to for JDBC to connect to DB2, you will need to     *
   // * know the following:                                          *
   // * DbLocationName will be the DB2 Location name. This can be
   // *                obtain from DB2 admin.
   // * Dbsubprotocol  is the JDBC subprotocol. DB2OS390 is the
   // *                sub protocol for OS390 DB2.
   // * jdbcDriver     for OS390 DB2, ibm.sql.DB2Driver is required.
   // * mytable        is the DB2 table being used in this program.
   // * url            is required for a database connection. It
   // *                consist of protocal + subprotocol + subname.
   // *                For our OS390 DB2 example, protocol is "jdbc",
   // *                subprotocol must be "DB2OS390, and subname is
   // *                the DB2 location name.
   // *
   // * jdbc also allows for user/password option. When running a
   // * servlet under the Go Domino Web Server, the user/password
   // * is ignored. The Go Domino Web Server will pass user to DB2.
   // * Go Dominto Web Server will pass either Surrogate ID, Client
   // * ID, or Server ID depending on the configuration options used.
   // *
```

Figure 77 (Part 1 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
static String DbLocationName = "SC58DDF";
static String mytable        = "ODONNEL.MANUFACTURER";
static String Dbsubprotocol  = "DB2OS390";
static String jdbcDriver     = "ibm.sql.DB2Driver";
static String url            = "jdbc:" +
                                Dbsubprotocol + ":" +
                                DbLocationName;

// * ### 2 ###
// *
// * You must register the DB2 for OS390 driver before attempting
// * to connect. This next statement will load the driver and
// * creates an instance of the class.

static {
   try {
      Class.forName(jdbcDriver);
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   }
}

// *****************************************************************
// * Initialize servlet when it is first loaded                    *
// *****************************************************************
public void init(ServletConfig config)
throws ServletException
{
   super.init(config);
}

// *****************************************************************
// * Respond to user GET request                                   *
// *****************************************************************
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException,IOException
{
   res.setContentType("text/html");  //Lets setup to do some HTML
   res.setHeader("Pragma", "no-cache");
   res.setHeader("Cache-Control", "no-cache");
   res.setDateHeader("Expires", 0);
   ServletOutputStream out = res.getOutputStream();
   try
   {

// * ### 3 ###
// *
// * Ready to establish connection to DB2 for OS390 URL.
// * Note: The url is defined in the Global Static definitions above.

     Connection con = DriverManager.getConnection(url);
```

Figure 77 (Part 2 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
// * ### 4 ###
// *
// * The statement interface needs to be estabished in order
// * to execute SQL statements. SQL statements will be executed
// * as part of the doManUpdateScreen class.

     Statement stmt = con.createStatement();

     doManUpdateScreen(req,res,stmt);
     stmt.close();
     con.close();  //Release connection
   }
   catch(SQLException e)
    {
     out.println("Error:");
     out.println(e.toString());
    }
     catch(IOException e)
      {
       out.println("Error:");
       out.println(e.toString());
      }
   out.close();
}
// ******************************************************************
// * Respond to user to handle Manufacturer Update Screen          *
// ******************************************************************
public void doManUpdateScreen(HttpServletRequest req,
                              HttpServletResponse res,
                              Statement stmt)
              throws ServletException,IOException
{
 Vector ManufNameList = new Vector();
 ResultSet rs ;
 ServletOutputStream out = res.getOutputStream();
 String button       =  req.getParameter("button");
 String manufacturer  =  req.getParameter("manufacturer");
 String address      =  req.getParameter("address");
 String city         =  req.getParameter("city");
 String state        =  req.getParameter("state");
 String zip          =  req.getParameter("zip");
 String firstname    =  req.getParameter("firstname");
 String lastname     =  req.getParameter("lastname");
 String phoneac      =  req.getParameter("phoneac");
 String phoneex      =  req.getParameter("phoneex");
 String phonenr      =  req.getParameter("phonenr");
 String ext          =  req.getParameter("ext");
 String email        =  req.getParameter("email");
```

Figure 77 (Part 3 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
   //Servlets just hate nulls if referenced.
   //sooo, I elected to remove all nulls at this time.
   if (button == null) button = "NEWSELECT";
   if (manufacturer  == null) manufacturer = "";
   if (address       == null) address      = "";
   if (city          == null) city         = "";
   if (state         == null) state        = "";
   if (zip           == null) zip          = "";
   if (firstname     == null) firstname    = "";
   if (lastname      == null) lastname     = "";
   if (phoneac       == null) phoneac      = "";
   if (phoneex       == null) phoneex      = "";
   if (phonenr       == null) phonenr      = "";
   if (ext           == null) ext          = "";
   if (email         == null) email        = "";

 // * ### 5 ###
 // *
 // * Lets do some DB2 queries.

   try {
   if (button.equals("UPDATE")) {
    String query2 = "update " + mytable +
                   "   set man_address        ='" + address + "', "  +
                   "       man_city           ='" + city    + "', "  +
                   "       man_state          ='" + state   + "', "  +
                   "       man_zip            ='" + zip     + "', "  +
                   "       man_con_last_name  ='" + lastname +"', "  +
                   "       man_con_first_name ='" + firstname+"', "  +
                   "       man_con_ext        ='" + ext     + "', "  +
                   "       man_con_email      ='" + email   + "', "  +
                   "       man_last_upd_uid   ='" + "JAVAID" + "', " +
                   "       man_last_upd_date  = current timestamp "  +
                   "   where man_name = '" + manufacturer + "'" ;
       rs   = stmt.executeQuery(query2); //Exec SQL
   }
   else if (button.equals("SAVE")) {
    String manlist =  req.getParameter("manlist");
    String query3 = "insert into " + mytable      +
                   "      (man_name," +
                   "       man_address," +
                   "       man_city," +
                   "       man_state," +
                   "       man_zip," +
                   "       man_con_last_name," +
                   "       man_con_first_name," +
                   "       man_con_phone_ac," +
                   "       man_con_phone_ex," +
                   "       man_con_phone_nr," +
                   "       man_con_ext," +
                   "       man_con_email," +
                   "       man_last_upd_uid," +
                   "       man_last_upd_date)"  +
```

Figure 77 (Part 4 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
                    "  values ('" + manufacturer + "'," +
                  "'" +  address     + "',  "  +
                  "'" +  city        + "',  "  +
                  "'" +  state       + "',  "  +
                  "'" +  zip         + "',  "  +
                  "'" +  lastname    + "',  "  +
                  "'" +  firstname   + "',  "  +
                  "'" +  phoneac     + "',  "  +
                  "'" +  phoneex     + "',  "  +
                  "'" +  phonenr     + "',  "  +
                  "'" +  ext         + "',  "  +
                  "'" +  email       + "',  "  +
                  "'" +  "JAVAID"    + "',  "  +
                 "   current timestamp)" ;
        rs   = stmt.executeQuery(query3); //Exec SQL
    }
    else if (button.equals("SEARCH")) {
           String manlist =  req.getParameter("manlist");
           String query4  =  "SELECT man_name,"  +
                             "       man_address," +
                             "       man_city,"    +
                             "       man_state,"   +
                             "       man_zip,"     +
                             "       man_con_last_name,"   +
                             "       man_con_first_name,"  +
                             "       man_con_phone_ac,"    +
                             "       man_con_phone_ex,"    +
                             "       man_con_phone_nr,"    +
                             "       man_con_ext,"         +
                             "       man_con_email  "      +
                             "  FROM " + mytable + " "     +
                             "  WHERE man_name = '" + manlist + "'";
        rs  = stmt.executeQuery(query4); //Exec SQL
        rs.next();
            manufacturer   = rs.getString(1);
            address        = rs.getString(2);
            city           = rs.getString(3);
            state          = rs.getString(4);
            zip            = rs.getString(5);
            lastname       = rs.getString(6);
            firstname      = rs.getString(7);
            phoneac        = rs.getString(8);
            phoneex        = rs.getString(9);
            phonenr        = rs.getString(10);
            ext            = rs.getString(11);
            email          = rs.getString(12);
    }
    if ((button.equals("NEWSELECT")) ((  // test buttons be pushed
        (button.equals("UPDATE"))     ((
        (button.equals("ADD"))        ((
        (button.equals("SAVE"))) {
```

Figure 77 (Part 5 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
    manufacturer  = ""  ;  // Clear fields
    address       = ""  ;
    city          = ""  ;
    state         = ""  ;
    zip           = ""  ;
    firstname     = ""  ;
    lastname      = ""  ;
    phoneac       = ""  ;
    phoneex       = ""  ;
    phonenr       = ""  ;
    ext           = ""  ;
    email         = ""  ;
    String query1  = "SELECT man_name " +
                     "  FROM " + mytable +
                     "  ORDER by 1" ;          //setup SQL
    rs   = stmt.executeQuery(query1); //Exec SQL
    while(rs.next())
    {
      ManufNameList.addElement(rs.getString(1));//Save SQL Results
     }
}
}
catch(SQLException e)
  {
    out.println("Error:");
    out.println(e.toString());
  }
try {
out.println("<html>");
out.println("<head>");
out.println("<title>Manufacturer Information</title></head>");
out.print("<BODY TEXT=\"#000000\" BGCOLOR=\"#FFFFFF\" LINK=\"#0000FF
out.print(" VLINK=\"#663366\" ALINK=\"#000088\"");
out.println(" BACKGROUND=\"/redfade.gif\" NOSAVE>");
out.println("<spacer type=block width=100 height=100 align=left>");
out.println("<center><B><font size=+3>Manufacturer Information</font
out.println("<br><br><br>");
out.println("<blockquote>");
out.print("<form name=dman method=get");
out.println(" action=\"/servlet/sample02\">");
out.println("<table cols=2 width=90% ><tr>");
out.println("<td align=right width=100>Manufacturer Name:</td>");
out.println("<td>");
if  ((button.equals("NEWSELECT"))   ((
    (button.equals("UPDATE"))       ((
    (button.equals("SAVE")))
{
   out.println("<select name=manlist>             ");
   if (ManufNameList.isEmpty()) {
     out.println("<option selected> Error");
     out.println("</select>");
     }
   else {
     for (int i = 0; i < ManufNameList.size(); i++) {
        out.println("<option selected> " +
        ManufNameList.elementAt(i));
        }
     out.println("</select>");
   }
```

Figure 77 (Part 6 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
       out.println("&nbsp;&nbsp;");
       out.println("<input type=submit value=SEARCH name=button>");
       out.println("</td></tr>");
       }
    else {
      out.print("<input name=manufacturer size=30");
      out.println(" &nmaxlength=30 value=\"" + manufacturer.trim() + "\"
      out.println("</td></tr>");
    }
    out.println("<tr><td align=right>Address:</td>           ");
    out.print("<td><input name=address size=30               ");
    out.println(" &nmaxlength=30 value=\"" + address.trim() + "\">");
    out.println("</td></tr>                                  ");
    out.println("<tr><td align=right>City:</td>              ");
    out.println("<td>                                        ");
    out.print("<input name=city size=20 maxlength=30         ");
    out.println(" value=\"" + city.trim() + "\">             ");
    out.println("&nbsp;State:&nbsp;                          ");
    out.print("<input name=state size=2 maxlength=2          ");
    out.println(" value=\"" + state.trim() + "\">            ");
    out.println("&nbsp;Zip:&nbsp;                            ");
    out.print("<input name=zip size=10 maxlength=10          ");
    out.println(" value=\"" + zip.trim() + "\">              ");
    out.println("</td></tr>                                  ");
    out.println("<tr>                                        ");
    out.println("<td align=right>Contact First Name:</td>    ");
    out.println("<td>");
    out.print("<input name=firstname size=16 maxlength=30    ");
    out.println(" value=\"" + firstname.trim() + "\">        ");
    out.println("&nbsp;Last Name:&nbsp;                      ");
    out.print("<input name=lastname size=16 maxlength=30     ");
    out.println(" value=\"" + lastname.trim() + "\">         ");
    out.println("</td>                                       ");
    out.println("</tr>                                       ");
    out.println("<tr>                                        ");
    out.println("<td align=right>Phone:</td>                 ");
    out.println("<td>                                        ");
    out.print("<input name=phoneac size=3 maxlength=3        ");
    out.println("value=\"" + phoneac.trim() + "\">           ");
    out.print("<input name=phoneex size=3 maxlength=3        ");
    out.println("value=\"" + phoneex.trim() + "\">           ");
    out.print("<input name=phonenr size=4 maxlength=4        ");
    out.println("value=\"" + phonenr.trim() + "\">           ");
    out.println("&nbsp;Ext:&nbsp                             ");
    out.print("<input name=ext size=4 maxlength=10           ");
    out.println(" value=\"" + ext.trim() + "\">              ");
    out.println("</td>                                       ");
    out.println("</tr>                                       ");
    out.println("<tr>                                        ");
    out.println("<td align=right>Email:                      ");
    out.println("</td>                                       ");
    out.println("<td>                                        ");
    out.print("<input name=email size=30 maxlength=50        ");
    out.println(" value=\"" + email.trim() + "\">            ");
    out.println("</td>                                       ");
    out.println("</tr>                                       ");
    out.println("</table>                                    ");
    out.println("<center>                                    ");
    out.println("&nbsp;                                      ");
    out.println("</center>                                   ");
```

Figure 77 (Part 7 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

```
    if (button.equals("SEARCH")) {
      out.println("<center>                                  ");
      out.print("<input type=submit value=NEWSELECT          ");
      out.println(" name=button>                             ");
      out.print("<input type=submit value=UPDATE             ");
      out.println(" name=button>                             ");
      out.print("<input type=submit value=ADD                ");
      out.println(" name=button>                             ");
      out.println("</center>                                 ");
    }
    else if (button.equals("ADD")) {
      out.println("<center>                                  ");
      out.print("<input type=submit value=NEWSELECT          ");
      out.println(" name=button>                             ");
      out.print("<input type=submit value=SAVE               ");
      out.println(" name=button>                             ");
      out.println("</center>                                 ");
    }
    else if ((button.equals("NEWSELECT")) ((
            (button.equals("SAVE"))       ((
            (button.equals("UPDATE"))) {
      out.println("<center>                                  ");
      out.print("<input type=submit value=NEWSELECT          ");
      out.println(" name=button>                             ");
      out.print("<input type=submit value=ADD                ");
      out.println(" name=button>                             ");
      out.println("</center>                                 ");
    }
    out.println("</form>                                     ");
    out.println("</blockquote>                               ");
    out.println("</body>                                     ");
    out.println("</html>                                     ");
    } // eof try
    catch(Exception e)
      {
        out.println("Error:");
        out.println(e.toString());
      }
   }
}
```

Figure 77 (Part 8 of 8). Example of a Servlet Doing a SELECT, UPDATE and INSERT Using JDBC

9.2 SQLJ Implementation for DB2 on OS/390

This section discusses the basic information about writing SQLJ programs including SQL statements, host variables, and comments in the program. In addition, this section will review SQL statements that are valid in an SQLJ program.

9.2.1 Including SQL Statements in an SQLJ Program

In an SQLJ program, all statements that are used for database access are in SQLJ clauses. SQLJ clauses that contain SQL statements are called executable clauses. An executable clause begins with the characters “#sql” and contains an SQL statement that is enclosed in curly brackets. The SQL statement itself has no terminating character. An example of an executable clause is:

```
#sql {DELETE FROM EMP};
```

An SQLJ program can contain the following types of static SQL elements:

- SELECT, SELECT INTO, FETCH
- INSERT
- searched UPDATE, positioned UPDATE
- searched DELETE, positioned DELETE
- COMMIT, ROLLBACK
- CREATE, ALTER, DROP
- CALL, for calls to stored procedures in supported languages
- SET special register, SET host variable

An executable clause can appear anywhere in a program where a Java statement can appear.

9.2.2 Using Java Variables and Expressions As Host Expressions

To pass data between a Java application program and DB2, you use host expressions. A Java *host expression* is a Java simple identifier or complex expression, preceded by a colon. The result of a complex expression must be a single value. When you use a host expression as a parameter in a stored procedure call, you can follow the colon with the IN, OUT, or INOUT parameter that indicates whether the host expression is intended for input, output, or both.

The following SQLJ clause uses a host expression that is a simple Java variable named EMPNO:

```
#sql {SELECT LASTNAME INTO :empname FROM EMP WHERE EMPNO='000010'};
```

The following SQLJ clause calls stored procedure A and uses simple Java variable EMPNO as an input or output parameter:

```
#sql {CALL A (INOUT :EMPNO)};
```

SQLJ evaluates host expressions from left to right before DB2 processes the SQL statements that contain them. For example, for the following SQL clause, Java increments variable x before DB2 executes the SELECT statement:

```
#sql {SELECT ACTDESC INTO :hvactdsc WHERE ACTNO=:(x++)};
```

Similarly, in the following example, Java determines array element yTiR and decrements i before DB2 executes the SELECT statement:

```
#sql {SELECT ACTDESC INTO :hvactdsc WHERE ACTNO=:(yTi--R)};
```

In an executable clause, host expressions, which are Java tokens, are case-sensitive. Everything else in an executable clause is case-insensitive, except for delimited SQL identifiers.

9.2.3 Including Comments

To include comments in an SQLJ program, use either Java comments or SQL comments. Java comments are denoted by “/*,” “*/” or “//.” You can include Java comments outside SQLJ clauses, wherever the Java language permits them. Within an SQLJ clause, use Java comments in host expressions. SQL comments are denoted by “*” at the beginning of a line or anywhere on a line in an SQL statement. You can use SQL comments in executable clauses anywhere, except in host expressions.

9.2.4 Handling SQL Errors and Warnings

SQLJ clauses use the JDBC class `java.sql.SQLException` for error handling. SQLJ generates an SQLException when an SQL statement returns a negative or positive SQLCODE. You can use the `getErrorCode` method to retrieve SQLCODEs and the `getSQLState` method to retrieve SQLSTATEs.

To handle SQL errors in your SQLJ application, import the `java.sql.SQLException` class, and use Java try/catch blocks to modify program flow when an SQL error occurs. For example:

```
try {
#sql {SELECT LASTNAME INTO :empname
FROM EMP WHERE EMPNO='000010'};
}
catch(SQLException e) {
System.out.println("SQLCODE returned: " + e.getErrorCode());
}
```

9.2.5 Including Code to Access SQLJ and JDBC Interfaces

Before you can execute any SQLJ clauses in your application program, you must include code to accomplish these tasks:

- Import the Java packages for SQLJ runtime support and the JDBC interfaces that are used by SQLJ.
- Load the DB2 for OS/390 SQLJ runtime JDBC driver and register it with the `DriverManager`. To load the DB2 for OS/390 SQLJ runtime JDBC driver and register it with the `DriverManager`, invoke method `Class.forName` with an argument of `COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`.

```
import sqlj.runtime.*; // SQLJ runtime support
import java.sql.*; // JDBC interfaces
.....
try {
Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
}
catch (ClassNotFoundException e) {
e.printStackTrace();
}
```

9.2.6 Connecting to a Data Source

In an SQLJ application, as in any other DB2 application, you must be connected to a data source before you can execute SQL statements. A data source in DB2 for OS/390 is a DB2 subsystem. In other environments, a data source is usually referred to as a database.

If you do not specify any data sources in an SQLJ program that you run on DB2 for OS/390, DB2 connects you to the local DB2 subsystem automatically. If you want to execute an SQL statement at another data source, you must specify a connection context, enclosed in square brackets, at the beginning of the execution clause that contains the SQL statement. For example, the following SQL clause executes an UPDATE statement at the data source associated with connection context myconn:

```
#sql TmyconnR {UPDATE DEPT SET MGRNO=:hvmgr WHERE DEPTNO=:hvdeptno};
```

A connection context is an instance of a connection context class. To define the connection context class and set up the connection context, use one of the methods discussed in 9.2.6.1, “Connection Method 1” or 9.2.6.2, “Connection Method 2,” before you specify the connection context in any SQL statements.

9.2.6.1 Connection Method 1

1. Execute a type of SQLJ clause called a “connection declaration clause” to define a connection context class.
2. Invoke the constructor for the connection context class with an argument that specifies the location name that is associated with the data source. This argument has the form:

   ```
   jdbc:db2os390sqlj:location_name
   ```

   Note: “location_name” must be defined in SYSIBM.LOCATIONS.

For example, suppose that you want to use the first method to set up connection context “myconn” to access data at a data source that is associated with location NEWYORK. First, execute a connection declaration clause to define a connection context class:

```
#sql context ctx;
```

Then invoke the constructor for generated class ctx with the argument: `jdbc:db2os390sqlj:NEWYORK:`:

```
ctx myconn=new ctx(jdbc:db2os390sqlj:NEWYORK);
```

9.2.6.2 Connection Method 2

1. Execute a connection declaration clause to define a connection content class.
2. Invoke the JDBC `java.sql.DriverManager.getConnection` method with an argument that specifies the location name that is associated with the data source. That argument has the form:

   ```
   jdbc:db2os390sqlj:location_name
   ```

   Note: “location_name” must be defined in SYSIBM.LOCATIONS.

   The invocation returns an instance of class `Connection`, which represents a JDBC connection to the data source.
3. Invoke the constructor for the connection context class.

   For the argument of the constructor, use the JDBC connection that results from invoking `java.sql.DriverManager.getConnection`. To use the second method to set up connection context “myconn” to access data at the data source associated with location NEWYORK, first execute a connection declaration clause to define a connection context class:

   ```
   #sql context ctx;
   ```

   Then, invoke `java.sql.Driver.GetConnection` with the argument:

   ```
   jdbc:db2os390sqlj:NEWYORK:
   ```

   ```
   Connection jdbccon=DriverManager.getConnection(jdbc:db2os390sqlj:NEWYORK);
   ```
4. Finally, invoke the constructor for class `ctx` using the JDBC connection as the argument:

```
    ctx myconn=new ctx(jdbccon);
```

SQLJ uses the JDBC `java.sql.Connection` class to connect to data sources. Your application can use the following methods in the `java.sql.Connection` class:

- `clearWarnings`
- `close`
- `commit`
- `getAutoCommit`
- `getMetaData`
- `getWarnings`
- `isClosed`
- `isReadOnly`
- `rollback`
- `setAutoCommit`

The following rules apply to the JDBC connection that you create for executing SQLJ clauses:

- If you use any methods in class `java.sql.Connection` other than those listed, SQLJ returns an error.
- The default value of `autoCommit` for all connections in an SQLJ program is `off`. That is, DB2 does not automatically commit units of work in SQLJ applications. You can use `setAutoCommit` to set autoCommit to on.
- For SQLJ, DB2 for OS/390 supports only type 1 connections. Therefore, DB2 does not support multiple concurrent connection contexts. If an application creates a new connection context, then DB2 attempts to create a new type 1 DB2 connection. If the application has an incomplete unit of work, the connection fails. However, if there are no incomplete units of work, DB2 creates the new connection and discards the old connection.
- You cannot use a connection that you create for executing SQLJ clauses to execute dynamic SQL statements through JDBC.

9.2.7 Using Result Set Iterators to Retrieve Rows from a Result Table

In DB2 application programs that are written in traditional host languages, you use a cursor to retrieve individual rows from the result table that is generated by a SELECT statement. The SQLJ equivalent of a cursor is a result set iterator. A *result set iterator* is a Java object that you use to retrieve rows from a result table. Unlike a cursor, you can pass a result set iterator as a parameter to a method.

You define a result set iterator using an *iterator declaration clause* The iterator declaration clause specifies a list of Java data types. Those data type declarations represent columns in the result table and are referred to as columns of the result set iterator.

Table 5 on page 133 shows each SQLJ data type that you can specify in a result set iterator declaration and the equivalent SQL data type.

Table 5. Equivalent SQLJ and SQL Data Types

SQLJ data type	SQL data type
java.lang.String	CHAR, VARCHAR, LONGVARCHAR, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC 1
java.math.BigDecimal	NUMERIC, INTEGER, DECIMAL, SMALLINT, FLOAT, REAL, DOUBLE
Boolean	INTEGER, SMALLINT
int, Integer	SMALLINT, INTEGER, DECIMAL, NUMERIC, FLOAT, DOUBLE
float, Float	SMALLINT, INTEGER, DECIMAL, NUMERIC, FLOAT, DOUBLE
double, Double	SMALLINT, INTEGER, DECIMAL, DECIMAL, NUMERIC, FLOAT, DOUBLE
byte 2	CHAR, VARCHAR, LONGVARCHAR, GRAPHIC, VARGRAPHIC, LONG, VARGRAPHIC
java.sql.Date 3	DATE
java.sql.Time 3	TIME
java.sql.Timestamp 3	TIMESTAMP

Notes:

1 If the data type of a column is GRAPHIC, VARGRAPHIC or LONG VARGRAPHIC, data is converted from CCSID 500 to Unicode when it is retrieved from a DB2 table into a Java host expression.

2 SQLJ performs no data type conversion for this data type.

3 This class is part of the JDBC API.

There are two types of result set iterators:

1. Positioned iterators
2. Named iterators

The type of result set iterator that you choose depends on the way that you plan to use that result set iterator. The following sections explain how to use each type of iterator.

9.2.8 Using Positioned Iterators

For a positioned iterator, the columns of the result set iterator correspond to the columns of the result table, in left-to-right order. For example, if an iterator declaration clause has two data type declarations, the first data type declaration corresponds to the first column in the result table, and the second data type declaration corresponds to the second column in the result table. You declare positioned iterators to execute FETCH statements.

For example, the following iterator declaration clause defines a positioned iterator named ByPos with two columns. The first column is of type String and the second column is of type Date.

```
#sql public iterator ByPos(String,Date);
```

When SQLJ encounters an iterator declaration clause for a positioned iterator, it generates a positioned iterator class with the name that you specify in the iterator declaration clause. You can then declare an object of the positioned iterator class to retrieve rows from a result table.

For example, suppose that you want to retrieve rows from a result table that contains the values of the LASTNAME and HIREDATE columns from the employee table. Figure 78 shows how you can declare an iterator named “ByPos” and use an object of the generated class ByPos to retrieve those rows.

```
{
#sql public iterator ByPos(String,Date);
// Declare positioned iterator class ByPos
ByPos positer; // Declare object of ByPos class
String name = null;
Date hrdate;
#sql positer = { SELECT LASTNAME, HIREDATE FROM EMP };  :rk.1:erk.
#sql { FETCH :positer INTO :name, :hrdate };  :rk.2:erk.
// Retrieve the first row from the result table
while ( !positer.endFetch() )  :rk.3:erk.
{ System.out.println(name + " was hired in " +
hrdate);
#sql { FETCH :positer INTO :name, :hrdate };
// Retrieve the rest of the rows
}
}
```

Figure 78. Example of an Iterator Declaration

Notes:

1 This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable positer.

2 SQLJ checks that the types of the host variables in the INTO clause match the positional corresponding types of the iterator columns.

3 Method `endFetch()`, which is a method of the generated iterator class `ByPos`, returns a value of true when all rows have been retrieved from the iterator.

9.2.9 Using Named Iterators

You can use named iterators to select rows from a result table using SQL statements other than FETCH statements. When you declare a named iterator for a query, you specify names for each of the iterator columns. Those names must match the names of columns in the result table for the query. An iterator column name and a result table column name that differ only in case are considered to be matching names.

When SQLJ encounters a named iterator declaration, it generates a named iterator class with the same name that you use in the iterator declaration clause. In the named iterator class, SQLJ generates an accessor method for each column name in the iterator declaration clause. The accessor method name is the same name as the column name in the iterator declaration clause. You use the accessor method to retrieve data from the corresponding column of the result table.

When you execute an SQL clause that has a named iterator, SQLJ matches the name of each iterator column to the name of a column in the result table. The columns of the iterator do not need to be in the same order as the columns of the result table.

The following iterator declaration clause defines named iterator ByName, which has two columns. The first column of the iterator is named LastName and is of type String. The second column is named HireDate and is of type Date.

```
#sql public iterator ByName(String LastName, Date HireDate);
```

You use a named iterator in an SQLJ assignment clause. An assignment clause assigns the result table from a SELECT statement to an instance of a named iterator class. For example:

```
#sql nameiter={SELECT LASTNAME, HIREDATE FROM EMP};
```

Figure 79 shows how you can use a named iterator to retrieve rows from a result table that contains the values of the LASTNAME and HIREDATE columns of the employee table.

```
{
#sql public iterator ByName(String LastName, Date HireDate);  1
ByName namiter; // Declare object of ByName class
#sql nameiter={SELECT LASTNAME, HIREDATE FROM EMP};  2
String name;
Date hrdate;
// advances to next row
while (namiter.next())  3
{
hrdate = namiter.HireDate();  4
// Returns value of column named HIREDATE
name = namiter.LastName();
// Returns value of column named LASTNAME
System.out.println(name + " was hired on " + hrdate);
}
}
```

Figure 79. Example of a Named Iterator

Notes:

1 This SQLJ clause creates named iterator class `ByName`, which has accessor methods `LastName()` and `HireDate()` that return the data from result table columns LASTNAME and HIREDATE.

2 This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable nameiter.

3 `next()`, which is a method of the generated class `ByName`, advances the iterator to successive rows of the result set. `next()` returns a value of true when a next row is available, and a value of false when all rows have been fetched from the iterator.

4 SQLJ checks that the types of the host variables in the assignment clause match the types returned by the corresponding accessor methods.

The column names for named iterators must be valid Java identifiers. The column names must also match the column names in the result table from which the iterator retrieves rows. If a SELECT statement that uses a named iterator selects data from columns with names that are not valid Java identifiers, you need to use SQL AS clauses in the SELECT statement to give the columns of the result table acceptable names.

For example, suppose you want to use a named iterator to retrieve the rows that are specified by this SELECT statement:

```
SELECT "bad colname" FROM GOODTABLE
```

The iterator column name must match the column name of the result table, but you cannot specify an iterator column name of bad colname. You must therefore use an AS clause to rename bad colname to a valid Java identifier in the result table. For example:

```
SELECT "bad colname" AS GOODCOLNAME FROM GOODTABLE
```

You can then declare a named iterator with a column name that is a valid Java identifier and matches the column name of the result table:

```
#sql public iterator ByName(String GoodColName);
ByName namiter;
#sql nameiter={SELECT "bad colname" AS GOODCOLNAME FROM GOODTABLE};
```

9.2.10 Using Iterators for Positioned UPDATE and DELETE Operations

Writing SQL statements to perform a positioned UPDATE or a positioned DELETE is somewhat different from writing SQL statements to retrieve data from a table. Positioned UPDATE and DELETE operations require two Java source files. In one source file, you must declare the iterator.

In the declaration, you must use an “SQLJ implements” clause to implement the `sqlj.runtime.ForUpdate` interface. You must also declare the iterator as public. For example, use the following SQL clause to declare iterator ByPos, which has string column EmpNo, for use in a positioned DELETE statement:

```
#sqlj public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

You can then use the iterator in a different source file. To use the iterator:

1. Import the generated iterator class.
2. Declare an instance of the generated iterator class.
3. Assign the SELECT statement for the positioned UPDATE or DELETE to the iterator instance.
4. Execute positioned UPDATE or DELETE statements using the iterator.

After the iterator has been created, any SQLJ source file that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator. The authorization ID under which a positioned UPDATE or DELETE statement executes is the authorization ID under which the DB2 package that contains the UPDATE or DELETE executes.

For example, suppose that you have declared iterator DelByName like this in file1.sqlj:

```
#sqlj public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

To use DelByName for a positioned DELETE in file2.sqlj, execute the following statements:

```
#sql TexecCtxR {DELETE FROM EMP WHERE SALARY > 10000};
import DelByName;  1
{
DelByName deliter; // Declare object of DelByName class
String enum;
#sql deliter = { SELECT EMPNO FROM EMP
WHERE WORKDEPT="D11"};  2
while (deliter.next())
{
enum = deliter.EmpNo(); // Get value from result table  3
#sql { DELETE WHERE CURRENT OF :deliter };  4
// Delete row where cursor is positioned
}
}
```

Notes:

1 This statement imports named iterator class `DelByName`, which was created by the iterator declaration clause for DelByName in file1.sqlj.

2 This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable deliter.

3 This statement positions the iterator to the next row to be deleted.

4 This SQLJ clause performs the positioned DELETE.

9.2.11 Monitoring and Modifying SQL Statement Execution

You can use methods of the SQLJ `ExecutionContext` class to query and modify the characteristics of SQL statements during execution.

To execute `ExecutionContext` methods on an SQL statement, you must declare an execution context and associate that execution context with the SQL statement.

To declare an execution context, invoke the constructor for `ExecutionContext` and assign the result to a variable of type ExecutionContext. For example:

```
ExecutionContext execCtx=new ExecutionContext();
```

To associate an execution context with an SQL statement, specify the name of the execution context, enclosed in square brackets, at the beginning of the execution clause that contains the SQL statement.

You can associate a different execution context with each SQL statement. If You also use an explicit connection context for an SQL statement, specify the connection context, followed by the execution context in the execution clause for the SQL statement. For example:

```
#sql TconnCtx, execCtxR {DELETE FROM EMP WHERE SALARY > 10000};
```

If you do not specify an execution context for an execution clause, SQLJ uses the default execution context.

After you associate an execution context with an SQL statement, you can execute `ExecutionContext` methods on that SQL statement. For example, you can use method `getUpdateCount()` to count the number of rows deleted by a DELETE statement:

```
#sql TconnCtx, execCtxR {DELETE FROM EMP WHERE SALARY > 10000};
System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");
```

9.2.12 Restrictions on DB2 for OS/390 SQLJ Programs

When you write your SQLJ programs, be aware of the following restrictions in DB2 for OS/390 SQLJ:

- You can execute only static SQL statements in your SQLJ application.
- Your program cannot have multiple concurrent connections.
- You can call stored procedures from an SQLJ application, but you cannot write stored procedures that use SQLJ. You cannot call stored procedures that return multiple result sets.
- DB2 for OS/390 SQLJ converts all Unicode strings to CCSID 500 before it passes the strings to DB2. Similarly, when SQLJ passes data from character columns to Java host variables, SQLJ converts the column data from CCSID 500 to Unicode.
- SQLJ applications run only in the call attachment facility (CAF) or Recoverable Resource Manager Services attachment facility (RRSAF) environments.

To help you begin coding your program, the sample servlet in Figure 80 on page 139 will display SYSIBM.SYSTABLES similar to the SAMPLE01 JDBC sample program presented in Figure 76 on page 119:

```
***************************** Top of Data **********************
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import sqlj.runtime.ref.*;
import sqlj.runtime.*;
import java.math.*;

// * ### 1 ###
// *
// * You must set connection context declaration
// *

#sql context  samplesqljctx;

// * ### 2 ###
// *
// * You must set iterator for the select
// *

#sql iterator samplesqljIter (String NAME);

public
class sample03 extends HttpServlet {
   static {
      try {

// * ### 1 ###
// *
// * You must register the DB2 for OS390 driver before attempting
// * to connect. This next statement will load the driver and
// * creates an instance of the class.

         Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver"); }
      catch (ClassNotFoundException e) {
         e.printStackTrace();
      }
   }

    public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
 PrintWriter  out;
 Connection con =null ;
 samplesqljctx con1 = null;
 res.setContentType("text/html");
 out = res.getWriter();
 out.println("<html>");
 out.println("<head><title>Hello OS390</title></head>");
 out.println("<body>");
 out.println("<h1>Hello OS390</h1>");
  try {
       out.println("<p>**** Lets try SQLJ ****</p>");

          out.println("<p> Lets trying to connect to db2.</p>");
```

Figure 80 (Part 1 of 3). Example of a Simple Servlet Accessing DB2 Via SQLJ

```
// ******************************************************************
// *
// * ### 2 ###
// *
// * In order to for SQLJ to connect to DB2, you will need to     *
// * know the following:                                          *
// * DbLocationName will be the DB2 Location name. This can be
// *                optain from DB2 admin.
// * Dbsubprotocol  is the JDBC subprotocol. DB2OS390SQLJ the
// *                sub protocol for OS390 DB2.
// * jdbcDriver     for OS390 SQLJ Driver
// *                COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
// * url            is required for a database connection. It
// *                consist of protocal + subprotocol + subname.
// *                For our OS390 DB2 example, protocol is "jdbc",
// *                subprotocol must be "DB2OS390SQLJ, and subname is
// *                the DB2 location name.
// *
// * jdbc also allows for user/password option. When running a
// * servlet under the Go Domino Web Server, the user/password
// * is ignored. The Go Domino Web Server will pass user to DB2.
// * Go Dominto Web Server will pass either Surrogate ID, Client
// * ID, or Server ID depending on the configuration options used.
// *
        String url = "jdbc:db2os390sqlj:SC58DDF";
// * ### 3 ###
// *
// * Ready to establish connection to DB2 for OS390 URL.
// * Note: The url is defined in the Global Static definitions above.

        con1 = new samplesqljctx(url);
        out.println("<p>**** We are connected to DB2 for OS/390.</p>");

// * ### 4 ###
// *
// * Lets do some DB2 queries.
// * Execute a Query

     samplesqljIter iter;
     int count=0;
     #sql  con1U iter = { SELECT NAME FROM SYSIBM.SYSTABLES };

     out.println("<p> Query now successful. Here is Results:</p>");

     while (iter.next()) {
       System.out.println(iter.NAME());
       out.println(" NAME = " + iter.NAME());
       count++;
        }
       out.println("<p> Result completed. Count = " + count + "</p>");
```

Figure 80 (Part 2 of 3). Example of a Simple Servlet Accessing DB2 Via SQLJ

```
 // * ### 5 ###
 // *
 // * Time to close
 // *
         con1.close();
         out.println("<p>Now we are Disconnect from DB2 for OS/390.</p>");

      } catch( Exception e ) {
      out.println("Error:");
      out.println(e.toString());
      }

      out.println("</body></html>");
   }
}
```

Figure 80 (Part 3 of 3). Example of a Simple Servlet Accessing DB2 Via SQLJ

Our next example, shown in Figure 81, will demonstrate JDBC SELECT, INSERT, and UPDATE similar to our sample02 JDBC program presented in 9.1, "JDBC Implementation for DB2 on OS/390" on page 117:

```
// *******************************************************************
// * sample03.java -                                                 *
// *
// *  This sample servlet will demonstrate SQLJ to OS390 for DB2. It
// *  will perform a SELECT, UPDATE, and INSERT operations.  In
// *  addition, this servlet will pass data between this servlet
// *  and the client.
// *
// *  The DB2 connect was done in the doget instead of the
// *  init() since we were using CAF and did not have RRSAF running
// *  until the end of our lab work for this book. RRSAF is
// *  required if you plan on performing the connect in the init()
// *  method.
// *
// *******************************************************************
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import sqlj.runtime.ref.*;
import sqlj.runtime.*;
import java.math.*;
import java.util.*;

// connection context declaration
#sql context  sample03sqljctx;

public class sample03  extends HttpServlet
{
   // ****************************************************************
   // * Define Global Static Variables to this class
   // * Note: An alternative to defining these variables as Static is
   // * to pass these using properties files.
   // ****************************************************************
```

Figure 81 (Part 1 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
   // ****************************************************************
   // *
   // * ### 1 ###
   // *
   // * In order to for SQLJ to connect to DB2, you will need to     *
   // * know the following:                                         *
   // * DbLocationName will be the DB2 Location name. This can be
   // *                optain from DB2 admin.
   // * Dbsubprotocol  is the JDBC subprotocol. DB2OS390SQLJ is the
   // *                sub protocol for OS390 DB2.
   // * jdbcDriver     for OS390 DB2 the following driver is required:
   // *                COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
   // * mytable        is the DB2 table being used in this program.
   // * url            is required for a database connection. It
   // *                consist of protocal + subprotocol + subname.
   // *                For our OS390 DB2 example, protocol is "jdbc",
   // *                subprotocol must be "DB2OS390SQLJ, and subname is
   // *                the DB2 location name.
   // *
   // * jdbc also allows for user/password option. When running a
   // * servlet under the Go Domino Web Server, the user/password
   // * is ignored. The Go Domino Web Server will pass user to DB2.
   // * Go Dominto Web Server will pass either Surrogate ID, Client
   // * ID, or Server ID depending on the configuration options used.
   // *
          sample03sqljctx con;
          sample03sqljIter1 iter1     ;
          sample03sqljIter2 iter2     ;
   static String DbLocationName = "SC58DDF";
   static String mytable        = "ODONNEL.MANUFACTURER";
   static String Dbsubprotocol  = "DB2OS390SQLJ";
   static String jdbcDriver     =
          "COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver";
   static String url            = "jdbc:" +
                                   Dbsubprotocol + ":" +
                                   DbLocationName;

   // * ### 2 ###
   // *
   // * You must setup the iterator which will be used later in
   // * this program.
   // *

#sql public iterator sample03sqljIter1 (String manufacturer);
#sql public iterator sample03sqljIter2 (String manufacturer,
                                        String address,
                                        String city,
                                        String state,
                                        String zip,
                                        String lastname,
                                        String firstname,
                                        String phoneac,
                                        String phoneex,
                                        String phonenr,
                                        String ext,
                                        String email) ;
```

Figure 81 (Part 2 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
// * ### 3 ###
// *
// * You must register the DB2 for OS390 driver before attempting
// * to connect. This next statement will load the driver and
// * creates an instance of the class.

static {
   try {
      Class.forName(jdbcDriver);
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   }
}

// ****************************************************************
// * Initialize servlet when it is first loaded                   *
// ****************************************************************
public void init(ServletConfig config)
throws ServletException
{
   super.init(config);
}

// ****************************************************************
// * Respond to user GET request                                  *
// ****************************************************************
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException,IOException
{
   res.setContentType("text/html");  //Lets setup to do some HTML
   res.setHeader("Pragma", "no-cache");
   res.setHeader("Cache-Control", "no-cache");
   res.setDateHeader("Expires", 0);
   ServletOutputStream out = res.getOutputStream();
   try
   {

// * ### 4 ###
// *
// * Note: The url is defined in the Global Static definitions above.

     con = new sample03sqljctx (url);   // uses sqlj conntype 1.

     doManUpdateScreen(req,res,con);
     con.close();  //Release connection
     out.println("connection close:");
   }
   catch(SQLException e)
    {
     out.println("Error:");
     out.println(e.toString());
    }
   out.close();
}
// ****************************************************************
// * Respond to user to handle Manufacturer Update Screen         *
// ****************************************************************
```

Figure 81 (Part 3 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
public void doManUpdateScreen(HttpServletRequest req,
                              HttpServletResponse res,
                              sample03sqljctx con)
              throws ServletException,IOException
{
 Vector ManufNameList = new Vector();
 ResultSet rs ;
 ServletOutputStream out = res.getOutputStream();
 String button       =  req.getParameter("button");
 String manufacturer  =  req.getParameter("manufacturer");
 String address      =  req.getParameter("address");
 String city         =  req.getParameter("city");
 String state        =  req.getParameter("state");
 String zip          =  req.getParameter("zip");
 String firstname    =  req.getParameter("firstname");
 String lastname     =  req.getParameter("lastname");
 String phoneac      =  req.getParameter("phoneac");
 String phoneex      =  req.getParameter("phoneex");
 String phonenr      =  req.getParameter("phonenr");
 String ext          =  req.getParameter("ext");
 String email        =  req.getParameter("email");
 //Servlets just hate nulls if referenced.
 //sooo, I elected to remove all nulls at this time.
 if (button == null) button = "NEWSELECT";
 if (manufacturer  == null) manufacturer = "";
 if (address       == null) address      = "";
 if (city          == null) city         = "";
 if (state         == null) state        = "";
 if (zip           == null) zip          = "";
 if (firstname     == null) firstname    = "";
 if (lastname      == null) lastname     = "";
 if (phoneac       == null) phoneac      = "";
 if (phoneex       == null) phoneex      = "";
 if (phonenr       == null) phonenr      = "";
 if (ext           == null) ext          = "";
 if (email         == null) email        = "";
```

Figure 81 (Part 4 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
// * ### 5 ###
// *
// * Lets do some DB2 queries.

  try {
  if (button.equals("UPDATE")) {
       #sql  conU  {UPDATE ODONNEL.MANUFACTURER
                       SET MAN_ADDRESS        = :address,
                           MAN_CITY           = :city,
                           MAN_STATE          = :state,
                           MAN_ZIP            = :zip,
                           MAN_CON_LAST_NAME  = :lastname,
                           MAN_CON_FIRST_NAME = :firstname,
                           MAN_CON_PHONE_AC   = :phoneac,
                           MAN_CON_PHONE_EX   = :phoneex,
                           MAN_CON_PHONE_NR   = :phonenr,
                           MAN_CON_EXT        = :ext,
                           MAN_CON_EMAIL      = :email,
                           MAN_LAST_UPD_UID   ='JAVAID'
                     WHERE MAN_NAME = :manufacturer };
  }
  else if (button.equals("SAVE")) {
       #sql  conU  {INSERT INTO ODONNEL.MANUFACTURER
                         (MAN_NAME,
                          MAN_ADDRESS,
                          MAN_CITY,
                          MAN_STATE,
                          MAN_ZIP,
                          MAN_CON_LAST_NAME,
                          MAN_CON_FIRST_NAME,
                          MAN_CON_PHONE_AC,
                          MAN_CON_PHONE_EX,
                          MAN_CON_PHONE_NR,
                          MAN_CON_EXT,
                          MAN_CON_EMAIL,
                          MAN_LAST_UPD_UID,
                          MAN_LAST_UPD_DATE)
                 VALUES (:manufacturer,
                         :address,
                         :city,
                         :state,
                         :zip,
                         :lastname,
                         :firstname,
                         :phoneac,
                         :phoneex,
                         :phonenr,
                         :ext,
                         :email,
                         'JAVAID',
                         CURRENT TIMESTAMP ) };
  }
```

Figure 81 (Part 5 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
  else if (button.equals("SEARCH")) {
        String manlist =  req.getParameter("manlist");
        #sql  conU iter2 = {SELECT MAN_NAME,
                               MAN_ADDRESS,
                               MAN_CITY,
                               MAN_STATE,
                               MAN_ZIP,
                               MAN_CON_LAST_NAME,
                               MAN_CON_FIRST_NAME,
                               MAN_CON_PHONE_AC,
                               MAN_CON_PHONE_EX,
                               MAN_CON_PHONE_NR,
                               MAN_CON_EXT,
                               MAN_CON_EMAIL
                          FROM ODONNEL.MANUFACTURER
                         WHERE MAN_NAME = :manlist };
      while(iter2.next())
      {
          manufacturer   = iter2.manufacturer();
          address        = iter2.address();
          city           = iter2.city();
          state          = iter2.state();
          zip            = iter2.zip();
           lastname       = iter2.lastname();
           firstname      = iter2.firstname();
           phoneac        = iter2.phoneac();
           phoneex        = iter2.phoneex();
           phonenr        = iter2.phonenr();
           ext            = iter2.ext();
           email          = iter2.email();
        }
   }
   if ((button.equals("NEWSELECT"))  ||  // test buttons be pushed
       (button.equals("UPDATE"))     ||
       (button.equals("ADD"))        ||
       (button.equals("SAVE"))) {
       manufacturer  = ""  ;  // Clear fields
       address       = ""  ;
       city          = ""  ;
       state         = ""  ;
       zip           = ""  ;
       firstname     = ""  ;
       lastname      = ""  ;
       phoneac       = ""  ;
       phoneex       = ""  ;
       phonenr       = ""  ;
       ext           = ""  ;
       email         = ""  ;
       sample03sqljIter1 iter1     ;
       #sql   conU iter1 = { SELECT MAN_NAME
                          FROM ODONNEL.MANUFACTURER
                         ORDER BY 1 } ;
```

Figure 81 (Part 6 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
    while(iter1.next())
    {
      ManufNameList.addElement(iter1.manufacturer());//SaveResults
     }
}
}
catch(SQLWarning e)
  {
    out.println("Warning:");
    out.println(e.toString());
  }
catch(SQLException e)
  {
    out.println("Error:");
    out.println(e.toString());
  }
try {
out.println("<html>");
out.println("<head>");
out.println("<title>Manufacturer Information</title></head>");
out.print("<BODY TEXT=\"#000000\" BGCOLOR=\"#FFFFFF\"
                LINK=\"#0000FF\"");
out.print(" VLINK=\"#663366\" ALINK=\"#000088\"");
out.println(" BACKGROUND=\"/redfade.gif\" NOSAVE>");
out.println("<spacer type=block width=100 height=100 align=left>");
out.println("<center><B><font size=+3>Manufacturer Information
               </font></center></B>");
out.println("<br><br><br>");
out.println("<blockquote>");
out.print("<form name=dman method=get");
out.println(" action=\"/servlet/sample03\">");
out.println("<table cols=2 width=90% ><tr>");
out.println("<td align=right width=100>Manufacturer Name:</td>");
out.println("<td>");
if  ((button.equals("NEWSELECT"))   ((
    (button.equals("UPDATE"))       ((
    (button.equals("SAVE")))
{
   out.println("<select name=manlist>             ");
   if (ManufNameList.isEmpty()) {
     out.println("<option selected> Error");
     out.println("</select>");
     }
   else {
     for (int i = 0; i < ManufNameList.size(); i++) {
        out.println("<option selected> " +
        ManufNameList.elementAt(i));
        }
     out.println("</select>");
```

Figure 81 (Part 7 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

```
       }
       out.println("&nbsp;&nbsp;");
      out.println("<input type=submit value=SEARCH name=button>");
      out.println("</td></tr>");
      }
   else {
     out.print("<input name=manufacturer size=30");
     out.println(" &nmaxlength=30 value=\"" + manufacturer.trim() + "\">");
     out.println("</td></tr>");
   }
   out.println("<tr><td align=right>Address:</td>           ");
   out.print("<td><input name=address size=30               ");
   out.println(" &nmaxlength=30 value=\"" + address.trim() + "\">");
   out.println("</td></tr>                                  ");
   out.println("<tr><td align=right>City:</td>              ");
   out.println("<td>                                        ");
   out.print("<input name=city size=20 maxlength=30         ");
   out.println(" value=\"" + city.trim() + "\">             ");
   out.println("&nbsp;State:&nbsp;                          ");
   out.print("<input name=state size=2 maxlength=2          ");
   out.println(" value=\"" + state.trim() + "\">            ");
   out.println("&nbsp;Zip:&nbsp;                            ");
   out.print("<input name=zip size=10 maxlength=10          ");
   out.println(" value=\"" + zip.trim() + "\">              ");
   out.println("</td></tr>                                  ");
   out.println("<tr>                                        ");
   out.println("<td align=right>Contact First Name:</td>    ");
   out.println("<td>");
   out.print("<input name=firstname size=16 maxlength=30    ");
   out.println(" value=\"" + firstname.trim() + "\">        ");
   out.println("&nbsp;Last Name:&nbsp;                      ");
   out.print("<input name=lastname size=16 maxlength=30     ");
   out.println(" value=\"" + lastname.trim() + "\">         ");
   out.println("</td>                                       ");
   out.println("</tr>                                       ");
   out.println("<tr>                                        ");
   out.println("<td align=right>Phone:</td>                 ");
   out.println("<td>                                        ");
   out.print("<input name=phoneac size=3 maxlength=3        ");
   out.println("value=\"" + phoneac.trim() + "\">           ");
   out.print("<input name=phoneex size=3 maxlength=3        ");
   out.println("value=\"" + phoneex.trim() + "\">           ");
   out.print("<input name=phonenr size=4 maxlength=4        ");
   out.println("value=\"" + phonenr.trim() + "\">           ");
   out.println("&nbsp;Ext:&nbsp                             ");
   out.print("<input name=ext size=4 maxlength=10           ");
   out.println(" value=\"" + ext.trim() + "\">              ");
   out.println("</td>                                       ");
   out.println("</tr>                                       ");
   out.println("<tr>                                        ");
   out.println("<td align=right>Email:                      ");
   out.println("</td>                                       ");
   out.println("<td>                                        ");
   out.print("<input name=email size=30 maxlength=50        ");
   out.println(" value=\"" + email.trim() + "\">            ");
   out.println("</td>                                       ");
   out.println("</tr>                                       ");
   out.println("</table>                                    ");
   out.println("<center>                                    ");
   out.println("&nbsp;                                      ");
   out.println("</center>                                   ");
```

Figure 81 (Part 8 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE

```
   if (button.equals("SEARCH")) {
      out.println("<center>                                  ");
      out.print("<input type=submit value=NEWSELECT         ");
      out.println(" name=button>                            ");
      out.print("<input type=submit value=UPDATE            ");
      out.println(" name=button>                            ");
      out.print("<input type=submit value=ADD               ");
      out.println(" name=button>                            ");
      out.println("</center>                                ");
    }
    else if (button.equals("ADD")) {
      out.println("<center>                                  ");
      out.print("<input type=submit value=NEWSELECT         ");
      out.println(" name=button>                            ");
      out.print("<input type=submit value=SAVE              ");
      out.println(" name=button>                            ");
      out.println("</center>                                ");
    }
    else if ((button.equals("NEWSELECT")) ||
            (button.equals("SAVE"))       ||
            (button.equals("UPDATE"))) {
      out.println("<center>                                  ");
      out.print("<input type=submit value=NEWSELECT         ");
      out.println(" name=button>                            ");
      out.print("<input type=submit value=ADD               ");
      out.println(" name=button>                            ");
      out.println("</center>                                ");
    }
    out.println("</form>                                      ");
    out.println("</blockquote>                                ");
    out.println("</body>                                      ");
    out.println("</html>                                      ");
    } // eof try
    catch(Exception e)
      {
        out.println("Error:");
        out.println(e.toString());
      }
   }
}
```

Figure 81 (Part 9 of 9). Example of a Servlet Doing a SELECT, UPDATE and DELETE Using SQLJ

9.2.13 Steps in the SQLJ Program Preparation Process

After you write an SQLJ application, you must generate an executable form of the application. This involves:

- Translating the source code to produce modified Java source code and serialized profiles.
- Customizing the serialized profiles to produce DBRMs.
- Binding the DBRMs into a plan.

9.2.14 Translating SQLJ Source Code

The first step in preparing an executable SQLJ program is to use the SQLJ translator to generate a Java source program and one or more serialized profiles.

You must invoke the DB2 for OS/390 SQLJ translator from the OS/390 UNIX System Services command line. The command syntax is:

```
sqlj                           javapgm.sqlj
     -help
     -version
     -dir = directory
     -props = properties-file
     -warn = all
             none
             verbose
             nonverbose
             portable
             nonportable
             file-list
```

The meanings of the parameters are:

-help Specifies that the SQLJ translator describes each of the options that the translator supports.

-version Specifies that the SQLJ translator returns the version of the SQLJ translator.

-dir=directory Specifies the name of the directory into which SQLJ puts output from the translator. This output consists of Java source files and serialized profile files. The default directory is the current directory. The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:

- file1.sqlj, which is not in a Java package
- file2.sqlj, which is in Java package sqlj.test

Also suppose that you specify the parameter "-dir=/src" when you invoke the translator. Then the translator puts the serialized profiles and Java source file for file1.sqlj in directory `/src` and puts the serialized profiles and Java source file for file2.sqlj in directory `/src/sqlj/test`.

-props=properties-file
Specifies the name of a file from which the SQLJ translator will obtain a list of options.

-warn=warning-level
Specifies the types of messages that the SQLJ translator returns. The meanings of the warning levels are:

all	The translator displays all warnings and informational messages. This is the default.
none	The translator displays no warnings or informational messages.

verbose The translator displays informational messages about the semantic analysis process.

nonverbose The translator displays no informational messages about the semantic analysis process.

portable The translator displays warning messages about the portability of SQLJ clauses.

nonportable The translator displays no warning messages about the portability of SQLJ clauses.

file-list Specifies a list of SQLJ source files to be translated. This is a required parameter. All SQLJ source file names must have the extension .sqlj.

For each source file, program-name.sqlj, the SQLJ translator produces the following files:

- The modified source program.

  The modified source file is named <program name>.java. Use javac (the Java compiler) to compile the modified source program.

- A serialized profile file for each connection context that is specified in the program.

  A serialized profile file is named <program name>_SJProfile<n>.ser, where n is 0 for the first serialized profile generated for the program, 1 for the second serialized profile generated, and so on.

  You must run the SQLJ customizer on each serialized profile file to produce a standard DB2 for OS/390 DBRM.

9.2.15 Customizing a Serialized Profile

After you use the SQLJ translator to generate serialized profiles for an SQLJ program, you must customize each serialized profile to produce a standard DB2 for OS/390 DBRM. To customize a serialized profile, execute the following command on the OS/390 UNIX System Services command line:

```
db2sqljc                       -pgmname=DBRM-member-name
         DATE(ISO|USA|EUR|JIS)
         TIME(ISO|USA|EUR|JIS)
         SQL(ALL|DB2)
-userid=authorization-ID serialized-profile-name
```

The parameters and options are:

DATE(ISO|USA|EUR|JIS) Specifies that date values that you retrieve from an SQL table should always be in a particular format, regardless of the format specified as the location default. For a description of these formats, see Chapter 3 of SQL Reference. The default is the value that was specified during DB2 installation in the DATE FORMAT field of Application Programming.

TIME(ISO|USA|EUR|JIS) Specifies that time values that you retrieve from an SQL table should always be in a particular format, regardless of the format specified as the location default. For a description of these formats, see Chapter 3 of SQL

Reference. The default is the value that was specified during DB2 installation in the TIME FORMAT field of Application Programming.

SQL(ALL|DB2) Indicates whether the source program contains SQL statements other than those recognized by DB2 for OS/390. SQL(ALL) is recommended for application programs whose SQL statements must execute on a server other that DB2 for OS/390. SQL(ALL) indicates that the SQL statements in the program are not necessarily for DB2 for OS/390. The SQLJ translator then accepts statements that do not conform to the DB2 for OS/390 syntax rules. The SQLJ translator interprets and processes SQL statements according to distributed relational database architecture (DRDA) rules. The SQLJ translator also issues an informational message if the program attempts to use IBM SQL reserved words as ordinary identifiers. SQL(ALL) does not affect the limits of the SQLJ translator. SQL(DB2), the default, indicates that the SQLJ translator should interpret SQL statements and check syntax for use by DB2 for OS/390. SQL(DB2) is recommended when the application server is DB2 for OS/390.

-pgmname=DBRM-member-name
Specifies the name for the DBRM that the SQLJ customizer generates. This name must be eight or fewer characters in length and must conform to the rules for naming members of MVS partitioned data sets.

-userid=authorization-ID
Specifies the authorization ID under which the DBRM that the customizer generates will be bound. That authorization ID must have the privileges to execute SQL statements in the plan from which the DBRM is bound.

serialized-profile-name Specifies the name of the serialized profile that is to be customized. Serialized profiles are generated by the SQLJ translator and have names of the form <program-name>_SJProfile<n>.ser.

In this case, “program-name” is the name of the SQLJ source program, without the extension .sqlj., and n is an integer between 0 and m-1, where m is the number of serialized profiles that the SQLJ translator generated from the SQLJ source program.

When the SQLJ customizer runs, it modifies the contents of the serialized profile. If the customizer runs successfully, it generates the following message:

```
Serialized Profile serialized-profile-name has
been customized for DB2 for OS/390.
```

You must customize all serialized profiles that were generated from all source files that constitute an SQLJ program.

9.2.16 Binding a Plan for an SQLJ Program

After you have customized the serialized profiles for your SQLJ application program, you must bind the DBRMs that are produced by the SQLJ customizer. You can bind the DBRMs directly into a plan or bind the DBRMs into packages and then bind the packages into a plan. The authorization ID under which you bind the plan must be the same as the value of the -userid parameter that you specified when you customized the profiles. For information on binding packages and plans, see Chapter 2 of *DB2 for OS/390 V5 Command Reference*, SG26-8960.

When using SQLJ programs running under WebAS, you are only allowed one plan per WebAS server. You will need to bind all of your servlets to a DB2 package and bind all packages to one DB2 plan per WebAS server.

9.2.17 Example of Using the SQLJ Translator

You will need to complete the configuration in the OS/390 OS/390 UNIX System Services for SQLJ before trying this example. The configuration is discussed in 8.3, “Configuring JDBC and SQLJ in OS/390 UNIX Shell” on page 112.

After you have coded your Java program using a .sqlj extension on the file, you will first need to run the translator. In our example, our program will be called sample03.sqlj. Also, our example will be done on OS/390 in the UNIX shell. At the command prompt, you will simply execute the following command:

```
:> sqlj sample03.sqlj
```

If there are no errors, you will get another prompt indicating that the translator completed. In addition, the translator will create a new sample03.java file and it will create a Serialized Profile named sample03_SJProfile0.ser.

Next, you will need to execute the Java compiler. For example, at the next command prompt, you enter:

```
:> javac sample03.java
```

If there are no errors, you will get another prompt indicating that the compiler completed. In addition, the compiler will create a new sample03.class file.

Next, you will need to execute the SQLJ profiler. For example, at the next command prompt, you enter:

```
:> db2profc -pgmname=sample03 sample03_SJProfile0.ser
```

If there are no errors, you will get a message indicating that the Serialized Profile sample03_SJProfile0.ser has been customized. In addition, the example above will generate a DBRM member called SAMPLE03. It will generate the member in a pds called <hlq>.DBRMLIB.DATA

The last step is to run the bind. Here is a sample of the bind statements we used to bind our program.

```
BIND PACKAGE(SAMPLE03) MEMBER(SAMPLE03) ACTION(REPLACE) -
     LIBRARY('<tsoprefix>.DBRMLIB.DATA') ISOLATION(CS)

BIND PLAN(SQLJPLAN) PKLIST(SAMPLE06.SAMPLE06, -
                          SAMPLE07.SAMPLE07, -
                          SAMPLE04.SAMPLE04, -
```

```
                                      SAMPLE03.SAMPLE03) -
           ISOLATION(CS) ACTION(REPLACE) RETAIN
```

Note: You must make sure that appropriate access is granted for the package and plan.

Chapter 10. Develop Java Solutions for CICS on OS/390

In this chapter we describe how you can develop applications in Java accessing CICS on OS/390.

Note that when we discuss the CICS Gateway for Java, we use the abbreviation CJGW.

Important

The solutions described in this chapter are based on products that were available at the time of writing this book. New products in the area of connecting CICS to Java may be introduced shortly after publication of this book. However, this chapter is not meant as a technical update of the latest CICS products, but to show you how you can integrate Java with CICS and how you can easily develop your solution on the workstation and deploy it on OS/390.

10.1 Overview of Internet Access to CICS on OS/390

CICS on OS/390 provides a number of ways of making it accessible from the Web. Table 6 provides a summary of the offerings available at the time of writing this book.

Table 6. Different Options for Accessing CICS

Description	Features	Where From
CICS Gateway for Java	Supports any Java-enabled Web browser	Download without charge from www.ibm.com/software/ts/cics/platforms/clients/. It is also included as part of IBM CICS Clients V2 and IBM Connectors
CICS Internet Gateway	Supports any 3270 application	Included in IBM Transaction Servers, IBM Internet Connection Servers, IBM CICS Clients V2 and IBM Connectors
CICS Web Interface	Provides direct connection to CICS/ESA. Well suited to intranet applications	Included in IBM CICS Transaction Server for OS/390. Installable function of CICS/ESA V4.1 from 11/96
EXCI CGI	Well-suited to general Internet access to CICS	SupportPac can be downloaded without charge from the Web

10.1.1 CICS Gateway for Java

The CICS Gateway for Java (CJGW) provides two ways of accessing CICS programs/transactions:

1. The CJGW classes may be used by a local application (for example, a servlet) to directly access CICS programs/transactions. In this case, the CJGW runs on the same machine as the servlet.

2. CJGW itself can also run as an application. When run as a Java application (that is, UNIX daemon, or MVS started task), it allows a Java-enabled Web browser or network computer to download a Java applet and transparently access CICS data and applications.

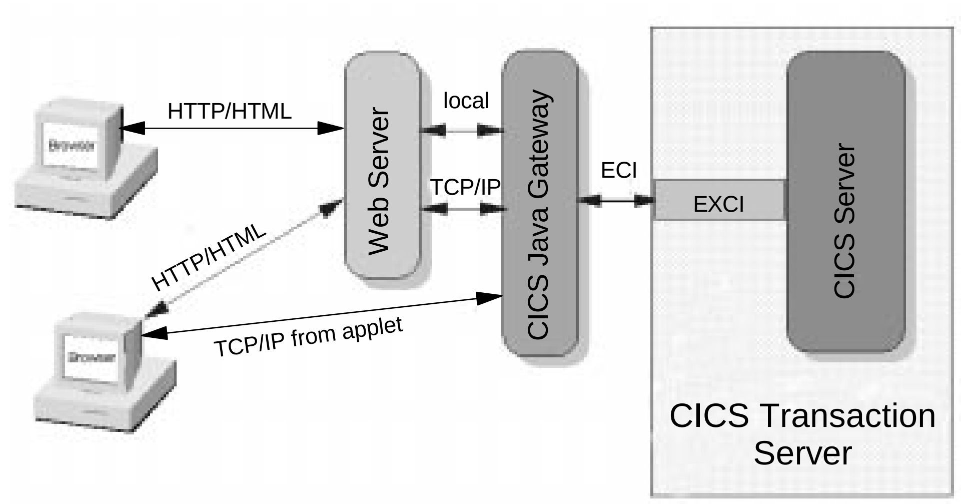


Figure 82. Accessing CICS Transactions from the Internet Using the CICS Gateway for Java

In Figure 82, notice that there is no CICS client. Instead, the CICS Transaction Server on OS/390 provides an EXCI interface. This interface will accept ECI calls from a non-CICS program, but not EPI. Note that this differs from Transaction Server on the UNIX, NT and OS/2 platforms where a CICS client is required.

Attention

The external CICS interface (EXCI) is an application programming interface that enables a non-CICS program (a client program) running in MVS to call a program (a server program) running in a CICS region and to pass and receive data by means of a communications area. The CICS application program is invoked as if linked to by another CICS application program.

10.1.2 CICS Internet Gateway

The CICS Internet Gateway is an IBM-provided CGI script that makes a Web browser appear like a 3270 terminal and hence provides Internet access to existing 3270 applications.

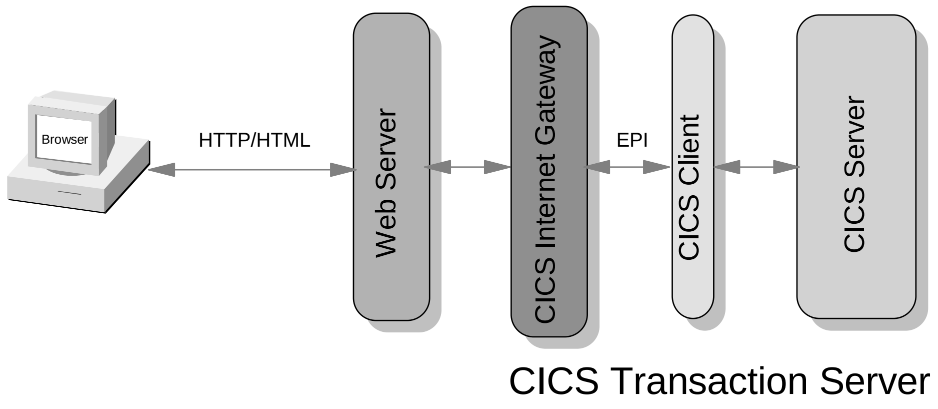


Figure 83. Accessing CICS Transactions from the CICS Internet Gateway

10.1.3 CICS Web Interface

Support in the CICS Transaction Server for OS/390 provides a TCP/IP connection directly into CICS (without going through a Webserver or gateway) and is the most efficient way for a Web browser to access CICS/ESA. It requires no co-requisite products other than TCP/IP for MVS (see Figure 84).

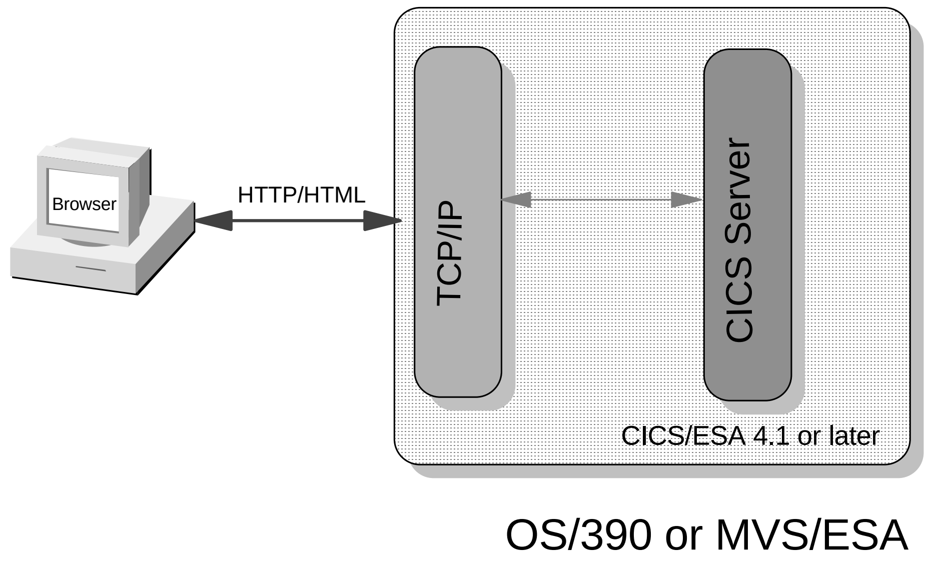


Figure 84. Accessing CICS Transactions from the Internet Using CICS Web Interface

10.1.4 EXCI CGI

IBM provides a sample of a CGI program which shows an interface between the IBM Webserver for OS/390 (running on MVS Open Edition) and CICS/ESA. This sample may be tailored by the user to meet specific application needs (see Figure 85 on page 158).

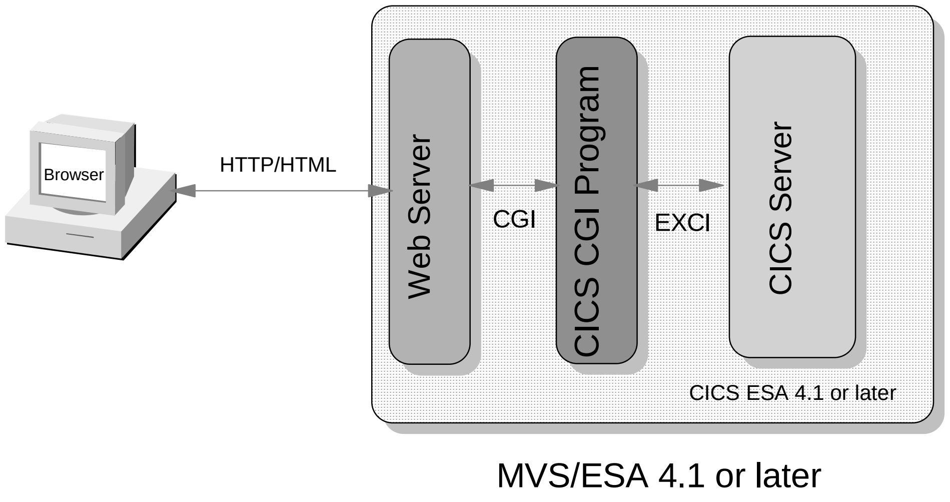


Figure 85. Accessing CICS Transactions from the Internet Using EXCI CGI

10.2 CICS Gateway for Java for OS/390 Architecture

In this section we focus on the CICS Gateway for Java. Our interest is in servlets and JavaServer Pages. This requires that Java be supported and to have the ability to support the ECI interface to CICS programs. Those requirements are well-supported by the CJGW.

CJGW on OS/390 provides two connectivity options to CICS. They are:

- Via CICS Gateway for Java Version 2.0 on MVS. We refer to this as a *two-tier architecture*.
- Via CICS Gateway for Java Version 2.0 on a middle tier, being UNIX or NT. We refer to this as a *three-tier architecture*.

Note that while there are many ways you can connect to CICS from the Web, as indicated in the sections preceding this, CICS Gateway for Java provides you with full Java support and supports the ECI interfaces.

With the Java support and combined with WebSphere Application Server, access can be made via servlets and JSPs.

10.2.1 CICS Gateway for Java Two-Tier Architecture

Let us look at the two-tier architecture.

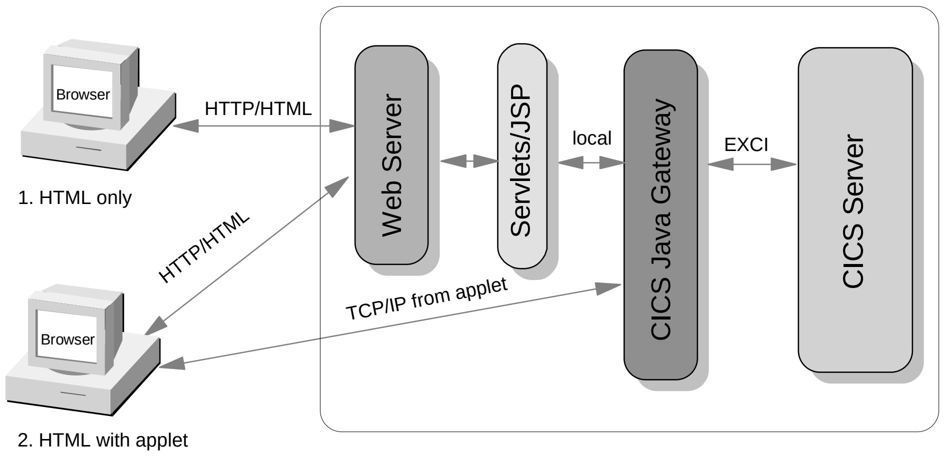


Figure 86. CICS Gateway for Java Two-Tier Architecture

In Figure 86, within the two-tier architecture, there are two methods of accessing the CICS transactions/programs:

1. An HTML-only approach that uses either servlets or JSPs. Both the servlets and JSPs will deliver only HTML. All communication with the CJGW is via the servlets on the server. The servlet does a local call to the CJGW. With this approach, CJGW itself need not be running.
2. An HTML plus an applet approach. It is the applet on the browser which will communicate with the CJGW. The applet is downloaded from the Webserver to the browser. The applet then initiates a communication channel with the CJGW. With this approach, the CJGW needs to be running as a daemon, or as an MVS started task, or as a long-running batch job.

Both approaches have merits and customers should choose whichever method fits their needs best. Note that these methods can also be combined. With the two-tier architecture, one of the key attractions is that OS/390 customers can start simple. There is little, if any, additional software required.

10.2.2 CICS Gateway for Java Three-Tier Architecture

In a three-tier architecture, the middle tier is typically a UNIX (AIX or Solaris) machine. It can also be another OS/390 partition/machine, or a low-end server such as Windows NT or OS/2. The architecture is very similar to the previous two-tier architecture.

With the three-tier architecture, the CICS Client exists as a separate component and resides in the middle tier. In Figure 87 on page 160, you see that the CICS client communicates with the CICS server using either the TCP62 or APPC protocols. (If the middle tier is OS/390, use EXCI instead of the CICS client.).

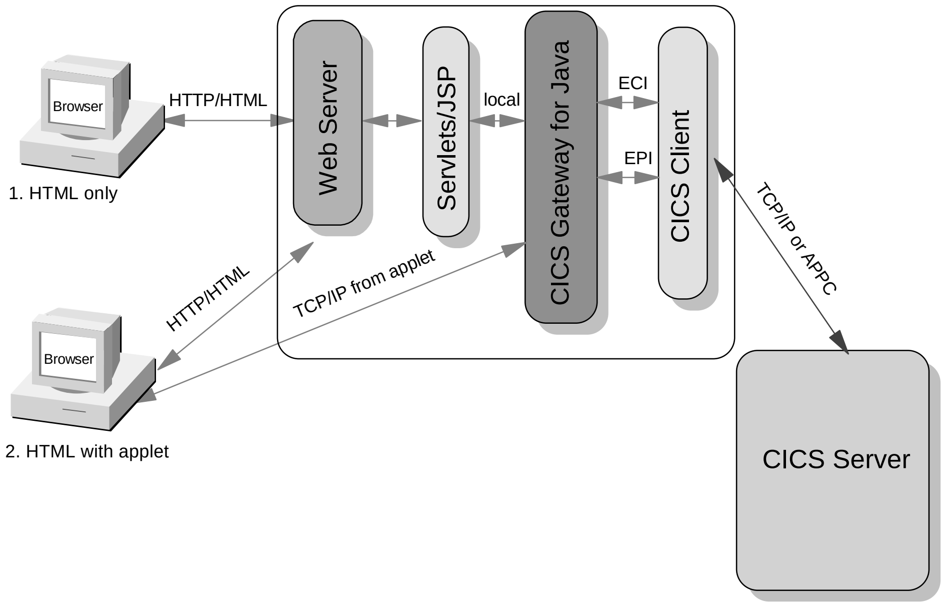


Figure 87. CICS Gateway for Java Three-Tier Architecture

10.2.2.1 The TCP62 Protocol

TCP62 is a protocol mapper that allows LU6.2 applications to communicate over a TCP/IP network with minimal SNA configuration of clients. It uses AnyNet as the underlying protocol. The following software provides integrated AnyNet support

- On the server side, OS/390 Version 1 Release 3 or later comes with VTAM Version 4 Release 4, which has integrated AnyNet support
- On the client side, IBM Personal Communications AS/400 and 3270 4.2 or later

Attention

CICS Transaction Server for OS/390 Release 3, the latest of CICS for OS/390, provides support for inbound requests to CICS/Java application programs. The client and the CICS server establish the basis for communication using an interface as defined by the Object Management Group (OMG): Interface Definition Language (IDL) standard using the IIOP protocol.

10.2.3 CICS Gateway for Java - A Further Introduction

The IBM CICS Gateway for Java provides a powerful way to access CICS transactions from the Internet. It combines the portable, architecture-neutral, object-oriented strengths of the Java programming environment with the power, high integrity, robustness and flexibility of CICS to bring state-of-the-art, open, easy, access from the Internet to mission-critical business CICS applications running on a wide variety of server platforms. This is highly relevant as various estimates by both IBM and third parties estimate that about 70% of all mission-critical applications run on IBM systems, which are predominantly CICS.

The CICS Gateway for Java is designed to link two different worlds: the world of enterprise computing with its mission-critical data and the rapidly growing, universal access world of the Internet. Java is the key that gives access to the Internet. Not only can it turn any Internet-connected computer into a universal client, but it brings far greater richness to interaction with the Web. So, the CICS Gateway for Java enables you and your customers and business partners, equipped with just a Network Computer or a PC, to transact your business directly and promptly.

Java applets and Java servlets can be used with JavaServer Pages (JSP) to access CICS transactions via the CJGW. With servlets and JSPs, you are able to provide dynamic content based on the data from CICS transactions to your customers in pure HTML format, or an HTML/Java applet combination. Note that in both cases, you can always have JavaScript for the front-end. JSPs provide a nice way to logically separate the presentation interface from the processing and data layer.

In the CICS section of this book, we concentrate on accessing CICS transactions via servlets and JSPs.

The CICS Gateway for Java is provided for the OS/390 (MVS), OS/2 (R), Windows NT, AIX (R) and Solaris platforms.

The CICS Gateway for Java consists of two components:

1. A CICS Java class library, which includes three classes that provide interfaces, and are used to communicate between the Java gateway application and a Java application or applet. If the Java application or servlet is local (that is, on the same machine as the CJGW), the Java application or servlet may use these supplied classes to communicate directly with CICS programs/transactions.
2. A supplied Java application that uses these classes can be run as an MVS started task, long-running batch job or UNIX System Services (USS) daemon to handle remote requests (that is, applets on browsers).

   This application communicates with CICS applications running in CICS servers through the EXCI interfaces provided by the CICS Transaction Server. The EXCI interface enables a non-CICS client application as a subroutine. The EPI interface (provided by CICS Client but not by EXCI) enables a non-CICS client application to act as a logical 3270 terminal and so control a CICS 3270 application.

The CICS Java Class library consists of a number of Java classes and interfaces, which provide a simple way of making calls to CICS programs.

The CICS Gateway for Java can concurrently manage many communication links to connected Web browsers, and can control asynchronous conversations to multiple CICS server systems.

The multithreaded architecture of the Gateway enables a single Gateway to support multiple concurrently connected users.

The CICS Gateway for Java (MVS) runs directly in the OS/390 UNIX System Services with CICS Transaction Server for OS/390 V1R2 or later as the CICS server in a two-tier configuration. For more detail, refer to information on the CICS Transaction Server for OS/390 V1R2 (Ivory Letter 297-393), announced on September 9, 1997. Announcement letters may be viewed on the Internet at URL:

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

The CICS Gateway for Java incorporates NLS (including DBCS) support. Messages and documentation are translated to a range of languages; the desired language is selected when downloading from the Internet.
Note, however, that the correct display of translated messages and documentation (especially in DBCS languages) is dependent on respectively the locales installed and the capabilities of the browser being used. For more information, refer to the CICS Home page at URL:

```
http://www.ibm.com/software/ts/cics/
```

In this redbook, we focus on the CICS Gateway for Java Version 2.0, which may be used with any JDK 1.1 compliant Java-enabled Web browser. At the time of writing, the latest version, which is called CICS Transaction Gateway, is not supported on OS/390 yet and therefore is not explored further in this book.

The CICS Gateway for Java V2.0.1 runs on a Java-enabled platform with the following minimum levels of JDK (Java Development Toolkit) installed:

- Windows NT: JDK 1.1.4 or later
- OS/2: JDK 1.1.1 or later preferably 1.1.4
- AIX: JDK 1.1.1 or later preferably 1.1.2
- Solaris: JDK/JIT V1.1.4 for SPARC-based machines or later, with native threads support for Solaris Version 2.5.1 or later
- OS/390: JDK 1.1.1 or later

With CICS Transaction Server for OS/390 (TM) V1R2 and later, CICS Gateway for Java (MVS) provides equivalent ECI support, similar to that provided by the ECI of CICS Client, and therefore operates without the need for any CICS clients.

10.2.4 The CICS Gateway for Java Classes

The CICS Gateway for Java consists of the following classes and interfaces:

- Basic classes for writing Java-client programs:

```
ibm.cics.jgate.client.JavaGateway
ibm.cics.jgate.client.ECIRequest
ibm.cics.jgate.client.EPIRequest
ibm.cics.jgate.client.CicsCpRequest
ibm.cics.jgate.client.Callbackable
ibm.cics.jgate.client.GatewayRequest
```

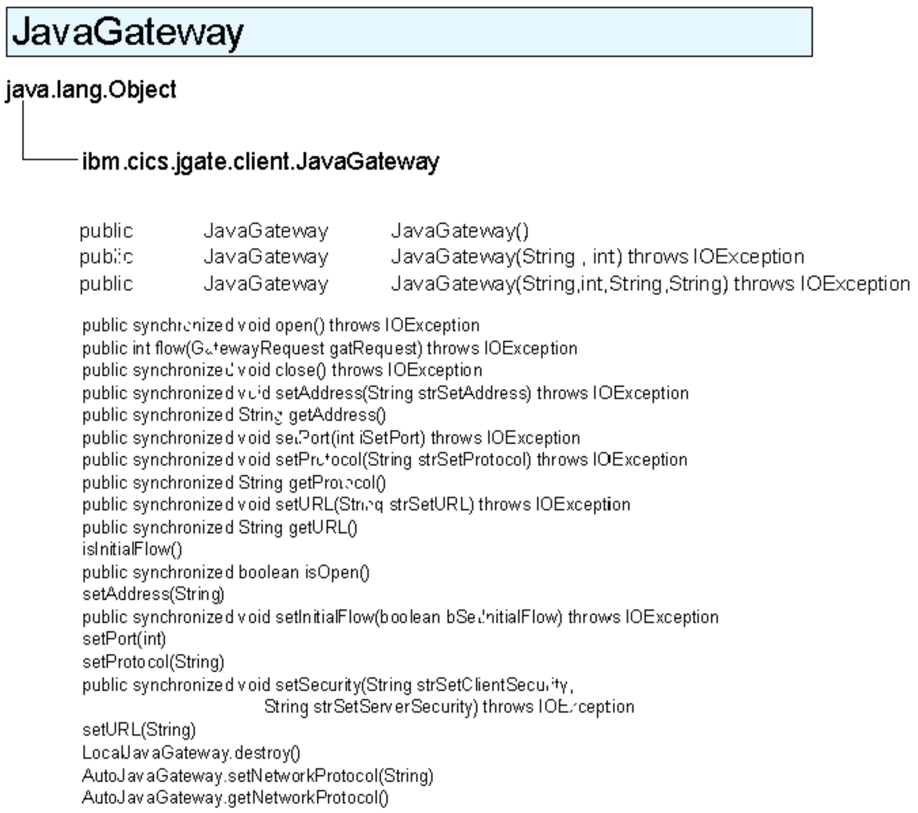
- Interface definitions for writing Gateway security classes:

```
ibm.cics.jgate.security.ClientSecurity
ibm.cics.jgate.security.ServerSecurity
```

- Higher-level EPI support classes:

```
ibm.cics.jgate.epi.*
```

Figure 88 on page 163 shows the `ibm.cics.jgate.client.JavaGateway` class.



JavaGateway

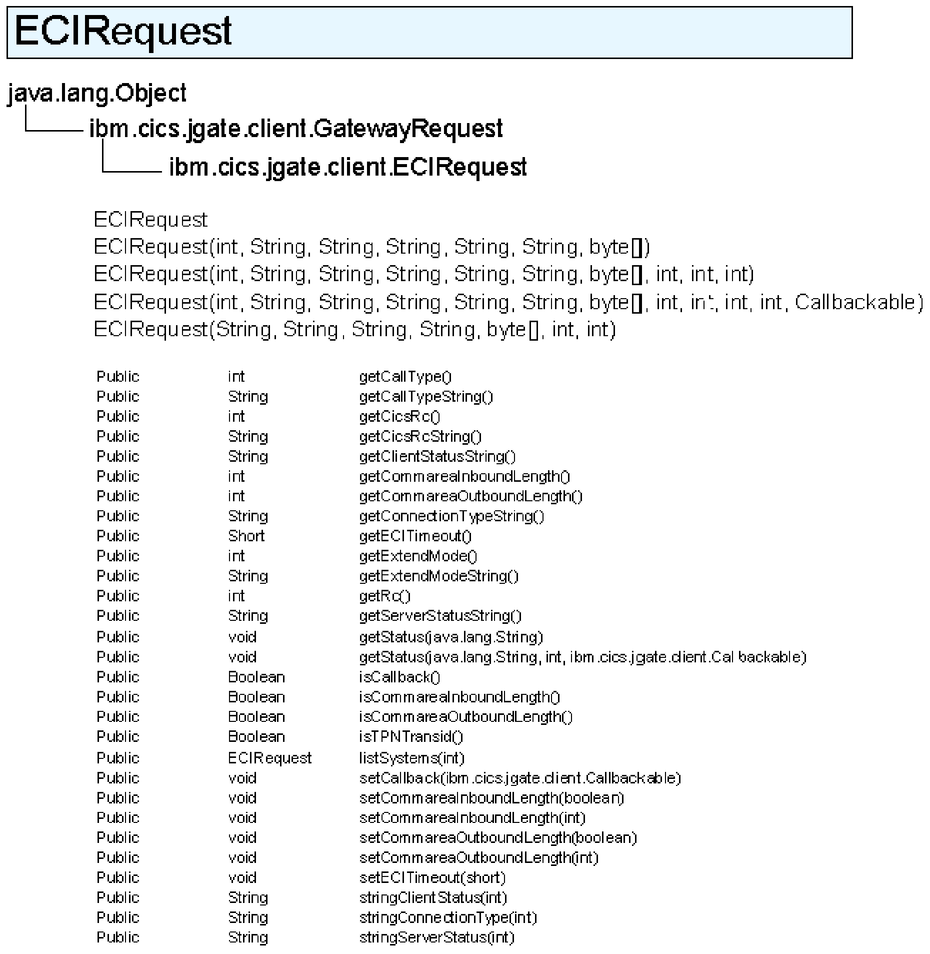
java.lang.Object

ibm.cics.jgate.client.JavaGateway

```
public      JavaGateway      JavaGateway()
public      JavaGateway      JavaGateway(String , int) throws IOException
public      JavaGateway      JavaGateway(String,int,String,String) throws IOException

public synchronized void open() throws IOException
public int flow(GatewayRequest gatRequest) throws IOException
public synchronized void close() throws IOException
public synchronized void setAddress(String strSetAddress) throws IOException
public synchronized String getAddress()
public synchronized void setPort(int iSetPort) throws IOException
public synchronized void setProtocol(String strSetProtocol) throws IOException
public synchronized String getProtocol()
public synchronized void setURL(String strSetURL) throws IOException
public synchronized String getURL()
isInitialFlow()
public synchronized boolean isOpen()
setAddress(String)
public synchronized void setInitialFlow(boolean bSetInitialFlow) throws IOException
setPort(int)
setProtocol(String)
public synchronized void setSecurity(String strSetClientSecurity,
                                     String strSetServerSecurity) throws IOException
setURL(String)
LocalJavaGateway.destroy()
AutoJavaGateway.setNetworkProtocol(String)
AutoJavaGateway.getNetworkProtocol()
```

Figure 88. CICS JavaGateway Class

Figure 89 on page 164 shows the `ibm.cics.jgate.client.ECIRequest` class.


ECIRequest

java.lang.Object
└── ibm.cics.jgate.client.GatewayRequest
  └── ibm.cics.jgate.client.ECIRequest

ECIRequest
ECIRequest(int, String, String, String, String, String, byte[])
ECIRequest(int, String, String, String, String, String, byte[], int, int, int)
ECIRequest(int, String, String, String, String, String, byte[], int, int, int, int, Callbackable)
ECIRequest(String, String, String, String, byte[], int, int)

Public	int	getCallType()
Public	String	getCallTypeString()
Public	int	getCicsRc()
Public	String	getCicsRcString()
Public	String	getClientStatusString()
Public	int	getCommareaInboundLength()
Public	int	getCommareaOutboundLength()
Public	String	getConnectionTypeString()
Public	Short	getECITimeout()
Public	int	getExtendMode()
Public	String	getExtendModeString()
Public	int	getRc()
Public	String	getServerStatusString()
Public	void	getStatus(java.lang.String)
Public	void	getStatus(java.lang.String, int, ibm.cics.jgate.client.Callbackable)
Public	Boolean	isCallback()
Public	Boolean	isCommareaInboundLength()
Public	Boolean	isCommareaOutboundLength()
Public	Boolean	isTPNTransid()
Public	ECIRequest	listSystems(int)
Public	void	setCallback(ibm.cics.jgate.client.Callbackable)
Public	void	setCommareaInboundLength(boolean)
Public	void	setCommareaInboundLength(int)
Public	void	setCommareaOutboundLength(boolean)
Public	void	setCommareaOutboundLength(int)
Public	void	setECITimeout(short)
Public	String	stringClientStatus(int)
Public	String	stringConnectionType(int)
Public	String	stringServerStatus(int)

Figure 89. CICS Java Gateway Class for ECI

However, for the purposes of our sample applications, we will use only two classes:

- `ibm.cics.jgate.client.JavaGateway`
- `ibm.cics.jgate.client.ECIRequest`

`ibm.cics.jgate.client.JavaGateway` is used to establish communication with the long running Gateway process using Java's socket protocol (or directly, if it is local).

`ECIRequest` is used to specify the CICS ECI calls which are “flowed” to the Gateway. The Gateway channels the ECI calls through a CICS Client to the desired CICS server applications, manages the many communication links to the connected browser or network computers, and controls asynchronous conversations to the CICS server systems.

Attention

The CICS Gateway for Java (MVS) is supported only with CICS Transaction Server for OS/390 V1R2 and higher.

The CICS Gateway for Java Version 2.0.1 (including the CICS Gateway for Java (MVS)) uses and requires JDK 1.1 classes.

The CICS Gateway for Java, supports the HTTP communications protocol as well as the TCP protocol, enabling communication through firewalls. Note that the CICS Gateway for Java needs to run as a Java application. It can run on the same machine as the one used to run the Webserver.

CICS Gateway for Java, combined with Java servlets and JSPs, leads to many powerful scenarios. For example, an HTML page in the Web client can call a Java servlet which can call a CICS program via the CJGW. The servlet can build dynamic HTML "on the fly" based on results returned by the CICS program. Alternatively, a JSP page can call a JavaBean which can also interact with CJGW and deliver dynamic HTML to the client.

If your Java servlet is on the same machine as your CICS Client, you specify "local" (or "auto") as the location of the CICS Gateway for Java. This allows the servlet to directly access CICS via the CGJW classes, without the need to go through a daemon or started task. If the Webserver is on a different machine than the CJGW, then the servlet will communicate with the CJGW daemon (or started task) via TCP/IP.

Another approach is to use Java applets. A Java applet in the Web-client can directly call CICS programs and data simply by invoking the small Java class supplied with the Gateway. When the applet is invoked, all the necessary code is downloaded to the client platform automatically, so no work is needed to prepare Web clients for CICS access.

The CICS Gateway for Java is downloadable from the Web if you have a CICS Transaction Server product. Its home page is at URL:

```
http://www.ibm.com/software/ts/cics/platforms/internet/cicsgw4j/
```

10.3 Overview of Approach

This section describes the steps required to develop and set up Web access to CICS Transactions via Java servlets on the OS/390. We show this with and without the use of JSPs. We used Windows NT as our development platform, and then deployed our application on OS/390. This allowed us to utilize the productivity tools on the workstation such as VisualAge for Java, VisualAge for COBOL, Enterprise Version 2.2, TxSeries, DB2 UDB, NetObject ScriptBuilder, and NetObject Fusion. It also serves to demonstrate the true platform-independence of the finished Web applications. This provides the freedom to choose, a key promise of the Java phenomenon.

In the next two chapters, we concentrate on several ways of developing servlets that call and execute a sample CICS/DB2 program called MANUFACT.

10.4 A Brief Discussion of Servlets and CICS

Servlets can invoke CJGW classes to interact with CICS programs and transactions. In a general scenario, a Web client would send a request to a Webserver indicating that the recipient should be a servlet. The Webserver passes on the request to the servlet with the appropriate data specified in the form of the HTML page.

The servlet pulls the data supplied off the request object. It then invokes the CJGW classes directly or indirectly. Through the CJGW, the CICS program is executed and returns the data back to the servlet, again directly or indirectly. The servlet could build an HTML data stream in the response object and send it back to the Web browser.

10.5 Developing a Java Application Using CICS Gateway for Java

As Java and many of the supporting pieces of software required to develop and run the CICS/COBOL/DB2 applications run on multiple platforms, the customer has many choices as to where to develop, and to deploy. We chose the workstation as our development platform, with a view of running the completed application on OS/390.

10.5.1 Requirements on the Development Platform (Workstation)

In the following sections, we list the technical requirements to develop and run Web-based applications accessing CICS transactions via the CICS Gateway for Java on Windows NT.

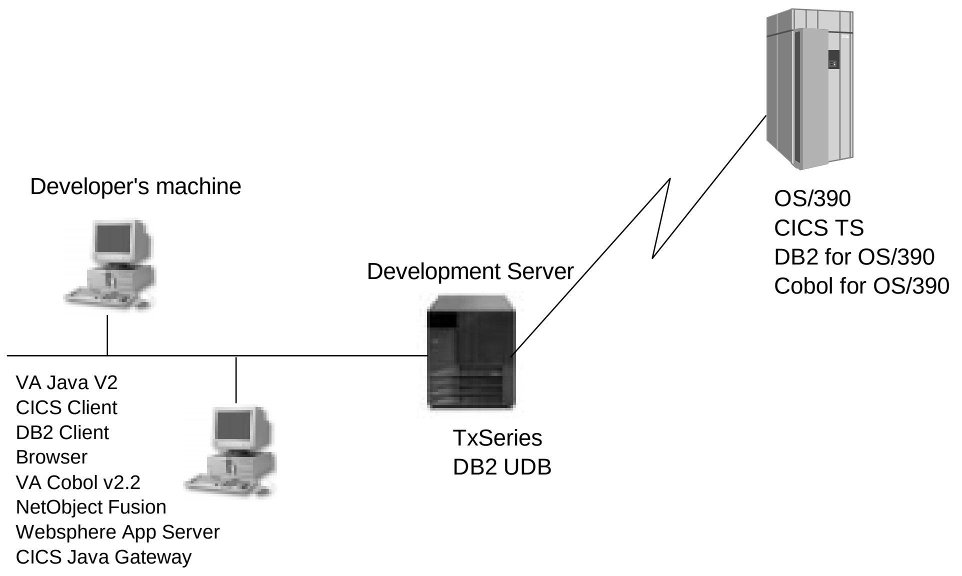


Figure 90. High-Level Overview of Development Environment Architecture

Our proposed development architecture consists of doing development largely on the workstation, and transferring the code to run on OS/390. Figure 90 shows the various pieces of software running on the PCs, the server and OS/390.

10.5.2 Components

We briefly discuss the roles played by each piece of software in the following sections.

10.5.2.1 Operating Systems

On the developer's machines and on our development server, we used Windows NT 4.0 and ServicePack 3. Note that OS/2 Warp is an alternative as the developer's machine. The server may also be AIX or OS/2 Warp.

10.5.2.2 Webserver

You will need a Webserver that can run WebSphere Application Server. There is a list of supported Webservers at URL:

```
http://www.ibm.com/software/webwervers/appserv/
```

10.5.2.3 WebSphere Application Server

The WebSphere Application Server contains a Java Servlet engine software plug-in (that was previously referred to as ServletExpress) that allows your existing Webservers (like the Lotus Domino Go Webserver, Apache Server, Microsoft IIS, and Netscape Enterprise Server) to run servlets. The WebSphere Application Server's home page is at URL:

```
http://www.ibm.com/software/webservers/appserv/
```

The WebSphere Application Server is installed after the Webserver and a JDK is installed. Note that WebSphere Application Server for Windows NT Version 2.0 comes bundled with both Apache and IBM HTTP Server. However, in our examples we used WebSphere Application Server for Windows NT Version 1.1..

10.5.2.4 CICS Gateway for Java

The CICS Gateway for Java makes the communication between Java and CICS transactions/programs possible. The CICS Gateway for Java can receive requests from Java applets, applications and servlets. See the overview section of 10.2, “CICS Gateway for Java for OS/390 Architecture” on page 158 for more information on CICS Gateway for Java.

We chose to put the CICS Gateway for Java on the developer's machine as this allows the CICS Gateway for Java classes that are used in your servlet to communicate directly with a CICS Client without the need to run a separate CICS Gateway for Java process.

10.5.2.5 CICS Client

The CICS Client accepts the ECI and EPI requests to run CICS programs or transactions to a CICS Server. If you specify “local” when you open a CICS Gateway for Java connection, then the CICS Client must be on the same machine as the CICS Gateway for Java. For our purposes, we have the CICS Client software on the developer's machine.

The CICS Client home page is at URL:

```
http://www.ibm.com/software/ts/cics/platforms/clients/
```

CICS Client for NT V2.0.4 was used during the development of our examples. Note that there is no need for a CICS Client on OS/390; instead, it uses EXCI to support ECI requests.

10.5.2.6 CICS Server
The CICS Server runs the back-end programs. These programs usually access data controlled by CICS or a database like DB2. TxSeries 4.2 was used as our CICS server for the development of the sample. We put TxSeries on a separate machine to act as our CICS server. The CICS home page is at URL:

```
http://www.ibm.com/software/ts/cics/
```

10.5.2.7 Web Browser
You will need a Web browser to display the HTML generated by the Java servlet or JSP. The HTML generated by our servlet (or JSP) is very simple HTML, so older browsers should work fine. Note that the HTML generated by NetObjects Fusion may require later versions of browsers. In our samples, we used Netscape Communicator 4.07. Note that if you use applets, it will require the browsers to support Java 1.1.

10.5.2.8 JDK
The Java Development Kit (JDK) contains the runtime classes needed to test the Java programs. While the JDK also provides utilities such as the Java compiler to compile a Java servlet, we used VisualAge for Java to develop and compile our samples.

The JDK was used for the installation of Domino Go Webserver and ServletExpress. The JDK can be downloaded from URL:

```
http://java.sun.com/products/jdk/1.1/
```

For our purposes we used JDK 1.1.6 for Windows NT.

10.5.2.9 Visualage for Java
The IDE of VisualAge for Java Enterprise Edition Version 2.0 provided us with a powerful development environment. Some of the key features we really liked were automatic syntax checking and class/method validation, the graphical debugging facility of VisualAge for Java, and the version control capability.

10.5.2.10 VisualAge for COBOL
We used VisualAge for COBOL, Enterprise Version 2.2 to develop and deploy the CICS/COBOL/DB2 backend programs. We used this product in conjunction with DB2 UDB and TxSeries 4.2.

10.5.2.11 Database
Our sample Cobol application accesses a DB2 table called the MANUFACTURER table.

For the workstation, we used DB2 UDB V5. In our development setup, each of us had the full-blown DB2 product on the workstation. An obvious alternative is to have a central DB2 server, with the developers accessing it via a DB2 client in a customer development environment. This is reflected in Figure 90 on page 166.

10.5.2.12 Web Pages

We used NetObjects ScriptBuilder V2 to build simple HTML and JSP pages. One of our team members was using NetObjects Fusion, a WYSIWYG Web tool to build professional looking pages. This is documented in Chapter 7, “NetObjects Fusion (NOF) Version 3” on page 93.

10.5.3 CICS Gateway for Java Customization on the Workstation

This section contains information to guide you through the basic tasks associated with running and using the CICS Gateway for Java on Windows NT.

10.5.3.1 Downloading the CICS Gateway for Java for Windows NT

If you have not already downloaded the CICS Gateway for Java for Windows NT package, you can download it from URL:

```
http://www.ibm.com/software/ts/cics/downloads/
```

When you have downloaded and unpacked the package, read the README.TXT file which tells you how to access the product web pages locally.

10.5.3.2 Installing the CICS Gateway for Java for Windows NT

The file you have downloaded is an archive file appropriate to the operating system and language you chose for your installation platform. If you downloaded it to a different platform, remember to transfer the file before attempting installation.

For Windows NT, the downloaded archive is a self-extracting executable file. The following instructions use English file names:

1. Move the downloaded file to the directory where you will create the root directory for the installation.
2. Run the downloaded file. The executable will expand now and will create a root directory, named `JGate`. The result is a directory structure as shown in the diagram in Figure 91 on page 170.

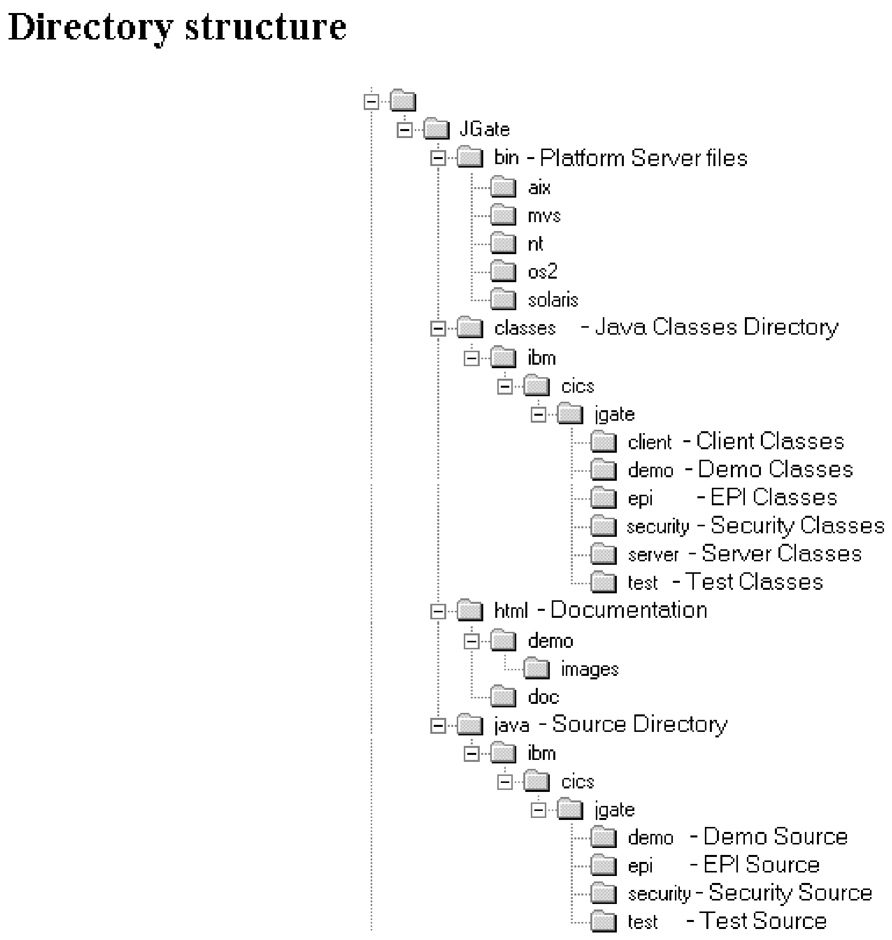


Figure 91. CICS Gateway for Java for Windows NT Directory Structure

You will find the README.TXT file in the installation root directory JGATE.

10.5.3.3 Configure CICS Gateway for Java for Windows NT

Use the following steps to configure the CICS Gateway for Java for Windows NT:

1. Configure your Webserver for CICS Gateway for Java.

   Define to your Webserver the location of the directory into which you installed the CICS Gateway for Java (JGATE). Your Webserver's documentation will explain how to do this.

2. Configure your programming environment for the CICS Gateway for Java.

   If you wish to compile or run Java applications, add JGATE/classes to your CLASSPATH of your environment.

3. Set the time.

   You must set your locale to match your timezone to get the right time. For example, if you set your locale to en_US, Java will be set to the time in the eastern part of the United States.

10.5.3.4 The Gateway.properties File

The Gateway.properties file allows you to set persistent properties for the CICS Gateway for Java. These properties are read when the Gateway is started, and can be broadly split into three categories:

General start-up properties
These properties are those that can also be specified via command line options when the Gateway is started.

Network protocol handler properties
These properties define which network protocol handlers are started. The Gateway supports dynamic protocol handler loading, and so additional protocol handlers can be added by adding entries in the properties file. Also protocol handler specific parameters can be specified.

Platform specific properties
Some platforms may need additional properties to determine operation of the Gateway.

The Gateway.properties file is located in, `JGATE/bin/nt` directory.

10.5.3.5 Starting the CICS Gateway for Java for Windows NT

You start the CICS Gateway for Java at the operating system command prompt of the computer on which you have installed it. First, you must set your working directory to `JGATE/bin/nt`.

You can use the start command in three ways with preset options, or with user-defined options, or you can get help on startup options.

- To start the Gateway with preset options:

  Preset options are those predefined in the Gateway code itself, or set in the Gateway.properties file. If an option was not specified in the Gateway.properties file, then the predefined Gateway value will be used. Type `JGate` at the command prompt and press **Enter**. You will then see the startup message:

  ```
  CCL6500I: Starting the CICS Gateway for Java with default values.
  ```

  This will be followed by two lines showing the values which are being used:

  ```
  CCL6502I: [ Initial ConnectionManagers = 1 , Maximum ConnectionManagers = 100 ,
  CCL6502I:   Initial Workers = 1 , Maximum Workers = 100, tcp: Port = 2006 ]
  ```

- To start the Gateway with user-specified options:

  The user-definable options are shown in Figure 92 on page 172.

```
-port=<port_number>   - TCP/IP port number for the tcp: protocol

-initconnect=<number> - Initial number of ConnectionManager threads

-maxconnect=<number>  - Maximum number of ConnectionManager threads

-initworker=<number>  - Initial number of Worker threads

-maxworker=<number>   - Maximum number of Worker threads

-trace                - Enable extra tracing messages

-time                 - Enable timing information in messages

-noinput              - Disable the reading of input from the console

-nonames              - Do not display TCP/IP hostnames
```

Figure 92. User-Definable Options for Starting Up the Gateway

To override the startup defaults, type `JGate` at the command prompt, followed by start-up options you require, and press Enter. Options specified on the command line override those specified in the Gateway.properties file. You will now see the startup message:

```
CCL6501I: Starting the CICS Gateway for Java with user specified values.
```

This will be followed by two lines showing the values which are being used, for example:

```
CCL6502I: [ Initial ConnectionManagers = 10 , Maximum ConnectionManagers = 100 ,
CCL6502I:   Initial Workers = 10 , Maximum Workers = 100, tcp: Port = 2345 ]
```

- To get help on the startup options, type:

  ```
  JGATE ?
  ```

10.5.3.6 Stop the CICS Gateway for Java for Windows NT

If you did not start the Gateway with the `-noinput` parameter, the Gateway can be stopped by typing the correct character and pressing the Enter key in the Gateway console session. The allowable characters may be localized for your country; the default characters allowed are “Q” or “-.”

You can determine what characters will stop the Gateway by simply pressing the Enter key in the Gateway console session. The following message will then be displayed:

```
CCL6508I: Type Q or - to stop the CICS Gateway for Java.
```

If you have used the `-noinput` parameter, you must stop the Gateway process using some other method. Some examples of such methods are :

- Enter “Ctrl-C” in the Gateway console session
- Use the NT Task Manager

10.5.4 Set Up TxSeries

For development purposes, we chose the CICS Development Server as part of the TxSeries installation. Note that, with TxSeries 4.2 for NT, the Encina SFS product and a light version of DCE is automatically installed as part of the TxSeries standard install. Refer to the *TxSeries Quick Beginnings Guide*, GC33-1879. for installation details.

10.5.4.1 Set Up the Development CICS Region

After the installation and setup has run to its conclusion, and after the machine has been rebooted, the following are the summary steps executed to prepare the environment:

- Configure DCE as an RPC-only environment. For our development purposes, this is sufficient.

  You may run the following command in a DOS prompt window.

  ```
  cicscp -v create dce -R
  ```

- Set up the environment variables. The installation of TxSeries creates a number of environment variables for both CICS and Encina. You need to create two additional ones manually. They are:

  ```
  set CICS_HOSTS=<your_machine_name>
  ```

  and

  ```
  set ENCINA_BINDING_FILE=c:\var\cics_servers\server_bindings
  ```

  After you set these environment variables, reboot the machine.

- Create the CICS region, listener and program definitions, as follows:

  ```
  cicscp -v create region CICSNT
  ```

  The creation of the CICS region will also automatically create an SFS server which is named after the host name prefixed with an “S.”

- Start the CICS region now with the following command:

  ```
  cicscp -v start region CICSNT
  ```

  You can start up the CICS regions from the command line. The CICS can also be managed via the Administration utility that comes with the TxSeries installation. To use the Administration utility, go to the **Start** button, choose **CICS Server**, and you should see the Administration utility as one of the options.

- You can define your programs to CICS using “cicsadd” or you can do so via the Administration utility. To do so with the Administration utility, right click on the CICS region, select **resources**, then **programs**. However, for multiple programs, it is easier to use “cicsadd.”

  ```
  cicsadd -c pd -r CICSNT -B TIMEZONE PathName=/userprogs/runtime/TIMEZONE
  ProgType=program ActivateOnStartup=yes
  ```

  ```
  cicsadd -c pd -r CICSNT -B MANUFACT PathName=/userprogs/runtime/MANUFACT
  ProgType=program ActivateOnStartup=yes
  ```

- Modify the `\winnt\system32\drivers\etc\services` file to insert the following line:

  ```
  CICSTCP         1435/tcp              #TxSeries listener
  ```

10.5.4.2 Set Up the CICS Client

After the CICS Client is installed, modify the CICSCLI.INI file as follows:

```
Server = CICSNT                 ; Arbitrary name for the server
Description = TCP/IP Server     ; Arbitrary description for the server
Protocol = TCPIP                ; Matches with a Driver section below
NetName = 9.12.2.176            ; The server's TCP/IP address
Port = 0                        ; Use the default TCP/IP CICS port
```

Some of the useful commands to manage the CICS CLIENT are:

```
CICSCLI /s=CICSNT           where CICSNT is your server name
CICSCLI /l                  list out the active servers
CICSCLI /x                  terminate the connection to your CICS server
CICSCLI /i                  terminate your connection immediately
```

Note that you need to change to the `/CICSCLI/BIN` directory, or have it in the path. These commands are executed from the DOS prompt window.

10.5.5 Prepare the COBOL Programs

As noted earlier, we used VisualAge for COBOL, Enterprise Version 2.2 with TxSeries 4.2 to develop and test our COBOL programs on the workstation. On the workstation, TxSeries provides command files to translate, compile and linkedit COBOL programs. These steps may be done individually, or in a combined manner. It can call either the IBM Cobol or the MicroFocus Cobol compiler, depending on the option specified.

For example, to translate for CICS, compile and link-edit the TIMEZONE program using IBM's VisualAge for COBOL, Enterprise Version 2.2, issue:

```
cicstcl -lIBMCOB timezone.ccp
```

Attention

Note the following regarding the syntax of `cicstcl`:

The `-l` indicates the language. In our case, it is VisualAge COBOL.

The output in this case is timezone.ibmcob. This must be placed in the directory as defined to TxSeries via the Program Definitions (refer to the set-up of TxSeries as described in 10.5.4, “Set Up TxSeries” on page 173).

If you are using the cicstcl utility (as described), you can set your compile options via the CICS_IBMCOB_FLAGS environment variable, for example, `set CICS_IBMCOB_FLAGS=-qdynam` for dynamic linking.

VisualAge for COBOL, Enterprise Version 2.2 has a nice syntax-sensitive editor called “iwzwlx40” that you can invoke from the command line. An alternative is to use the Workframe feature of VisualAge for COBOL, Enterprise Version 2.2 to translate, compile and linkedit your COBOL programs. Finally, remember to point SYSLIB to your copybook directories.

10.5.6 Verify Your CICS Setup

Start up a CICS client. For the purposes of this demo set-up, you may wish to set the “Resource Level Security Key” of the test programs to public. To do this, bring up the CICS Administration menu, right click on the CICS region, select **Resources** and then **Program**. Then select the **Security/DCE** tab, and spin the “Security Key” to public.

To verify your CICS set-up, you can bring up a CICS terminal and issue the CECI transaction as follows:

```
CECI LINK PROG(TIMEZONE) COMMAREA('') LENGTH(16)
```

You should get the following resulting screen as shown in Figure 93.

```
LINK PROGRAM(TIMEZONE) COMMAREA(' ') LENGTH(16)
STATUS:  COMMAND EXECUTION COMPLETE                          NAME=
 EXEC CICS  LInk Program( 'TIMEZONE' )
  < Commarea( '11-12-9820:07:15' )
  < Length( +00016 ) >
  < Datalength() > >
  < SYSid() >
  < Transid() >
  < SYNconreturn >
```

Figure 93. CICS Setup Verification on NT

10.5.7 Test Your Setup

We suggest you test your setup using the following steps:

1. Test your CICS environment as described in 10.5.4, “Set Up TxSeries” on page 173. Leave your CICS Client running. This makes sure that your CICS environment is ready.
2. Start up or make sure your Webserver is up and ready. Verify this by displaying an HTML page, a servlet or a JSP that you know works.
3. Start up your CICS Gateway for Java. Look at the output. You could verify your gateway using the supplied testeci program, which is explained later in this section.
4. Make sure your CLASSPATH is correctly set, for example, in our case it is:

   ```
   SET CLASSPATH=
       .;D:\WebSphere\AppServer\lib\ibmwebas.jar;d:\WebSphere\AppServer\classes;
       d:\WebSphere\AppServer\lib\jsdk.jar;d:\WebSphere\AppServer\lib\x509v1.jar;
       d:\WebSphere\AppServer\lib;d:\WebSphere\AppServer\web\classes\ibmjbrt.jar;
       d:\WebSphere\AppServer\lib\databeans.jar;D:\JGate\classes;D:\jdk1.1.6\lib\classes.zip
   ```

5. Move the `itsorb.TimeZone` class to your `webserver\classes` directory. For example, in our case, move the itsorb directory to the `D:\WebSphere\AppServer\classes` directory. It should be on the CLASSPATH. In this example, `TIMEZONE.class` sits in the `D:\WebSphere\AppServer\classes\itsorb` directory.
6. Check your CICS Java Gateway using the supplied TESTECI sample, as follows:

```
java ibm.cics.jgate.test.TestECI jgate=jgate.machine.ip.address prog0=TIMEZONE
server=TOT71 commarealength=16
```

Refer to the supplied CICS Gateway for Java documentation for the full syntax. You should get the following output (the bottom half is shown):

```
Commarea length : 16

No of programs given : 1
  [0] : TIMEZONE

=== Connect to Gateway ===

Successfully created JavaGateway

=== Available Servers ===

System : TOT71, Description : TCP/IP Server

=== Call Programs ===

About to call : TIMEZONE
  Commarea    :
  Extend_Mode : 0
  Luw_Token   : 0
Commarea      : 11-12-9820:24:34
Return code   : 0
Abend code    :
Successfully closed JavaGateway
```

7. Move the Timezone.html and Timezone.jsp files to an appropriate directory that is accessible by WebSphere. In our case, it is `D:\WebSphere\AppServer\samples\itso` with an appropriate “pass” entry in the httpd.cnf file.

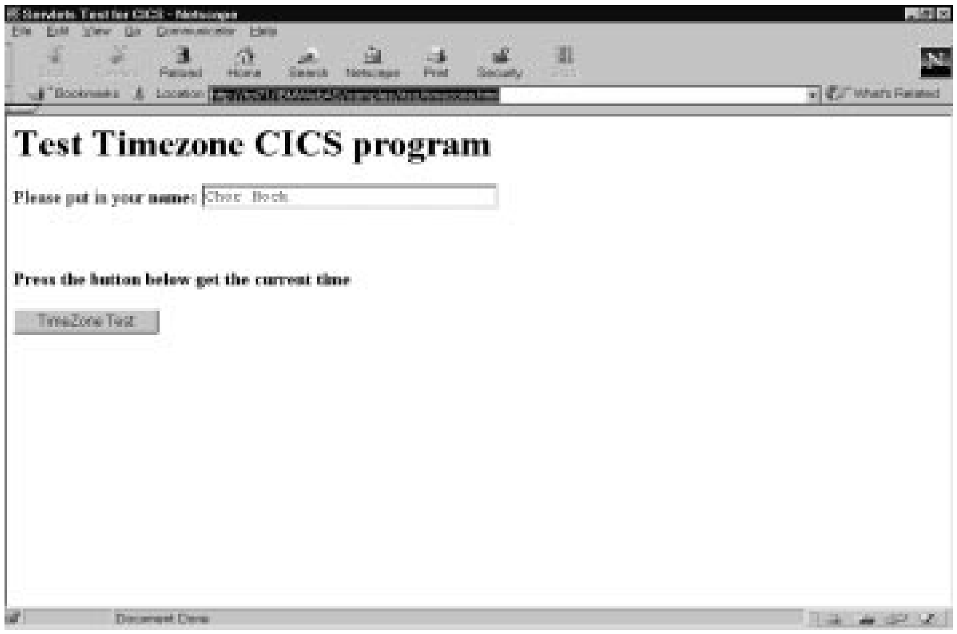
   Next, update the gateway settings to reflect your environment. Here, we show you an extract from the timezone.jsp file. Edit the highlighted lines to reflect your environment before running it.

```
<% // retrieve the fullname variable from the previous form
String fullName = request.getParameter("fullName");
// call the CICS program which will return the Time and Date
// but first set up your Gateway properties
 itsorb.JGateSettings jg = new itsorb.JGateSettings();

 jg.setCicsProgram("TIMEZONE");
 jg.setCicsServer("tot71");
 jg.setCicsTranid("MANU");
 jg.setCicsJGate("auto://tot71");
 jg.setCicsUserid("CICSUSER");
 jg.setCommareaLength(16);

 timeZoneBean.callCICS(jg);
%>
```

8. Open your Timezone.html using a browser in the normal way, as follows:

```
http://tot71/IBMWebAS/samples/itso/Timezone.html
```

9. When presented with the HTML page, type in your name and press the **Test** button. You should see a dynamic page built by JSP using a JavaBean displayed! Refer to Figure 94 on page 177 for our sample output.


Servlets Test for CICS - Netscape
File Edit View Go Communicator Help
Reload Home Search Netscape Print Security
Bookmarks Location
What's Related
Document: Done

Test Timezone CICS program

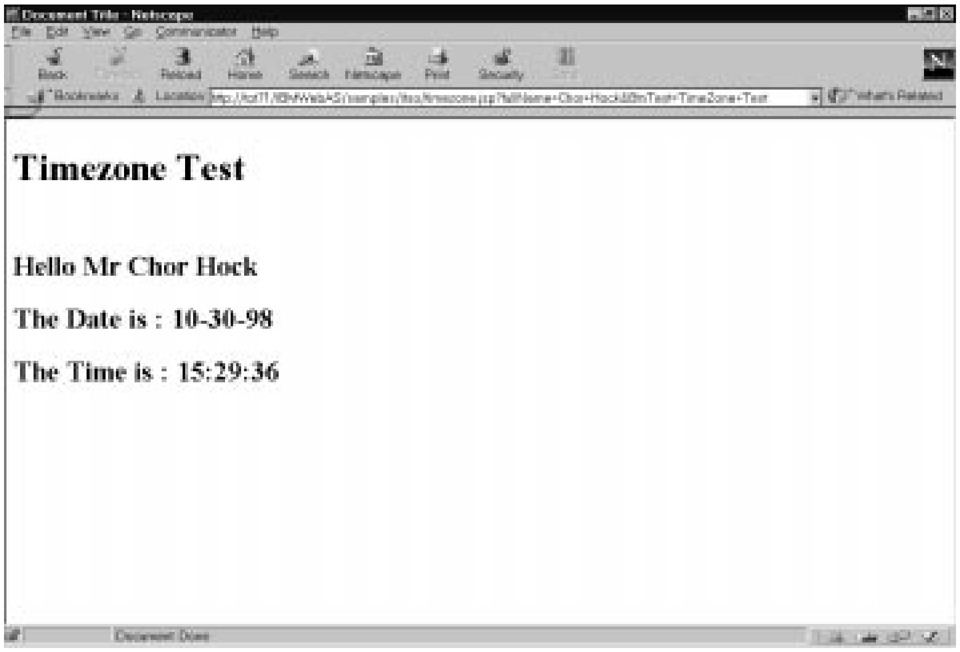
Please put in your name: Chor Hock

Press the button below get the current time

TimeZone Test

Figure 94. Sample Input to Test Access to a CICS Program Via a JSP/Bean

For input in Figure 94, you should get the output in Figure 95.


Document Title - Netscape
File Edit View Go Communicator Help
Back Reload Home Search Netscape Print Security
Bookmarks Location: http://tot71/IBMWebAS/samples/itso/timezone.jsp?fullName=Chor+Hock&btnTest=TimeZone+Test
What's Related
Document: Done

Timezone Test

Hello Mr Chor Hock

The Date is : 10-30-98

The Time is : 15:29:36

Figure 95. Sample Output to Test Access to a CICS Program Via a JSP/Bean

10.5.8 VisualAge for Java Setup to Develop Servlets for CICS Gateway for Java

Before you start developing any servlets, or extend existing ones, the servlet classes and the CICS Gateway for Java classes need to be imported into VisualAge for Java.

WebSphere comes with the java servlet classes jar file named jst.jar. Refer to Figure 96 for the directory structure of the contents of jst.jar file. We added the servlet classes into a project of its own called “WebSphere.”

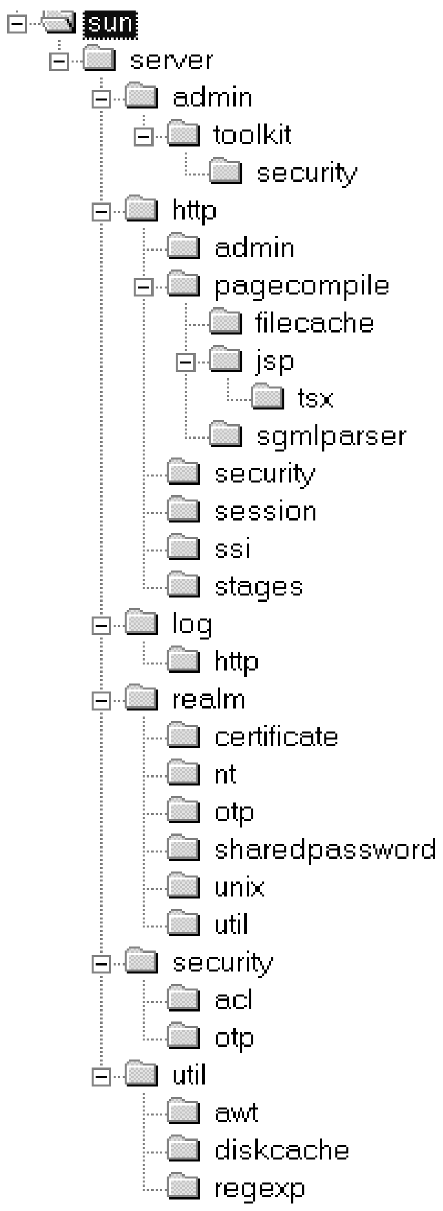


Figure 96. Adding the Servlet Builder in VisualAge for Java

The CICS Gateway for Java classes also need to be added to VisualAge for Java. In our case, we created a new project for it which we called “CICS Java Gateway.”

If you are using the Servlet Builder feature of VisualAge for Java, you need to add it to the VisualAge for Java repository and workspace. To add the Servlet Builder, select **Quick Start** under the File menu item of the VisualAge for Java workbench window. From here, select **Add Feature**, and select **Servlet Builder** for the feature you require.

You should see a window that looks like Figure 97

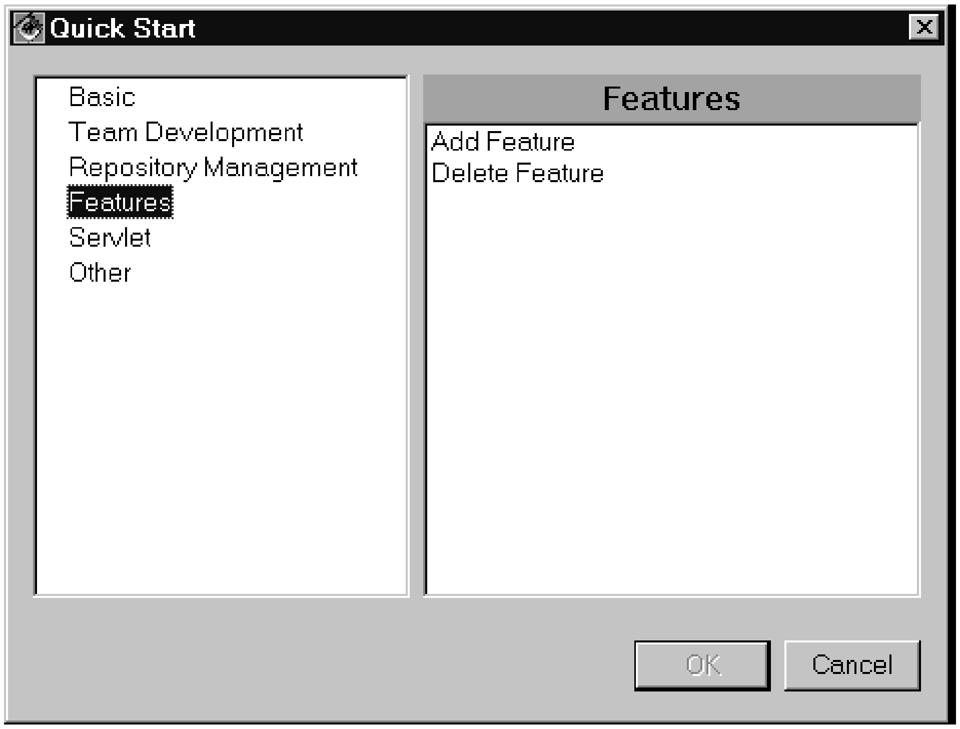


Figure 97. Adding the Servlet Builder in VisualAge for Java

10.5.9 Summary

If you have followed the steps documented here, you should now have a functional development environment which allows you to develop servlets accessing CICS/Cobol programs via CJGW on the workstation and deploy on OS/390.

10.6 OS/390 Setup to Run CICS/DB2 Programs Using CICS Gateway for Java

After you have developed and tested your CICS applications on the Windows NT workstation, you can move them quite easily to the OS/390 CICS environment. Of course, you need to have your CICS Transaction Server and CICS Gateway for Java running on OS/390. In this section we explain how to prepare your environment for running a Java application on OS/390, calling a CICS/DB2/COBOL transaction via the CICS Gateway for Java.

The descriptions are based on our scenario of calling the CICS programs via servlets based on the environment that we set up in ITSO, Poughkeepsie.

The summary of tasks needed to set up the environment to deploy and run our sample servlets for OS/390 are:

- Configuration of the Webserver on OS/390, being either the Lotus Domino Go Webserver Release 5.0 with ServletExpress or WebSphere Application Server for OS/390 V1.1
- Configuration of the CICS Gateway for Java
- Definition of our sample DB2 "manufacturer" table
- Preparation of the CICS COBOL program
- Definition of the CICS entries for COBOL programs
- Translation, compilation, link-edit and bind of the COBOL programs
- Set up of the CICS/DB2 entries for CSMI
- Customization of the Webserver for the sample programs
- Deploying the servlet(s)
- Deploying the HTML and JSP page(s)

10.6.1 Configuring the Webserver

Refer to Chapter 4, "Configuration of the OS/390 Web Server" on page 29 for details about the setup of the Webserver and ServletExpress on OS/390.

10.6.2 Setting Up the CICS Gateway for Java on OS/390

The CICS Gateway for Java on OS/390 uses the OS/390 UNIX System Services component of OS/390 (MVS). You will be working in a UNIX environment when setting up CICS Gateway for Java. Be aware of case-sensitivity. The task summary is as follows:

- Installing/verifying Java on OS/390
- Expanding the CICS Gateway for Java tar file in OS/390 UNIX System Services
- Changing the configuration file
- Setting the WEB=YES SIT Option
- Installing the CICS DFHJAVA Group
- Defining CICS Connections and Sessions
- Modifying the environment variables for CICS Gateway for Java

10.6.2.1 Installing/verifying Java on OS/390

We will assume that you have Java already installed. Refer to 5.1, "JDK Installation and Setup" on page 53 for any further details. You may run `java -fullversion` to see the version of Java on your machine. On our machine, we get the following output:

```
CHORHOC @ SC61:/u/chorhoc>java -fullversion
java full version "JDK 1.1.6 IBM build m116-19981009 Beta 2 (JIT enabled: jitc)"
```

10.6.2.2 Expand CICS Gateway for Java Tar File in OS/390 UNIX System Services

Use the `tar` command to expand the tar file. If you need to, you may download the CICS Gateway for Java tar file from the Internet at URL:

```
http://www.ibm.com/software/ts/cics/platforms/internet/
```

```
tar -xopf jg-11mvs.tar
```

10.6.2.3 Setting the WEB=YES SIT Option

You must specify the WEB=YES SIT option to enable the business logic interface. The CICS Gateway for Java (MVS) uses business logic interface program DFHWBA1. The default for the option is WEB=NO, so YES must be explicitly specified.

Attention

In CICS Transaction Server for OS/390 Version 1.3 (GA), this option cannot be specified anymore.

10.6.2.4 Installing the DFHJAVA Group

To support CICS Gateway for Java (MVS), the DFHJVCVT program definition must be installed in CICS TS. The definition is provided in the CICS group called DFHJAVA.

10.6.2.5 Adding Entries to the DFHCNV Table

Java applets and applications do not execute in an EBCDIC environment, even in the OS/390 Java Virtual Machine. Unless the applet can generate COMMAREA data for the CICS program in the correct format and codepage, a DFHCNV table entry is required for the program. Coding DFHCNV entries is described in *CICS Family: Communicating from CICS on System/390*, SC33-1697. Note that if you use the new CICS Transaction Gateway V3, then VisualAge for Java can generate the translation for you. In our sample scenario, we are using CICS Gateway for Java V2.01.

10.6.2.6 Configuring CICS Connection and Sessions

In order for the OS/390 program to use the EXCI to communicate with CICS TS, definitions for the connection and sessions must be installed. The CICS Gateway for Java (MVS) can use a specific or generic connection. Refer to *CICS TS for OS/390 V1R2 CICS Internet and External Interfaces Guide*, SC33-1944, for a detailed description of how to define EXCI connections and sessions.

Sample group DFH$EXCI contains sample definitions that can be used by the CICS Gateway for Java (MVS). Installing this group creates a generic connection that CICS Gateway for Java (MVS) uses by default. It also creates a specific connection with the netname of BATCHCLI. For simplicity purposes, we recommend that you use the generic connection. You can check that an EXCI connection exists with the CEMT transaction. On our system, this is the output of the CEMT INQUIRE CONNECTION:

```
I CON (EX*)
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Con(EXCG)                  Ins     Irc Exci
 Con(EXCS) Net(BATCHCLI)    Ins     Irc Exci
```

10.6.2.7 Setting Environment Variables

Environment variables are associations of names with values that can be set up independently of programs that access them. Programs can read the values associated with names and act on them accordingly. The *OpenEdition MVS User's Guide*, SC23-3013, contains detailed descriptions of environment variables and how they are set up. Here we describe three ways of setting the variables that the CICS Gateway for Java (MVS) uses.

The Export Command: The OpenEdition export command can be used to set variables before the CICS Gateway for Java (MVS) is started from a shell command line:

```
/u/java/JGate/bin/mvs: >export DFHJVPIPE=JVGATE1
/u/java/JGate/bin/mvs: >export DFHJVSYSTEM_00="SCSCPAA9-ITSO System TS 1.2"
/u/java/JGate/bin/mvs: >export DFHJVSYSTEM_01="BRANCH-Main branch server"
/u/java/JGate/bin/mvs: >JGate
CICS Gateway for Java, Version 1.1.3, 29H0948.
(C) Copyright IBM Corporation 1996. All rights reserved.
CCL6501I: Starting the CICS Gateway for Java with user specified values.
CCL6502I: [ Port = 2006 , Initial Connections = 1 , Maximum Connections = 100
CCL6502I: Initial Workers = 1 , Maximum Workers = 100 ]
CCL6505I: Successfully created the initial Connection and Worker threads.
```

Using Export in the JGate Script: To save having to type in the export commands each time you start the CICS Gateway for Java (MVS), you can place the statements in the JGate script file before the Java Virtual Machine is started. The following shows the last few lines in the script file when some export commands are included.

```
export     STEPLIB=${STEPLIB}:${EXCI_OPTIONS}:${EXCI_LOADLIB}
export     DFHJVPIPE=JVGATE1
export     DFHJVSYSTEM_00="SCSCPAA9-ITSO System TS 1.2"
export     DFHJVSYSTEM_01="BRANCH-Main branch server"
#
#     Start JGate
#
java     ibm.cics.jgate.server.JGate $*
```

Using the STDENV DD Name in the Startup JCL: In the JCL that runs the BPXBATCH program, you can code a DD card with the name STDENV. This name can refer to inline data or a file that contains the name-value pairs for the environment variables for the program that BPXBATCH starts. Here is how some variables are coded in the JCL:

```
//STDENV DD *
DFHJVSYSTEM_00=SCSCPAA9-ITSO System TS 1.2
DFHJVSYSTEM_01=BRANCH-Main branch server
/*
```

Quotes in Environment Variable Values: When variables are defined with the export command, either on the command line or in the shell script, the value may need to be surrounded by double quotes. If the value contains spaces or special characters, double quotes are needed. In the example, discussed in "The Export Command," the DFHJVSYSTEM_nn variables need double quotes because of the spaces, but DFHJVPIPE does not need double quotes.

In the example given in "Using the STDENV DD Name in the Startup JCL" double quotes are not needed because the variables are set in JCL, not the script file.

Overriding Variables: The variables that take effect when the CICS Gateway for Java (MVS) is started are the final variables to be set. So, if JCL is used to start the JGate script, any variables that are set in the script will override the variables of the same name in the JCL.

Environment Variables Used by the CICS Gateway for Java (MVS): The CICS Gateway for Java (MVS) reads environment variables to obtain its customization options. The two options controlled by the variables are whether to use a specific or generic EXCI connection, and which values to return for an `ECIRequest.listSystems` call. Two variables affect the operation of the CICS Gateway for Java (MVS):

1. DFHJVPIPE

   In order for the CICS Gateway for Java (MVS) to use a specific EXCI connection, the DFHJVPIPE must be set to the netname specified in the connection definition. If you are using the EXCI sample definitions in the DFH$EXCI group, a specific connection called EXCS is installed. In order for the CICS Gateway for Java (MVS) to use this connection, you must set the value of DFHJVPIPE to BATCHCLI before starting up the gateway. If DFHJVPIPE's value is left unset, the CICS Gateway for Java (MVS) uses the generic connection defined to CICS.

2. DFHJVSYSTEM_nn

   You can set up to 100 variables of this form, with nn ranging from 00 to 99. The values are the names and descriptions of CICS systems to be returned in response to an `ECIRequest.listSystems call`. The value must be in the form of a string containing the name of a system, followed by a hyphen and then its description. For example:

   ```
   /u/java: >export DFHJVSYSTEM_00="SCSCPAA9-ITSO System TS 1.2"
   /u/java: >export DFHJVSYSTEM_01="BRANCH-Main branch server"
   ```

If you start up the CICS Gateway for Java (MVS) after issuing these commands and use the TestECI program, here is the output you get:

```
TestECI - simple test of CICS Gateway for Java functionality
=== Test Parameters ===
CICS Gateway : wtsc52.itso.ibm.com:3006
ECI Server : null
ECI UserId : null
ECI Password : null
No of programs given : 0
=== Connect to Gateway ===
Successfully created JavaGateway
=== Available Servers ===
System : SCSCPAA9, Description : ITSO System TS 1.2
System : BRANCH, Description : Main branch server
Successfully closed JavaGateway
```

Important

In regard to system ID, note the following:

- On MVS, the terms *system* and *server* refer to the CICS APPLID.
- On NT or AIX, the terms *system* and *server* refer to the TCP/IP address of the CICS region.

10.6.3 Running the CICS Gateway for Java (MVS)

The CICS Gateway for Java (MVS) can be started either from an OpenEdition shell prompt as an "MVS started task," or by submitting JCL that runs the BPXBATCH program. BPXBATCH is an OS/390-supplied program that executes OpenEdition programs and shell scripts that reside in the HFS.

The following JCL worked fine for us when we started the gateway as a started task on OS/390.

```
//JGATE PROC
//CICS   EXEC PGM=BPXBATCH,
//       PARM='SH /u/java/JGate/bin/mvs/JGate -noinput
//             -trace'
//STDOUT     DD  PATH='/tmp/stdout',
//            PATHOPTS=(OWRONLY,OCREAT),
//            PATHMODE=SIRWXU
//STDERR     DD  PATH='/tmp/stderr',
//            PATHOPTS=(OWRONLY,OCREAT),
//            PATHMODE=SIRWXU
//
```

We successfully tested with the following JCL to start our CICS Gateway for Java as a batch job. When the PARM parameter exceeds one line, then there must be a continuation character on column 78. Failure to do so will result in the batch job ending prematurely.

```
//BPXBATA   JOB (999,POK),'L06R',CLASS=A,REGION=4096K,
//             MSGCLASS=T,TIME=10,MSGLEVEL=(1,1),NOTIFY=&SYSUID
//CICS    EXEC PGM=BPXBATCH,
//        PARM='SH /u/java/JGate/bin/mvs/JGate -noinput              -
//             -trace'
//*23456789012345678901234567890123456789012345678901234567890123456789012345678
//STDIN  DD PATH='/dev/null',
//          PATHOPTS=(ORDONLY)
//STDOUT DD PATH='/u/chorhoc/stdout.log',
//          PATHOPTS=(OWRONLY,OCREAT),
//          PATHMODE=SIRWXU
//STDERR DD PATH='/u/chorhoc/stderr.log',
//          PATHOPTS=(OWRONLY,OCREAT),
//          PATHMODE=SIRWXU
```

It is assumed that the CICS Gateway for Java (MVS) is installed in the `/u/java/JGate` directory.

Note that BPXBATCH does not support MVS files for its standard output and standard error log, so you must specify HFS files. To view these files, use the TSO `oedit` command or enter an OpenEdition command shell and use the OpenEdition.

10.6.3.1 Set Up Java Gateway Daemon as MVS Started Task

It is possible to run the CICS Gateway for Java using BPXBATCH as an MVS started procedure:

1. Build the BPXBATCH JCL in your PROCLIB:

```
//JGCICS PROC
//CICS   EXEC PGM=BPXBATCH,
//       PARM='SH /u/java/JGate/bin/mvs/JGate -noinput
//             -trace'
//STDOUT     DD  PATH='/tmp/jgcicsstdout',
//            PATHOPTS=(OWRONLY,OCREAT),
//            PATHMODE=SIRWXU
//STDERR     DD  PATH='/tmp/jgcicsstderr',
//            PATHOPTS=(OWRONLY,OCREAT),
//            PATHMODE=SIRWXU
//
```

2. Associate the JGCICS procedure with RACF.

   The JGCICS procedure must be associated with a RACF user ID that also has a valid OMVS segment. The following command can be used to associate the JGCICS procedure with a RACF user ID called CICSGAT.

   ```
   RDEFINE STARTED JGCICS.* STDATA(USER(CICSGAT) GROUP(CICS)
   PRIVILEGED(NO) TRUSTED(NO) TRACE(NO)
   ```

3. Define PATH, JAVA_HOME and CLASSPATH environment variables.

   It is important that the RACF user ID associated with the JGCICS procedure has a valid $HOME/.profile that exports the following environment variables:

   export PATH=/usr/lpp/java14/J1.1/bin

   export JAVA_HOME=/usr/lpp/java14/J1.1

   export CLASSPATH=/usr/lpp/java14/J1.1/lib/classes.zip

   export _BPX_SHAREAS=YES (optional)

   export _BPX_SPAWN_SCRIPT=YES (optional)

   Note: These values assume a certain root directory for the JDK. In your case, you should use the directory where you actually installed the JDK.

4. Customize the `/usr/lpp/jgCICS/JGate/bin/mvs/JGate` shell script and eventually place it in the desired directory.

10.6.3.2 Use the Local Gateway Function

You can use the Java classes that comprise the CICS Gateway for Java (MVS) in your own Java applications. The local gateway function allows your MVS Java application to access CICS without the need to explicitly run the CICS Gateway for Java, because the Java program has direct access to the CICS TS server. The programming interface used by Java applications is the same as the applet interface. When used in this manner, the gateway TCP/IP address is local. The CJGW also provides the “auto” option whereby the CJGW can decide whether it is local or remote. This is the mode we will use in our program.

Before your application can run, you must set up the environment variables in JGate script. The variables are:

CLASSPATH	This contains the Java classes that are packaged with the CICS Gateway for Java.
LIBPATH	This contains the executables (DLLs).
LD_LIBRARY_PATH	This contains the executables (DLLs).
STEPLIB	This contains the CICS load datasets.

Assuming that you have installed the CICS Gateway for Java (MVS) in the /u/java/JGate directory, and that you are using CICS Transaction Server Version 1.3, use the following commands to set the variables before running your Java application:

```
    export CLASSPATH=$CLASSPATH:/u/java/JGate/classes
    export LIBPATH=$LIBPATH:/u/java/JGate/bin/mvs
    export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/u/java/JGate/bin/mvs
    export STEPLIB=$STEPLIB:CICSTS13.CICS.SDFHEXCI:CICSTS13.CICS.SDFHLOAD
```

Note: Again, we assume in the example that the gateway is installed in /u/java/JGate. In your case, you should use the directory where you actually installed your gateway.

10.6.3.3 Using trace

The CICS Gateway for Java trace option can record information related to the information passing between the browser and the Gateway.

To route messages and trace information to a file, specify “2> trc001” when starting the Gateway (where trc001 is the name of your output file).

The CICS for Java Gateway messages have a CCL prefix. This is the same prefix as that used by CICS Clients.

10.6.3.4 Manufacturer Table Setup

The table and index definition is supplied on the disk that accompanies this redbook. Some sample data is included. You may use SPUFI to create the table and index.

10.6.4 Prepare the CICS COBOL Programs

The supplied sample program is called MANUFACT. The program needs to be precompiled, compiled, link-edited and binded in the normal way. The following is a listing of the JCL we used:

```
//CHORHOCB JOB (999,POK),'MANUFACT',CLASS=A,MSGCLASS=T,NOTIFY=&SYSUID
//*******************************************************************
//*  NAME = PRECOMP                                                 *
//*                                                                 *
//*                                                                 *
//* DB2 PRECOMPILE THE COBOL PROGRAM                                *
//*                                                                 *
//*******************************************************************
//*
//*
//PC       EXEC PGM=DSNHPC,
//        PARM='HOST(IBMCOB),XREF,SOURCE,FLAG(I),APOST'
//STEPLIB  DD  DSN=DB2V510.SDSNEXIT,DISP=SHR
//         DD  DSN=DB2V510.SDSNLOAD,DISP=SHR
//DBRMLIB  DD  DSN=DB2V510U.DBRMLIB.DATA(MANUFACT),DISP=SHR
//SYSCIN   DD DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//            SPACE=(800,(500,500))
//SYSLIB   DD DSN=CHORHOC.CICS.COBOL,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSUT1   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
```

```
//SYSIN    DD DSN=CHORHOC.CICS.COBOL(MANUFACT),DISP=SHR
//*
//TRN    EXEC PGM=DFHECP1$,
//       REGION=0K
//STEPLIB  DD  DSN=CICSTS12.CICS.SDFHLOAD,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSPUNCH DD  DSN=&&SYSCIN,
//             DISP=(,PASS),UNIT=SYSALLDA,
//             DCB=BLKSIZE=400,
//             SPACE=(400,(400,100))
//SYSIN    DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
//*
//*********************************************************
//*         COMPILE AND LINK THE COBOL PROGRAM
//*********************************************************
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K,
//       PARM=(MAP,OBJ,RENT,NODYNAM,OPT,
//       LIB,'DATA(31)',LIST,APOST)
//STEPLIB  DD  DSNAME=IGY.V2R1M0.SIGYCOMP,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=3200)
//SYSLIB   DD  DSN=CHORHOC.CICS.COPYBOOK,DISP=SHR
//         DD  DSN=CICSTS12.CICS.SDFHCOB,DISP=SHR
//*        DD  DSN=CICSTS12.CICS.SDFHSAMP,DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSIN    DD  DSN=&&SYSCIN,
//             DISP=(OLD,PASS)
//*
//*KED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=4024K
//LKED     EXEC PGM=IEWL,COND=(8,LT,COBOL),
//    PARM='LIST,XREF,RENT,AMODE=31,RMODE=ANY'
//SYSLIB   DD  DSNAME=CEE.SCEELKED,DISP=SHR
//         DD  DSN=DB2V510.SDSNLOAD,DISP=SHR
//         DD  DSN=CICSTS12.CICS.SDFHLOAD,DISP=SHR
//         DD  DSN=CICSTS12.CICS.SDFHEXCI,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=ITSO.CICS.LOAD,
//             DISP=SHR
//LIB      DD  DSN=CICSTS12.CICS.SDFHLOAD,DISP=SHR
//         DD  DSN=DB2V510.SDSNLOAD,DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSIN DD *
 INCLUDE LIB(DFHECI)
 INCLUDE LIB(DSNCLI)
  NAME MANUFACT(R)
//*
//*
//*********************************************************
```

```
//*         BIND THE PROGRAM
//*************************************************************
//BIND     EXEC PGM=IKJEFT01,COND=((4,LT,PC))
//STEPLIB  DD  DSN=DB2V510.SDSNEXIT,DISP=SHR
//         DD  DSN=DB2V510.SDSNLOAD,DISP=SHR
//DBRMLIB  DD  DSN=DB2V510U.DBRMLIB.DATA(MANUFACT),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN    DD *
   GRANT BIND, EXECUTE ON PLAN PLNMANU TO PUBLIC;
//SYSTSIN  DD *
DSN SYSTEM(DB2H)
BIND PACKAGE (MANUCOLL) MEMBER(MANUFACT) ACT(REP) ISO(CS)
BIND PLAN(PLNMANU)  PKLIST(MANUCOLL.*) ACT(REP) ISO(CS)
END
/*
//*
```

We used a collection called MANUCOLL and a plan called PLNMANU. If you do not have the appropriate authority, you may need to see your DBA to create it for you. Note that the plan only needs to be created once.

10.6.5 CICS Definitions for Our Samples

CICS definitions needed to be set up for the following:

- MANUFACT program
- DB2CONN definition for CICS to talk to DB2
- DB2Entry for the PLNMANU plan
- DB2Tran for the CSMI mirror transaction

We defined our entries in the ITSO group, which consists of:

```
EX GR(ITSO)
ENTER COMMANDS
 NAME     TYPE         GROUP                                  DATE   TIME
 MANUFACL PROGRAM      ITSO                                   98.307 10.43.31
 MANUFACT PROGRAM      ITSO                                   98.309 10.35.51
 MANUMAIN PROGRAM      ITSO                                   98.308 11.17.58
 TIMEZONE PROGRAM      ITSO                                   98.294 11.19.35
 MANU     TRANSACTION  ITSO                                   98.308 10.51.31
 DB2H     DB2CONN      ITSO                                   98.308 09.33.23
 LS3604   DB2ENTRY     ITSO                                   98.308 10.27.33
 CECI     DB2TRAN      ITSO                                   98.309 11.00.25
 CSMI     DB2TRAN      ITSO                                   98.309 11.00.07
```

Note that in this list, the MANU transaction which points to MANUMAIN is used to drive MANUFACT and may be ignored. We used it as part of our debugging only.

TIMEZONE is a simple CICS program that returns the date and time. It may be used to test that your environment is working.

DB2H is our DB2CONN entry that sets up the interface from CICS to DB2. It looks like the following:

```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0530
 CEDA  View DB2Conn( DB2H     )
  DB2Conn        : DB2H
  Group          : ITSO
  DEscription    :
 CONNECTION ATTRIBUTES
  CONnecterror   : Sqlcode           Sqlcode ¦ Abend
  DB2id          : DB2H
  MSGQUEUE1      : CDB2
  MSGQUEUE2      :
  MSGQUEUE3      :
  Nontermrel     : Yes               Yes ¦ No
  PUrgecycle     : 00 , 30           0-59
  SIgnid         : STC
  STANdbymode    : Reconnect         Reconnect ¦ Connect ¦ Noconnect
  STATsqueue     : CDB2
  TCblimit       : 0012              4-2000
  THREADError    : N906D             N906D ¦ N906 ¦ Abend
 POOL THREAD ATTRIBUTES
....
```

DB2id must point to your DB2 subsystem, which in our case is DB2H. To verify that your CICS/DB2 interface is working, you could issue the following command:

```
dsnc -dis thread(*)
```

The output would be as follows:

```
DSNV401I =DB2H DISPLAY THREAD REPORT FOLLOWS -
DSNV402I =DB2H ACTIVE THREADS -
NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
SCSCPAA9 N        3              STC               009A     0
SCSCPAA9 N      257 ENTRMANU0001 CICSUSER          009A     0
SCSCPAA9 T  *    21 COMDDSNC0002 CICSUSER          009A   195
SCSCPAA9 N       10 POOLCSMI0003 CICSUSER          009A     0
SCSCPAA9 N       13 ENTRMANU0004 CICSUSER          009A     0
DISPLAY ACTIVE REPORT COMPLETE
DSN9022I =DB2H DSNVDT '-DIS THREAD' NORMAL COMPLETION
DFHDB2301 11/06/98 14:25:04 SCSCPAA9 DSNC DB2 command complete.
```

The DB2Entry associates the transaction ID with the plan:

```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0530
 CEDA  View DB2Entry( LS3604   )
  DB2Entry       : LS3604
  Group          : ITSO
  DEscription    :
 THREAD SELECTION ATTRIBUTES
  TRansid        : MANU
 THREAD OPERATION ATTRIBUTES
  ACcountrec     : None              None ¦ TXid ¦ TAsk ¦ Uow
  AUTHId         :
  AUTHType       : Userid            Userid ¦ Opid ¦ Group ¦ Sign ¦ TErm
                                     ¦ TX
  DRollback      : Yes               Yes ¦ No
  PLAN           : PLNMANU
  PLANExitname   :
  PRIority       : High              High ¦ Equal ¦ Low
  PROtectnum     : 0002              0-2000
  THREADLimit    : 0010              0-2000
  THREADWait     : Pool              Pool ¦ Yes ¦ No
```

Note that MANU was our driver transaction which was used for testing. It may be ignored.

All calls that come through EXCI execute under the mirror transaction called CSMI. Therefore, this mirror transaction CSMI also needs to be associated with the DB2 plan, which in our case is PLNMANU. This used to be the Resource Control Table (or RCT) and is a mandatory requirement for all DB2 programs.

```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0530
 CEDA  View DB2Tran( CSMI     )
  DB2Tran        : CSMI
  Group          : ITSO
  Description    : FOR CICS MIRROR TX
  Entry          : LS3604
  Transid        : CSMI
```

You may wish to auto install the ITSO group. In our case, we have added ITSO to the list DB2STUFF. DB2STUFF is specified in the CICS SIT file as:

```
GRPLIST=(DFHLIST,PAALIST,TEMPLIST,DB2STUFF),
```

In CICS, the DB2STUFF list contains:

```
EX LIST(DB2*)
ENTER COMMANDS
 NAME     TYPE         LIST                                   DATE   TIME
 ITSO     GROUP        DB2STUFF                               98.308 09.29.07
```

10.6.6 Set Up the Conversion Table DFHCNV

Conversion needs to be done between ASCII and EBCIDIC even though we are running an OS/390 Webserver in the OS/390 UNIX System Services on the same machine. This conversion is done by specifying the conversion requirements in the CICS DFHCNV program. In Figure 98 on page 191, there are two entries, one for the TIMEZONE sample program, and the other for the MANUFACT sample program. In those two samples, only characters are used.

The entries for MANUFACT and TIMEZONE programs that we used are highlighted.

```
//DFHCNV1  JOB (999,POK),NOTIFY=&SYSUID,
//        CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),TIME=1440
//   EXEC DFHAUPLE,LNKED=IEWL,ASMBLR=ASMA90
//ASSEM.SYSUT1 DD *
         DFHCNV TYPE=INITIAL
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=DFHWBHH,USREXIT=NO,          X
               SRVERCP=037,CLINTCP=437
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=32767,   X
               LAST=YES
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=DFHWBUD,USREXIT=NO,          X
               SRVERCP=037,CLINTCP=437
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=32767,   X
               LAST=YES
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=AIRLIST,USREXIT=NO,          X
               SRVERCP=037,CLINTCP=437
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=BINARY,DATALEN=4
         DFHCNV TYPE=FIELD,OFFSET=4,DATATYP=CHARACTER,DATALEN=861,     X
               LAST=YES
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=TIMEZONE,USREXIT=NO,         X
              SRVERCP=037,CLINTCP=437
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=16,      X
               LAST=YES
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=MANUFACT,USREXIT=NO,         X
               SRVERCP=037,CLINTCP=437
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=271,     X
               LAST=YES
         DFHCNV TYPE=FINAL
         END
/*
//LNKEDT.SYSLMOD DD DSN=ITSO.CICS.LOAD,DISP=SHR
```

Figure 98. Example of DFHCNV

The resulting module must be placed in a load module that is pointed to by the DFHRPL DD card in your CICS started task job.

10.6.7 Deploy HTML and JavaServer Pages

You need to put your HTML, JSP and GIF files in a directory that is accessible as defined by your Webserver. For example, in our case, we put it in /u/chorhoc, and we have an entry in our httpd.conf file that looks like:

```
    Pass            /*                  /u/chorhoc/*
```

In our case, to test an HTML page, we would type in:

```
    http://wtsc61oe/manufact.html
```

10.6.8 Deploy Servlets

The servlets need to be put in the default servlet directory of the Webserver to be used. Different Webservers may have different conventions for placing servlets. Refer to 4.2.2, “WebAS Properties Files” on page 38 for details about specifying your default servlet directories. In our case the default servlet directory is:

```
/usr/lpp/ServletExpress/servlets
```

In the case of a package, the servlet may reside in a subdirectory of the default servlets directory. For example, if you have a servlet class in a package called `itsorb.MfServletMain`, then the class `MfServletMain` should be in the directory `/usr/lpp/ServletExpress/servlets/itsorb`. Alternatively, if it is packaged in a zip file called itsorb.zip, then itsorb.zip should be in the `/usr/lpp/ServletExpress/servlets directory`.

On OS/390, verify that your Webserver is correctly set up to support servlets by testing the SnoopServlet. The SnoopServlet can be activated by typing the following in your browser: `http://yourwebservertcpip/servlet/SnoopServlet`:

The output must look as follows:

```
Requested URL:
http://wtsc61oe/servlet/SnoopServlet
Request information:

 Request method: GET
 Request URI: /servlet/SnoopServlet
 Request protocol: HTTP/1.1
 Servlet path: /servlet/SnoopServlet
 Path info: <none>
 Path translated: <none>
 Query string: <none>
 Content length: <none>
 Content type: <none>
 Server name: wtsc61oe
 Server port: 80
 Remote user: <none>
 Remote address: 9.12.14.71
 Remote host: 9.12.14.71
 Authorization scheme: <none>

Request headers:

 Connection: Keep-Alive
 User-Agent: Mozilla/4.07 ⅛en‘ (WinNT; I)
 Host: wtsc61oe
 Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
 Accept-Encoding: gzip
 Accept-Language: en
 Accept-Charset: iso-8859-1,*,utf-8
```

10.6.9 Deploy Beans

The directory containing beans should be in your CLASSPATH variable. In our case, we put it in the same directory as the servlets and point the CLASSPATH to it. Packages must be a subdirectory of any of the paths pointed to by the CLASSPATH environment variable.

To verify your CLASSPATH, issue:

```
echo $CLASSPATH
```

To modify your CLASSPATH, use the `EXPORT` command:

```
EXPORT CLASSPATH=/usr/lpp/ServletExpress/servlets/itsorb.zip:${CLASSPATH}
```

This `EXPORT` command will pre-append the /usr/lpp/ServletExpress/servlets/itsorb.zip file to the current CLASSPATH.

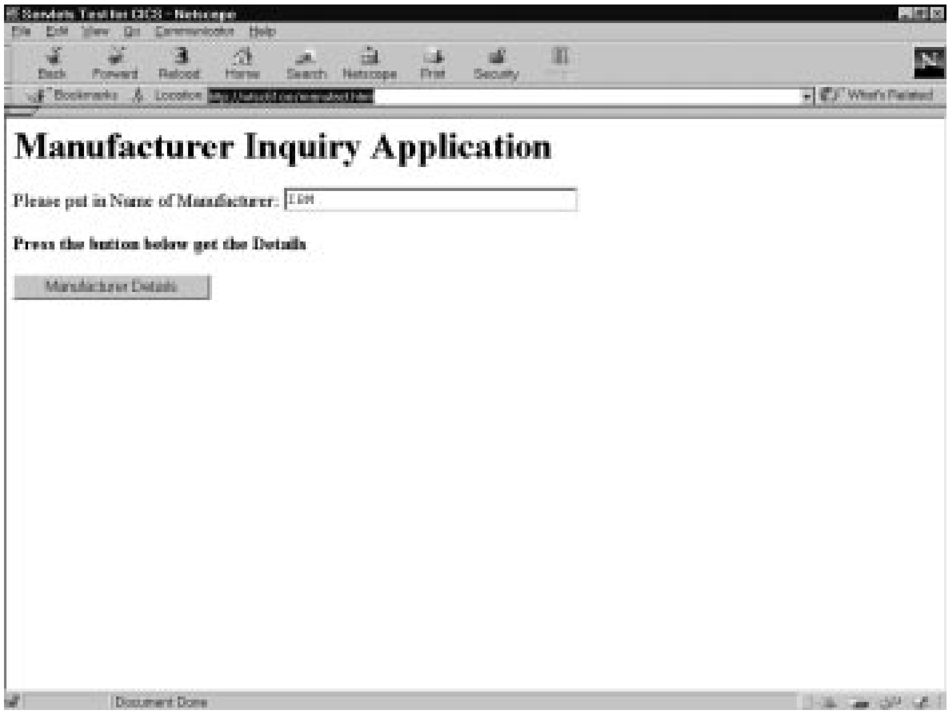
10.6.10 Run the Internet MANUFACTURER Application

First, update the gateway settings to reflect your environment. Figure 99 shows an extract from the manufact.jsp file. Edit the highlighted lines to reflect your environment before running it.

```
<% // retrieve the manufacturer variable from the previous form
String manuName = request.getParameter("manuName");
// set the manufacturer's name for retrieval by the CICS program
mfBean.setManufactName(manuName);
// call the CICS program which will populate the attributes of mfBean
// but first set up your Gateway properties
 itsorb.JGateSettings jg = new itsorb.JGateSettings();

 jg.setCicsProgram("MANUFACT");
 jg.setCicsServer("SCSCPAA9");
 jg.setCicsTranid("MANU");
 jg.setCicsJGate("auto://wtsc58oe.ITSO.IBM.COM");
 jg.setCicsUserid("CICSUSER");

 jg.setCommareaLength(271);
 mfBean.callCICS(jg);
%>
```

Figure 99. Extract from manufact.jsp

Run the application by pointing the browser to the location of your HTML page. Your output should look like the output shown in Figure 100 on page 194.



Servlets Test for CICS - Netscape

Manufacturer Inquiry Application

Please put in Name of Manufacturer: IBM

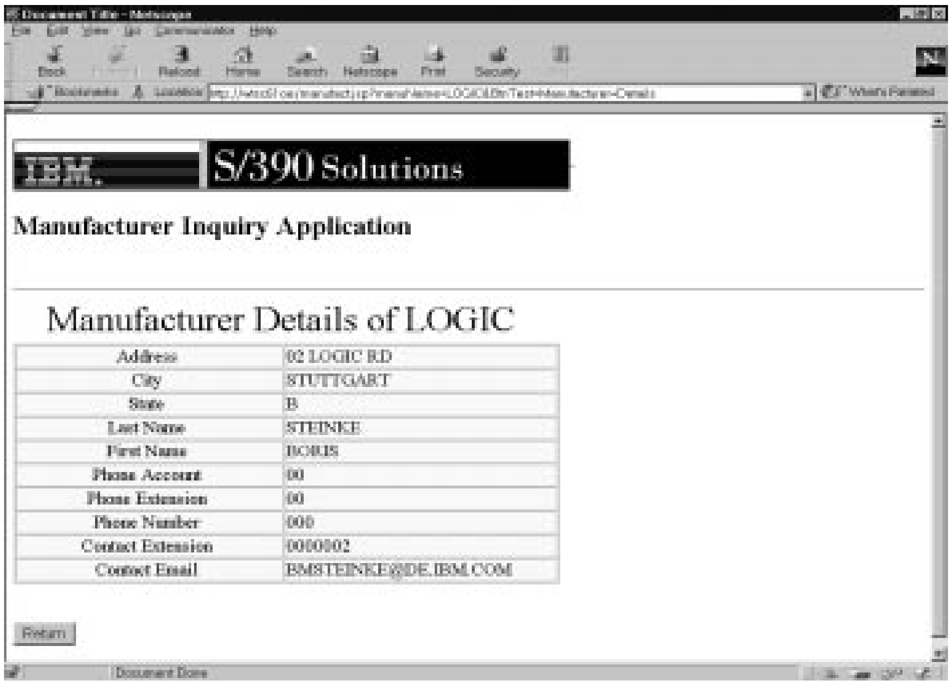
Press the button below get the Details

Manufacturer Details

Document Done

Figure 100. Manufacturer Application Screen

Typing in a manufacturer that exists, such as “Logic,” will give you a result screen as shown in Figure 101.



Document Title - Netscape

IBM. S/390 Solutions

Manufacturer Inquiry Application

Manufacturer Details of LOGIC

Address	02 LOGIC RD
City	STUTTGART
State	B
Last Name	STEINKE
First Name	BORIS
Phone Account	00
Phone Extension	00
Phone Number	000
Contact Extension	0000002
Contact Email	BMSTEINKE@DE.IBM.COM

Return

Document Done

Figure 101. Manufacturer Application Result Screen

If you do not have a DB2 environment set up, you could try using CICS Gateway for Java with TIMEZONE sample.

10.6.11 Running the Internet TIMEZONE Application

First, update the gateway settings to reflect your environment. Figure 102 shows an extract from the timezone.jsp file. Edit the highlighted lines to reflect your environment before running it.

```
<% // retrieve the fullname variable from the previous form
String fullName = request.getParameter("fullName");
// call the CICS program which will return the Time and Date
// but first set up your Gateway properties
 itsorb.JGateSettings jg = new itsorb.JGateSettings();
 jg.setCicsProgram("TIMEZONE");

 jg.setCicsServer("SCSCPAA9");
 jg.setCicsTranid("MANU");
 jg.setCicsJGate("auto://wtsc58oe.ITSO.IBM.COM");
 jg.setCicsUserid("CICSUSER");
 jg.setCommareaLength(16);

 timeZoneBean.callCICS(jg);
%>
```

Figure 102. Extract from timezone.jsp

Run the application by pointing the browser to the location of your HTML page.

Attention

This sample application is provided to show the code and definitions required to Web-enable your CICS applications using CJGW. They have not been written to take care of exception situations, security concerns or to show good coding practices. They are not production-ready.

10.6.12 Problem Resolution

If you run into problems in running the provided sample, you could try running the Manufacturer application as a CICS transaction initially.

10.6.12.1 Testing Your CICS Setup

On your CICS terminal, turn CEDF on, clear the screen and type in `MANU`. You should be able to ascertain whether the SQL statement successfully retrieved the IBM record. The MANUFACT program will by default attempt to retrieve the IBM record. Note that the MANUMAIN program has no screens. Its only purpose in life is to drive the MANUFACT program.

If successful, this validates your CICS and DB2 setup.

10.6.12.2 Testing Your Java to CICS/DB2 Setup

The next step is to test the Java program with CICS/DB2 via local access to CICS Gateway for Java As it is local, the CICS Gateway for Java daemon or started task does not have to be up. The MANUFACT program may also be run as a Java application. It will optionally take one parameter, which will be the manufacturer's name. Otherwise, it defaults to “IBM.”

Before you test, make sure your CLASSPATH and your STEPLIB environment variables are correctly set. In our case, our CLASSPATH is:

```
CHORHOC @ SC61:/u/chorhoc>echo $CLASSPATH
/usr/lpp/ServletExpress/servlets:/usr/lpp/jgCICS/JGate/classes:/usr/lpp/java16/
J1.1/lib/classes.zip:/usr/lpp/internet/server_root/cgi-bin/icsclass.zip:/usr/lpp
/db2/db2510/classes/db2jdbcclasses.zip:/usr/lpp/cicsts/cicsts13/classes/dfjcics.
jar:.:
```

Our MANUFACT class sits in the itsorb subdirectory of `/usr/lpp/ServletExpress/servlets`

If your CLASSPATH is not correct, you will get a `class not found` error.

Our STEPLIB is:

```
CHORHOC @ SC61:/u/chorhoc>echo $STEPLIB
CICSTS13.CICS.SDFHEXCI:CICSTS13.CICS.SDFHLOAD
CHORHOC @ SC61:/u/chorhoc>
```

If your STEPLIB is not set up correctly, you will get a -806 (`module not found`) error.

If you run the MANUFACT program successfully, you should get the following output:

```
CHORHOC @ SC61:/u/chorhoc>java itsorb.CicsManufact LOGIC
Successfully created JavaGateway Object
After flow Commarea      : 0LOGIC    RECORD FOUND         LOGIC
        02 LOGIC RD                   STUTTGART                    BW00002
STEINKE                      BORIS                        00200200020000002
BMSTEINKE@DE.IBM.COM
Return code   : 0
Abend code    : null
The commarea contains: 0LOGIC    RECORD FOUND         LOGIC
    02 LOGIC RD                   STUTTGART                    BW00002     STEI
NKE                      BORIS                        00200200020000002   BMST
EINKE@DE.IBM.COM
CHORHOC @ SC61:/u/chorhoc>
```

10.6.12.3 Testing Your Webserver

Initially, you may want to verify that your Webserver is configured properly by using the supplied snoop servlet i.e. `http:/<your_hostname>/servlet/snoop`. snoop is a sample servlet that comes with Websphere Application Server.

10.6.12.4 Common Errors

Following is a description of some of the errors that we encountered, and ways to get around these. Note that, by the time you read this book, some of these “error conditions” may not exist anymore.

RACF Access: The most common error that we made was giving insufficient RACF access to files and directories. On our system, files that are transferred via ftp to OS/390, by default, have no access to the public (that is, access via the Web). The following error was encountered:

```
IMW0254E

Error 403

Can't browse selected file.


Lotus Domino Go Webserver - North American Edition for OS/390 V5R0M0
```

You will also be able to see the MVS system log for the RACF rejections.

The directory must also be authorized for reading. Check it with the `ls -al` command. Otherwise, you will end up with the following error:

```
500 Internal Server Error

The servlet named invoker at the requested URL

reported this exception itsorb.MfServletMain: Cannot load  1
local code itsorb.MfServletMain.. Please report this to the  1
administrator of the web server.

    com.sun.server.http.InvokerException: itsorb.MfServletMain: Cannot load  1
    local code itsorb.MfServletMain. at
    com.sun.server.http.InvokerServlet.service(Compiled Code) at  1
    javax.servlet.http.HttpServlet.service(Compiled Code) at
    com.sun.server.ServletState.callService(Compiled Code) at
    com.sun.server.ServletManager.callServletService(Compiled Code) at
    com.sun.server.ProcessingState.invokeTargetServlet(Compiled Code) at
    com.sun.server.http.HttpProcessingState.execute(Compiled Code) at
    com.sun.server.http.stages.Runner.process(Compiled Code) at
    com.sun.server.ProcessingSupport.process(Compiled Code) at  1
    com.sun.server.Service.process(Compiled Code) at
    com.ibm.ServletExpress.service.SELauncher.processMultiThreaded(Compiled Code) at
    com.ibm.ServletExpress.service.SEServlet.service(Compiled Code) at
    com.ibm.ServletExpress.ServletSystemImp.invoke(Compiled Code) at
    com.ibm.ServletExpress.ServletSystem.icsInvoke(Compiled Code)
```

Notes:

> **1** These lines have been split for printing purposes; however, in the real output you would see these lines on one single line.

Case Sensitivity: This cannot be overemphasized. OS/390 UNIX System Services is case-sensitive and many problems occur from getting the case wrong.

Miscellaneous Errors: We find that changing a JSP that has been loaded (that is, compiled) can cause the following error. In our case, this required the Webserver to be recycled.

```
500 Internal Server Error

The servlet named pageCompile at the requested URL

reported this exception java.lang.NullPointerException.  1
Please report this to the administrator of the web server.

    java.lang.NullPointerException at pagecompile._manufact_xjsp.service(Compiled Code) at
    javax.servlet.http.HttpServlet.service(Compiled Code) at
    com.sun.server.http.pagecompile.PageCompileServlet.doService(Compiled Code) at
    com.sun.server.http.pagecompile.PageCompileServlet.doGet(Compiled Code) at
    javax.servlet.http.HttpServlet.service(Compiled Code) at  1
    javax.servlet.http.HttpServlet.service(Compiled Code)
    at com.sun.server.ServletState.callService(Compiled Code) at
    com.sun.server.ServletManager.callServletService(Compiled Code) at
    com.sun.server.ProcessingState.invokeTargetServlet(Compiled Code) at
    com.sun.server.http.HttpProcessingState.execute(Compiled Code) at
    com.sun.server.http.stages.Runner.process(Compiled Code) at
    com.sun.server.ProcessingSupport.process(Compiled Code) at  1
    com.sun.server.Service.process(Compiled Code)
    at com.ibm.ServletExpress.service.SELauncher.processMultiThreaded(Compiled Code) at
    com.ibm.ServletExpress.service.SEServlet.service(Compiled Code) at
    com.ibm.ServletExpress.ServletSystemImp.invoke(Compiled Code) at
    com.ibm.ServletExpress.ServletSystem.icsInvoke(Compiled Code)
```

Notes:

1 These lines have been split for printing purposes; however, in the real output you would see these lines on one single line.

10.7 A Closer Look at our Sample CICS Application

We will begin by taking a high-level view of the flow within the CJGW.

Attention

In the following sections we refer to various pieces of code. You can find the code on the CD shipped with this book. Refer to Appendix A, "CD-ROM" on page 299 for details.

10.7.1 Overview of the CICS Gateway for Java Flow

At the simplest level, the flow of program control needed to write a simple CICS Gateway for Java client program is as follows:

1. The Java program creates and opens an instance of an `ibm.cics.jgate.client.JavaGateway` object. The default `JavaGateway` constructor creates a blank `JavaGateway` object. You must then set the correct properties in this object using the relevant `set` methods. The `JavaGateway` is then opened by calling the `open` method. The resultant `JavaGateway` is open and connected to the requested CICS Gateway for Java.
2. The Java program creates an instance of one of the Gateway request classes containing the request that it wishes to make:
   - An `ibm.cics.jgate.client.ECIRequest` is created for an ECI request.
   - An `ibm.cics.jgate.client.EPIRequest` is created for an EPI request.
   - An `ibm.cics.jgate.client.CicsCpRequest` is created for querying the codepage of the CICS Client it is connected through.
3. The Java program then flows the request to the CICS Gateway for Java using the `flow` method of the `JavaGateway` object.
4. The Java program checks the return code of the flow operation to see whether the request was successful.
5. The Java program then closes the `JavaGateway` object.

Next, we will look at the scenario where we use a HTML (mfhtml) to drive a servlet (MfServletHtml), which calls a bean (Manufact) that interacts with the CICS Gateway for Java classes to call the MANUFACT CICS/COBOL/DB2 program.

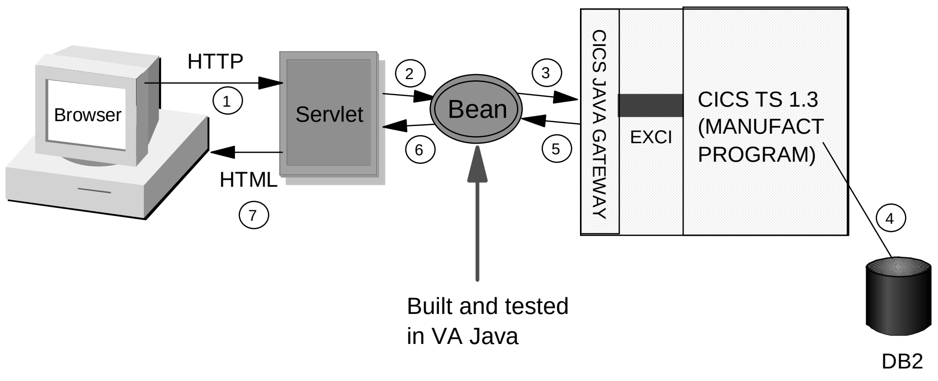


Figure 103. Overview of CICS/DB2 Program Access Via a Servlet

Referring to Figure 103, the steps involved are:

1. The browser/universal client sends an HTTP request to the Webserver.

   The Webserver recognizes the `/servlet/` string in the URL of the “Action” request and calls the servlet.

   If this is the first time that the servlet is requested, the Webserver will need to load the servlet and call its `init()` method. Once the servlet is loaded and compiled, it will remain after the first call, ready to service subsequent invocations. This is why servlets are more efficient than CGI calls, because the thread for the servlet is retained.

2. The servlet's `service()` method is invoked.

   The `service()` method pulls information from the Web page via the request object.

3. The `service` method instantiates the bean (Manufact) and sets its attributes with the data from the form. It then passes control to the bean.

4. The bean sets the parameters and instantiates the JavaGateway object, and similarly, does so for the `ECIRequest` object.

5. The `ECIRequest` object is set to MANUFACT as one of its parameters and calls the MANUFACT program via the EXCI interface. Note that it comes in through the CSMI mirror transaction.

6. The MANUFACT CICS program accesses the DB2 MANUFACT table via normal static SQL calls.

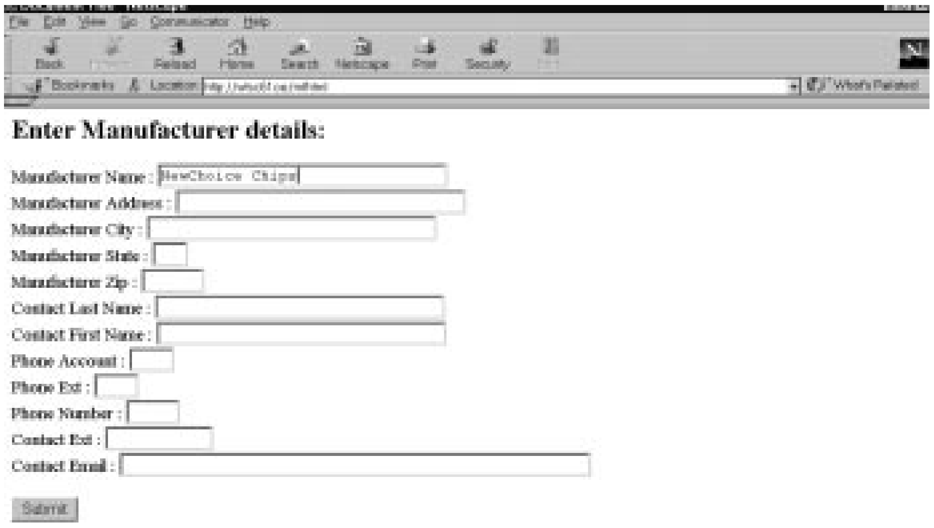
7. The data, when returned to the MANUFACT bean, has been translated to ASCII via an entry in the DFHCNV table.

8. With the values of the table results in the returned COMMAREA, the bean sets the attributes of the bean (itself).

9. The servlet checks the return code, and builds the appropriate HTML response stream. This reply is returned by the Webserver to the browser/universal client.

10.7.2 Looking at the Code for the Create Row Operation

We start by looking at the mfhtml file, which presents a form for users to input the data, as shown in Figure 104.



Enter Manufacturer details:

Manufacture Name :	NewChoice Chips
Manufacture Address :	
Manufacture City :	
Manufacture State :	
Manufacture Zip :	
Contact Last Name :	
Contact First Name :	
Phone Account :	
Phone Ext :	
Phone Number :	
Contact Ext :	
Contact Email :	

Figure 104. Entering Data Via an HTML Form

An extract of the HTML is shown in Figure 105 on page 201.

```
<FORM NAME="MfForm" ACTION="/servlet/itsorb.MfServletHtml" METHOD="get">
Manufacturer Name    : <INPUT TYPE="text" SIZE=30 NAME="mfName">
<BR>
Manufacturer Address : <INPUT TYPE="text" SIZE=30 NAME="mfAddress">
<BR>
Manufacturer City : <INPUT TYPE="text" SIZE=30 NAME="mfCity">

<BR>
Manufacturer State   : <INPUT TYPE="text" SIZE=2 NAME="mfState">
<BR>
Manufacturer Zip     : <INPUT TYPE="text" SIZE=5 NAME="mfZip">
<BR>
Contact Last Name    : <INPUT TYPE="text" SIZE=30 NAME="mfLname">
<BR>
Contact First Name   : <INPUT TYPE="text" SIZE=30 NAME="mfFname">
<BR>
Phone Account        : <INPUT TYPE="text" SIZE=3 NAME="mfPhoneAc">
<BR>
Phone Ext            : <INPUT TYPE="text" SIZE=3 NAME="mfPhoneEx">
<BR>
Phone Number         : <INPUT TYPE="text" SIZE=4 NAME="mfPhoneNr">

<BR>
Contact Ext          : <INPUT TYPE="text" SIZE=10 NAME="mfConExt">
<BR>
Contact Email        : <INPUT TYPE="text" SIZE=50 NAME="mfConEmail">

<BR><BR>

<INPUT TYPE="submit" NAME="mfButton" VALUE="Submit">
```

Figure 105. An Extract of the HTML for Data Entry of Manufacturer Details

The “Action” of the form will fire off the servlet: In the highlighted line in Figure 105, when the HTTP request reaches the Webserver, it will know that this is meant for a servlet called “MfServletHtml” sitting in package `itsorb`. The Webserver calls the MfServletHtml servlet and passes the data entered in the fields. Those fields are identified by the NAME= parameter.

In Figure 106 on page 202 we show the source of the MfServletHtml servlet.

```
public void service(HttpServletRequest req, HttpServletResponse res) throws IOException {
    ServletOutputStream out = res.getOutputStream();
    try {
         Manufact aMan = new Manufact();
         String mfName = req.getParameter("mfName");  1
         if (mfName != null && mfName.length() != 0) {
              res.setContentType("text/html"); //Required for HTTP
              retrieveFields(req, aMan);            2
              //calls the Manufact program via the callCICS method
              aMan.callCICS();                      3
              //write the html output stream
              out.println("<h1>Add operation Results</h1><br><br>");   4
              if (aMan.getReturnCode().equals("0")) {
                   out.println("<h2> Insert successful for Manufacturer : " + mfName + "</h2> <BR>");
              } else {
                   out.println("<h2> Insert failed : RC = " + aMan.getReturnCode() + "</h2>");
                   out.println("<h2> SqlCode is : RC = " + aMan.getSqlCode() + "</h2>");
                   out.println("<h2> Message : " + aMan.getMessage() + "</h2>");
              }
              out.println("<br><br>");
              out.println("<H2>Thank you</H2>");
              out.close();
         } else {
              res.setContentType("text/html"); //Required for HTTP
              out.println("<h2> Error : Manufacturer name cannot be blank </h2>");
              out.println("<br><br>");
              out.println("<H2>Thank you</H2>");
              out.close();
         }
    } catch (Exception e) {
         System.out.println("An error occurred : " + e);
    }
}
```

Figure 106. Code for the Service Method of MfServletHtml

Notes:

Whenever the MfServletHtml servlet is invoked, the `service()` method is called (except for the first time, when the `init()` method is called first). Looking at Figure 106, the steps involved are:

1 Check that the manufacturer's name exists as supplied via the form.

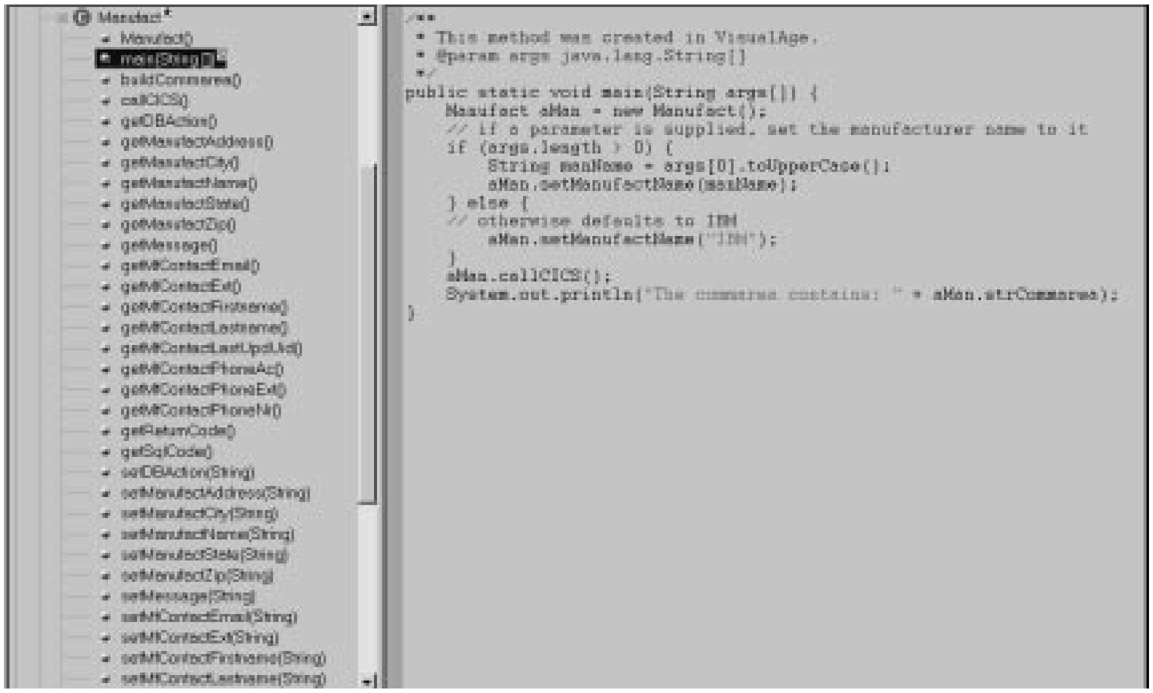
2 If it does, then get the data from the HTML form and set the attributes of the Manufacturer object, which was instantiated earlier.

3 Call the `JavaGateWay` classes to add the data to the Manufacturer table. The Manufacturer object has a method called `callCICS` which acts as the interface to the `CICSConnect` class. It is the `CICSConnect` class which does the direct interface to the `JavaGateWay` classes.

4 Based on the return code, build the appropriate datastream in the response object.

The java code for the CICS samples consists of the classes:

- `Manufact`
- `CICSConnect`



```
Manufact*
    Manufact()
    main(String[])
    buildCommarea()
    callCICS()
    getDBAction()
    getManufactAddress()
    getManufactCity()
    getManufactName()
    getManufactState()
    getManufactZip()
    getMessage()
    getMfContactEmail()
    getMfContactExt()
    getMfContactFirstname()
    getMfContactLastname()
    getMfContactLastUpdUid()
    getMfContactPhoneAc()
    getMfContactPhoneExt()
    getMfContactPhoneNr()
    getReturnCode()
    getSqlCode()
    setDBAction(String)
    setManufactAddress(String)
    setManufactCity(String)
    setManufactName(String)
    setManufactState(String)
    setManufactZip(String)
    setMessage(String)
    setMfContactEmail(String)
    setMfContactExt(String)
    setMfContactFirstname(String)
    setMfContactLastname(String)
```

```
/**
 * This method was created in VisualAge.
 * @param args java.lang.String[]
 */
public static void main(String args[]) {
    Manufact aMan = new Manufact();
    // if a parameter is supplied, set the manufacturer name to it
    if (args.length > 0) {
        String manName = args[0].toUpperCase();
        aMan.setManufactName(manName);
    } else {
    // otherwise defaults to IBM
        aMan.setManufactName("IBM");
    }
    aMan.callCICS();
    System.out.println("The commarea contains: " + aMan.strCommarea);
}
```

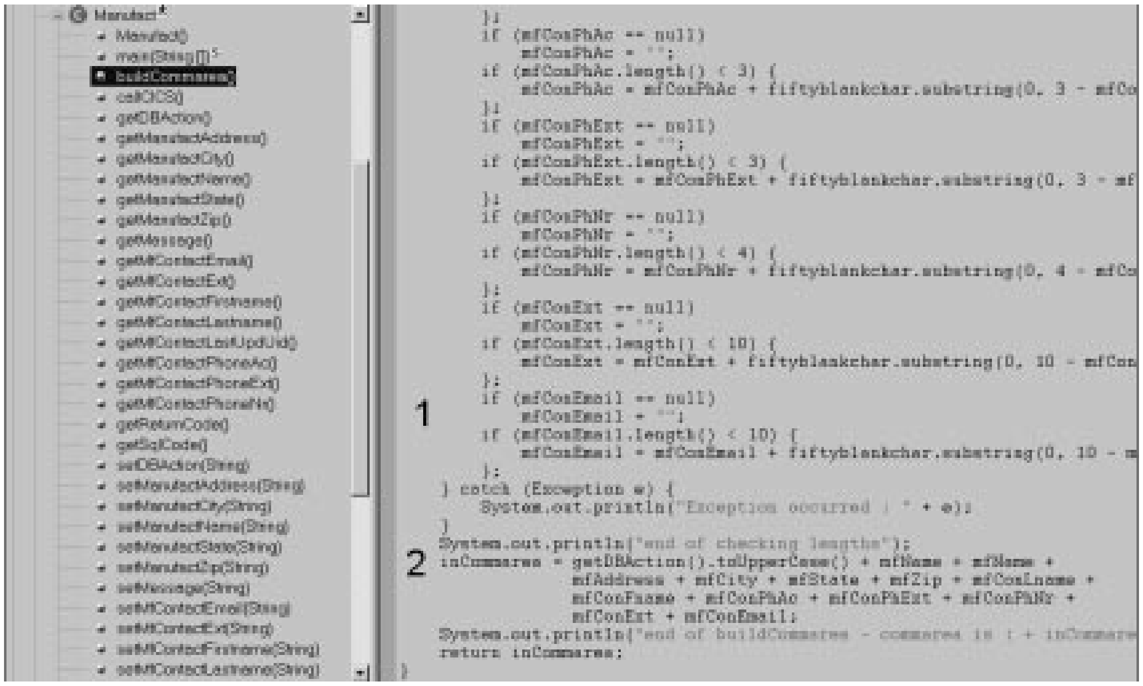
Figure 107. The Manufact Class

Figure 107 shows the Manufact Class. Each of the attributes of this class has getter and setter methods. The `service()` method of the MfServletHtml class would use the setter methods to set the attributes.

Once the attributes are set (in the code shown, the Manufacturer name is set), the call to CICS can be made via a call to `callCICS()` .

Notice that `Manufact` class has a `main()` method. However, this class does not require a main method. It is there to allow testing it as an application outside of the Webserver. It could be run by typing:

```
java itsorb.Manufact
```



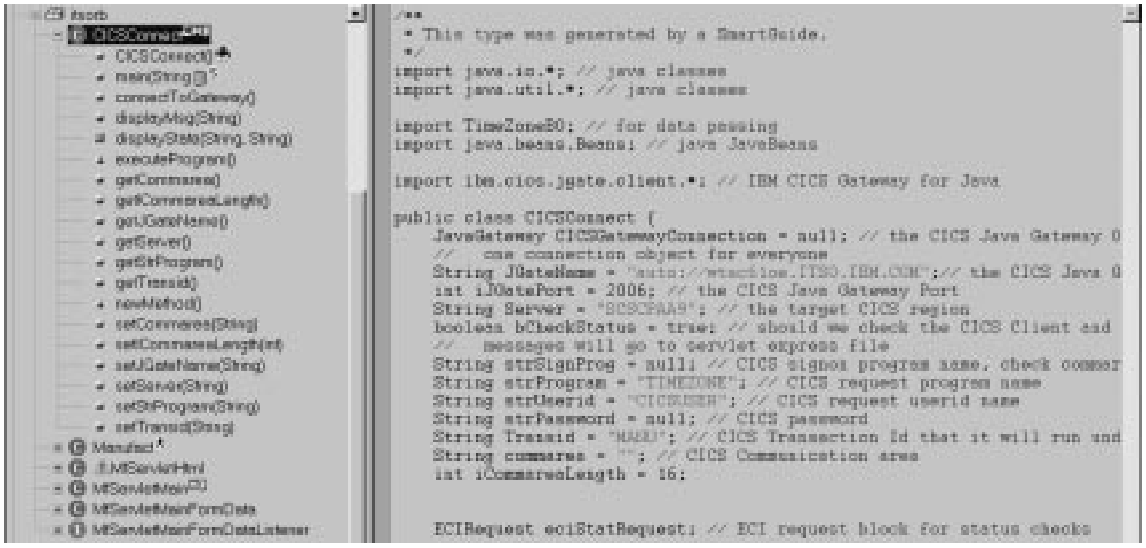
```
        };
        if (mfConPhAc == null)
            mfConPhAc = "";
        if (mfConPhAc.length() < 3) {
            mfConPhAc = mfConPhAc + fiftyblankchar.substring(0, 3 - mfCo
        };
        if (mfConPhExt == null)
            mfConPhExt = "";
        if (mfConPhExt.length() < 3) {
            mfConPhExt = mfConPhExt + fiftyblankchar.substring(0, 3 - mf
        };
        if (mfConPhNr == null)
            mfConPhNr = "";
        if (mfConPhNr.length() < 4) {
            mfConPhNr = mfConPhNr + fiftyblankchar.substring(0, 4 - mfCo
        };
        if (mfConExt == null)
            mfConExt = "";
        if (mfConExt.length() < 10) {
            mfConExt = mfConExt + fiftyblankchar.substring(0, 10 - mfCon
        };
1       if (mfConEmail == null)
            mfConEmail = "";
        if (mfConEmail.length() < 10) {
            mfConEmail = mfConEmail + fiftyblankchar.substring(0, 10 - m
        };
    } catch (Exception e) {
        System.out.println("Exception occurred : " + e);
    }
    System.out.println("end of checking lengths");
2   inCommarea = getDBAction().toUpperCase() + mfName + mfName +
                 mfAddress + mfCity + mfState + mfZip + mfConLname +
                 mfConFname + mfConPhAc + mfConPhExt + mfConPhNr +
                 mfConExt + mfConEmail;
    System.out.println("end of buildCommarea - commarea is : + inCommare
    return inCommarea;
}
```

Figure 108. Extract of the buildCommarea Method

In Figure 108, the code extract shows how the COMMAREA is built. In part 1, each of the fields of the COMMAREA is replaced by an empty string, if it is null. Then, each of the field is padded to its full length.

Note: This would not be an efficient approach for long variable length fields.

In part 2, the fields are concatenated together, thus completing the build of the COMMAREA. With the build complete, the next step is to call the `CICSConnect` class.



```
/**
 * This type was generated by a SmartGuide.
 */
import java.io.*; // java classes
import java.util.*; // java classes

import TimeZoneBO; // for data passing
import java.beans.Beans; // java JavaBeans

import ibm.cics.jgate.client.*; // IBM CICS Gateway for Java

public class CICSConnect {
    JavaGateway CICSGatewayConnection = null; // the CICS Java Gateway O
    //    one connection object for everyone
    String JGateName = "auto://wtsc61oe.ITSO.IBM.COM";// the CICS Java G
    int iJGatePort = 2006; // the CICS Java Gateway Port
    String Server = "SCSCPAA9"; // the target CICS region
    boolean bCheckStatus = true; // should we check the CICS Client and
    //    messages will go to servlet express file
    String strSignProg = null; // CICS signon program name, check commar
    String strProgram = "TIMEZONE"; // CICS request program name
    String strUserid = "CICSUSER"; // CICS request userid name
    String strPassword = null; // CICS password
    String Transid = "MAEU"; // CICS Transaction Id that it will run und
    String commarea = ""; // CICS Communication area
    int iCommareaLength = 16;


    ECIRequest eciStatRequest; // ECI request block for status checks
```

Figure 109. The CICSConnect Class

Figure 109 shows the `CICSConnect` class. The class has default values set when instantiated. However, since each of this attributes have getters and setters, they can be reset dynamically if required. The attribute `JGateName` is set to "auto://<your_hostname>." This allows the gateway to access the CICS Gateway for Java classes either locally or remotely. In our case, it will always be local, since we are using servlets running on the same machine.

This class has two main methods which are the `connectToGateway()` and the `executeProgram()` methods. The rest are getters and setters for the attributes.

We have designed the MANUFACT program as a bean to make it reusable between the different approaches, such as JSPs, servlets and applets. Let us start by looking at the JSP solution.

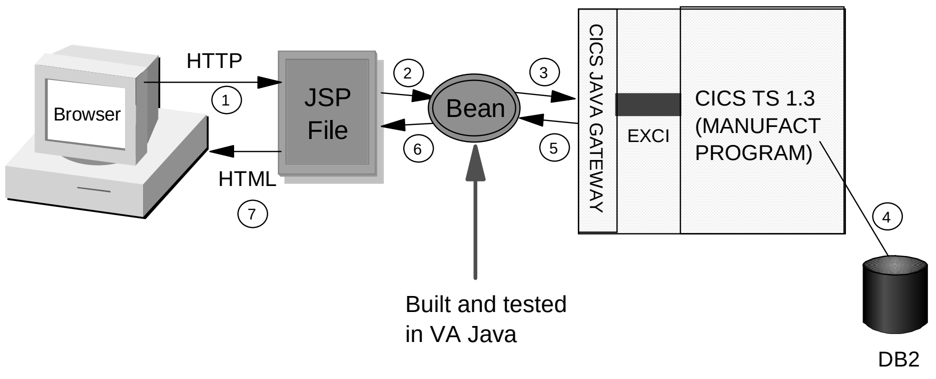


Figure 110. Overview of CICS/DB2 Program Access Via a JSP

Referring to Figure 110, the steps involved are:

1. The browser/universal client sends an HTTP request to the Webserver. The Webserver recognizes the .jsp extension in the "Action" request and loads the .jsp page. If this is the first time that the .jsp page is referenced, it will need to compile it first. You will notice the delay in response as ServletExpress compiles the JSP into bytecode. This bytecode is done once, and thereafter, the JSP (in the form of a compiled servlet) is stored in cache.
2. The .jsp servlet loads the bean (MANUFACT) as specified in the .jsp page and reads the input parameter(s) from the `request` object. The .jsp servlet then passes it on to the bean, and calls the appropriate method of the bean.
3. The bean sets the parameters and instantiates the `JavaGateway` object, and similarly, does so for the `ECIRequest` object.
4. The `ECIRequest` object is set to MANUFACT as one of its parameters and calls the MANUFACT program via the EXCI interface. Note that it comes in through the CSMI mirror transaction.
5. The MANUFACT CICS program accesses the MANUFACT table via normal static SQL calls.
6. The data, when returned to the MANUFACT bean, has been translated to ASCII via an entry in the DFHCNV table.
7. With the values of the table results in the returned commarea, the bean sets the attributes of the bean.

8. The .jsp servlet accesses the attributes of the bean via the getter methods. The .jsp servlet then builds the HTML stream in the response object, which is returned by the Webserver to the browser/universal client.

Chapter 11. Accessing IMS Transactions from the Web

In this chapter we provide an overview of the various ways of accessing IMS transactions from the Web. The information in the following sections is based on samples we created during ITSO residencies.

We focus on solutions using server-side logic responsible for the connection to the IMS back-end system. We refer you to *Integrating Java with Existing Data and Applications*, SG24-5142, for solutions in which the client directly accesses IMS, without the usage of server-side logic to make the connection.

11.1 Introduction

In this section we cover general considerations and possible pitfalls when we want to use existing or new IMS transactions from a browser or an applet loaded in the HTML page.

IMS is composed of two components:

1. The Transaction Manager

   This is responsible for accepting and scheduling the transactions arriving from the terminals. Most of the current transactions are written for 3270 terminals. This part is now often referred to as the DCCTL (Data Communication Control).

2. The Database Manager

   This allows access to the DLI/FP databases. This part is now often referenced as the DBCTL (Data Base Control).

Besides access to DL/I hierarchical databases, IMS transactions can also share access to DB2 relational databases and MQI queues with other TMs (IMS, CICS, or even batch). Access to those resources is coordinated through a "two-phase commit" syncpoint protocol.

Most of the transactions have been written for 3270 in languages like COBOL, PL/I, REXX and C, and most of the time the development was done without paying special attention to the 3270 specifics. This is possible because of a component called Message Format Services (MFS), which can isolate the logical message (on the program side) from the physical message (on the presentation (device) side).

The role of MFS is to add presentation control characters on the "outbound" message, and to rebuild the logical "inbound" message with the elements of the physical message from the device. MFS fulfills a lot of functions which can be described in one word: "mapping." For further details about MFS, refer to the IMS documentation.

MFS is only used for particular devices (in principle, 3270). In other cases, mostly for programmed clients, MFS is bypassed. This implies that, in case programs that have been written specifically for 3270 are used by browsers, the logical message as required by the Message Processing Program (MPP) has to be built by the client function.

On output, the message sent by the program is passed directly to the client. Eventually we can act on it by using the IMS "DFSLUEE0" exit. This routine enables you to edit input and output LU 6.2 messages for IMS-managed LU 6.2 conversations.

Figure 111 illustrates the difference between a traditional 3270 transaction using MFS and a Web-based scenario using either APPC or OTMA as the IMS communication protocol.

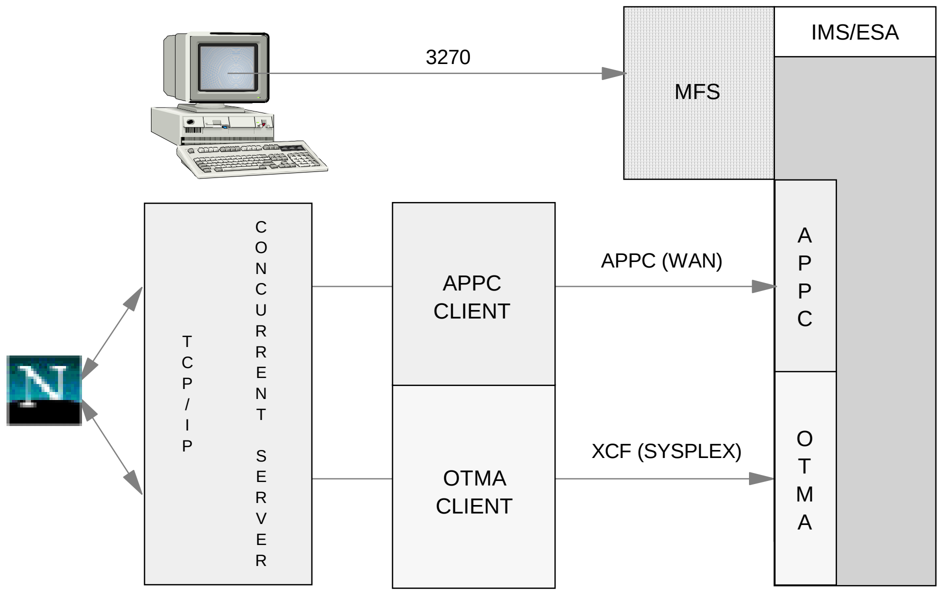


Figure 111. 3270/MFS versus Web-Based

In this book we are interested in using the transactions from a browser. This will always happen via an intermediary gateway. IMS can be accessed via the APPC protocol, which can be Wide Area Network (WAN) or Open Transaction Manager Access (OTMA), which can only be used in a sysplex/monoplex environment.

The gateway will be the real client program for IMS and will run preferably on the same OS/390 machine as the IMS Transaction Server. The gateway will need to have a server part accepting and processing access to/from the browser through TCP/IP with its several flavors: HTTP, sockets or IIOP. This gateway function can be fulfilled by several components on the OS/390 side. The most important scenarios are:

- A Webserver
- IMS TCP/IP OTMA Connection (IMS TOC)
- MQSeries
- A sockets server

Figure 112 on page 209 illustrates an almost complete overview of all the flavors in accessing an IMS back-end by means of a server-side gateway.

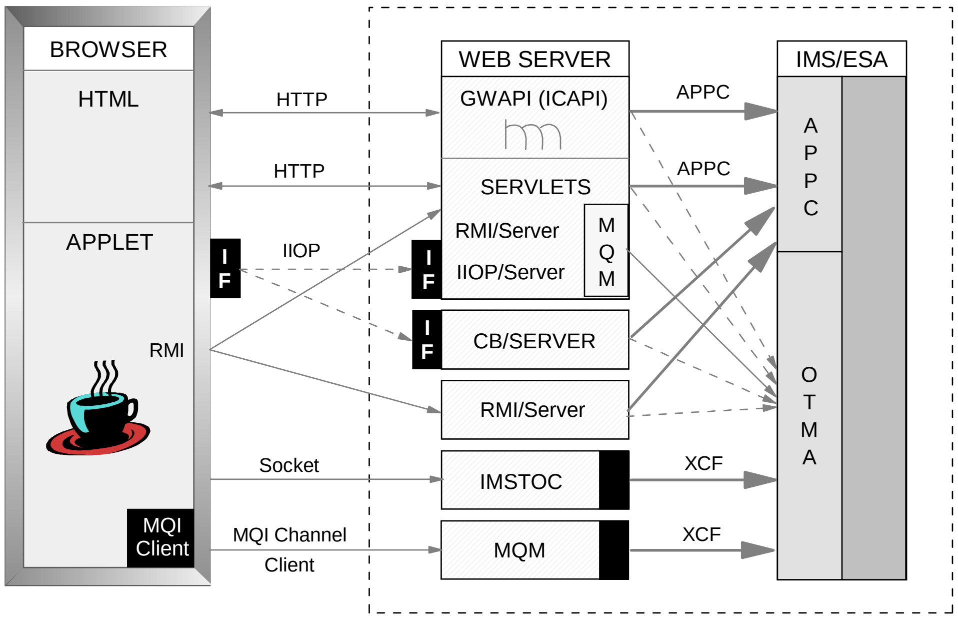


Figure 112. Server-Side Gateways to IMS

Whatever access method is used, in all cases we will have to deal with the aspects of logical messages for IMS transactions. In the following section we discuss in more detail the problems encountered when trying to use IMS 3270 transactions from the Web.

11.1.1 Datastreams

Figure 113 shows the exchange of messages between IMS and the browser.

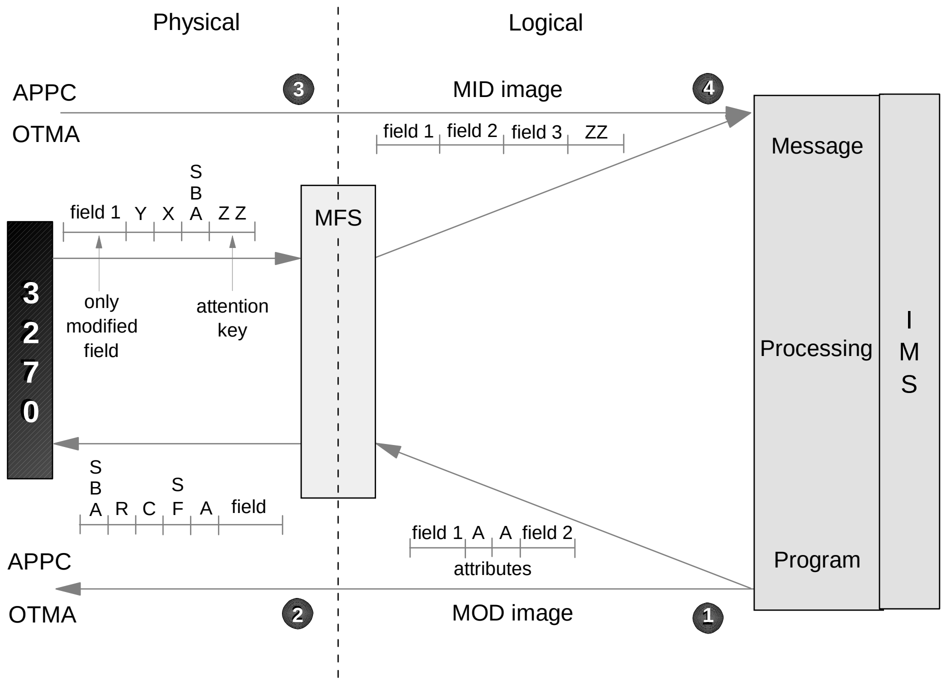


Figure 113. Flow of Messages from IMS to 3270 Terminal and Back

1. Logical outbound message

The logical outbound message is composed of segments. Each segment, which is the object of a separate send by IMS, contains fields. The layout of the message (segments) for 3270 has to correspond to the descriptions in the Message Output Descriptor (MOD) control block. The fields in general contain the data which has to be displayed on the screen, under the control of 3270 attributes and row/column coordinates. This control will be added by the MFS on outbound. On some occasions, the program may have to control the layout on the display. Attributes (which take 2 bytes) could precede the field data. This can be determined by looking at the MOD definition (ATTR = YES). Sometimes even more presentation control is passed by the MPP (such as extended attributes, cursor positioning and so on).

2. Physical outbound message (3270)

   MFS driven by the MOD and the Device Output Format (DOF) control, part of FMT will generally add, in front of each data field, 5 bytes: Set Buffer Address (SBA), ROW/COL(2) (a combined ROW/COL address), Start Field (SF) and Attribute (ATTR).

   The aspects related to these attributes are explicitly covered in other sections. Mapping between MOD and DOF is based on name tokens, and it is not required to have a one-to-one relationship for all fields.

   An APPC/OTMA client will not see this presentation layer. The client will only see the logical message and some presentation control (if it was passed in the logical message), but often we will have to mimic the attributes, which would have been set in the physical message.

3. Physical inbound message (3270)

   A 3270 input datastream will contain the datafields which were considered by the 3270 device as modified, preceded by their coordinates on the screen. The datastream will also indicate which “tabkey” (such as Enter, PFkey and so on) provoked the interrupt of the keyboard and inbound send.

4. Logical inbound message

   The logical inbound message layout can be determined by looking at the Message Input Descriptor (MID). In general, it will contain the modified data from the physical input message, if they were mapped. Note that “not modified” fields, which are not part of the physical input, will be primed under the direction of the MID. In this case, *primed* means that some filler characters can be substituted if a field was *not* modified by the 3270. It is also possible that literals are added in certain positions.

   The interrupt can also be translated to a logical indication (2-byte field) in the input. The cursor location, when the interrupt key was hit, could eventually also be propagated in the input message.

11.1.2 Attributes

The mapping of 3270 attributes to the browser is not straightforward.

We examine those aspects in detail and conclude this chapter with an idea of how to handle the attributes in a browser environment. The attribute sent to the device could indicate the following:

- Protected

This is very simple to mimic. It means that this data *cannot* be changed, so in general it becomes just text. If this field is also tagged as *modified*, this would be not very wise in general, because the data is still available in IMS. A conversational IMS transaction could bring an exception to this remark.

If, in conjunction with protected, a pre-modified indication was also sent, then we are not allowed to present the data as a “text input” field. The only way to achieve the same behavior is to send the data twice to the browser, as follows:

1. As simple text (which, of course, cannot be changed, but not sent back)
2. As “hidden field,” which is not visible, but will send back.

- Highlighted.

  We simply have to add a special layout tag (for instance, “color” or “font”) to the text, but otherwise it is generally handled as in the previous case.

- Modified

  A pre-modified indication causes a 3270 screen to always send this field back. If this field is also protected, a double define on the HTML page will do it.

How can we mimic the setting of the Modified Data Tag (MDT) bit for fields which will only be sent back if modified on the screen (browser)? Obviously, we have to display the data in an “input text” field, but what if the data is not changed? How can we prevent the data from being sent anyway?

The use of Javascript can help. The solution described here is certainly not exhaustive, but presents some ideas and shows the complementary character of Javascript. The example shows a solution with IMS Web Templates (IWT). The same solution could also be realized with JavaServer Pages (JSP). The idea is that for each field in the form, a boolean expression is reserved in a table called “mod.” The number of elements in this table are available through Javascript, as follows:

```
    nrfld = document.forms[0].elements.length;
```

The booleans in the array are originally set to “false.” The idea is to have all booleans for “modified” set to “true.” When the field is changed we drive the `setMode` script via the `onChange` clause and set its corresponding field in the mod table to “true.” Some fields could be mandatory or have to be considered as “pre-modified.” Those field names are registered in a second table called “modnm.”

When we submit the page, the “on Submit” script is driven which will first turn on the “mod” field to “true” for all registered names in the “modnm” table and then empty all fields whose corresponding indication in the mod table has not been changed from “false” to “true.” You may notice that each pre-modified field is followed by a small script where it is registered in the “modnm” table.

At the end of the HTML page, we use Javascript to turn all vars in the mod table to “false.”

Following is an excerpt of the Javascript source.

```
<HTML>
<HEAD>
<TITLE>How to Handle MOD tags example </TITLE>
<!-----------------------------------------------------------------------------------------
<!    JAVAscript
```

```
<!#-----------------------------------------------------------------------------------------
<script language="JavaScript">
var    mod = new Array();   1
var    modnm = new Array();   1
var    nrmodnm = 0;   1
var    nrfld = 0;   1

<!------------------------------------------------------------------------
function  subMod() {   2
   for (var ix = 0; ix < nrfld; ix++) {
      for (var iy = 0; iy < nrmodnm; iy++) {
         if (document.forms[0].elements[ix].name == modnm[iy] ) {
            mod[ix] = true   3
         }
      }
   }
   for (var ix = 0; ix < nrfld; ix++) {
      if (mod[ix] == false) {
         if (document.forms[0].elements[ix].type == "text") {   4
            document.forms[0].elements[ix].value="";
            document.forms[0].elements[ix].size = 0;
<!          alert( "Field " + document.forms[0].elements[ix].name + "(" + ix + ") set to NoMod");
         }
      }
   }
   return true;
}
<!------------------------------------------------------------------------
function  setMod(myfld) {   5
   for (var ix = 0; ix < nrfld; ix++) {
      if (document.forms[0].elements[ix].name == myfld.name) {
         mod[ix] = true;
         document.forms[0].mymessage.value =
           "formsfld nr " + ix + " " + myfld.name + "(" + myfld.type +") has been modified";
         return;
      }
   }
   alert(" Field " + myfld.name + " NOT found ");
   return;
}
<!------------------------------------------------------------------------
</script>
</HEAD>
<BODY>
<FORM Method="post" Action="http://...url... onSubmit="subMod()">
  .....

   <td><input type=text name=field1 size=25 maxlength=35 value="$(field1)"
       onChange="setMod(this)"></td>   6


   <td><input type=text name=field2 size=25 maxlength=35 value="$(field2)">
       </td>
<script>
modnm[nrmodnm] = "field2"; <- this field considered as premodified   7
nrmodnm = nrmodnm + 1;
</script>

  .....

<!------------------------------------------------------------------------
<!    JAVAscript
<!------------------------------------------------------------------------
<script>
nrfld = document.forms[0].elements.length;
for (var ix = 0; ix < nrfld ; ix++) {
   mod[ix] = false;
}
nrmodnm = $(nrmodnm);
document.forms[0].mymessage.value =
   "Form contains " + nrfld + " fields (Nr Premodified fields " + nrmodnm + " )";
</script>
<!-----------------------------------------------------------------------------------------
</FORM>
</BODY>
</HTML>
```

These explanations relate to the preceding example:

Notes:

1 Definitions of the “vars” and “var arrays” used in the Javascript.

2 The “subMod” script activated by the “Submit” button and the “onSubmit” clause.

3 These “mod” fields, with corresponding fields in the “modnm” table, are set to “true.”

4 The values of the “non-modified” fields are squeezed to empty.

5 The “setMod” script activated by the “onChange” clause on the field.

6 This is a field with an “onChange” clause.

7 This is a pre-modified field.

11.1.3 Basic Design of a Webserver Service Thread

Accessing IMS/VS transactions from the Web through the Webserver and keeping in mind that the Webserver will act as an HTTP server for the browser and one of its threads will act as an OTMA/APPC client for IMS requires the following steps for our design based on a container concept:

1. Data from the browser arrives as Variable/Name-Value pairs. Those “varvals” are put in the container, which, depending on the approach, could be a linked list or a hashtable.
2. The information in the container, with additional hardcoded literals, has to be used to build the input message. This can be done by Java classes or other code (template code is an example).
3. This input message is sent to IMS via one of the available techniques (MQM, APPC, OTMA).
4. The output from IMS/VS has to be analyzed and parsed into the existing container. This can also be done via Java classes or template-alike code.
5. Finally, from the combined information available in the container, the output has to be prepared in an HTML format and sent to the browser. This can be done with JSP, Java classes or template-alike code.

Figure 114 on page 214 shows the concept of a “container” to pass messages back and forth between the client and the server.

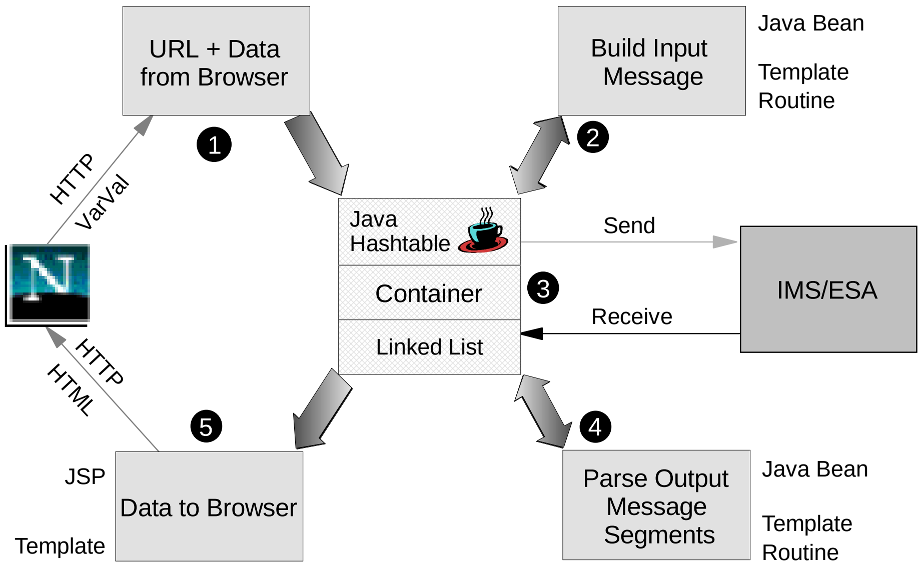


Figure 114. Exchanging Messages between IMS and Browser Based on the hashtable Concept

Some approaches will require the use of Java Native Interface (JNI). This JNI allows the invocation of C programs from Java and the interchange of data between both environments.

11.1.4 Conversational Transactions and HTTP

Conversational transactions in IMS are defined with a Scratch Path Area (SPA). When a conversational transaction is started over the APPC protocol, an APPC conversation is started over a SNA session between two LUs (see Figure 115). Multiple parallel APPC connections (sessions) can exist between two LUs (for instance, between the Webserver and IMS).

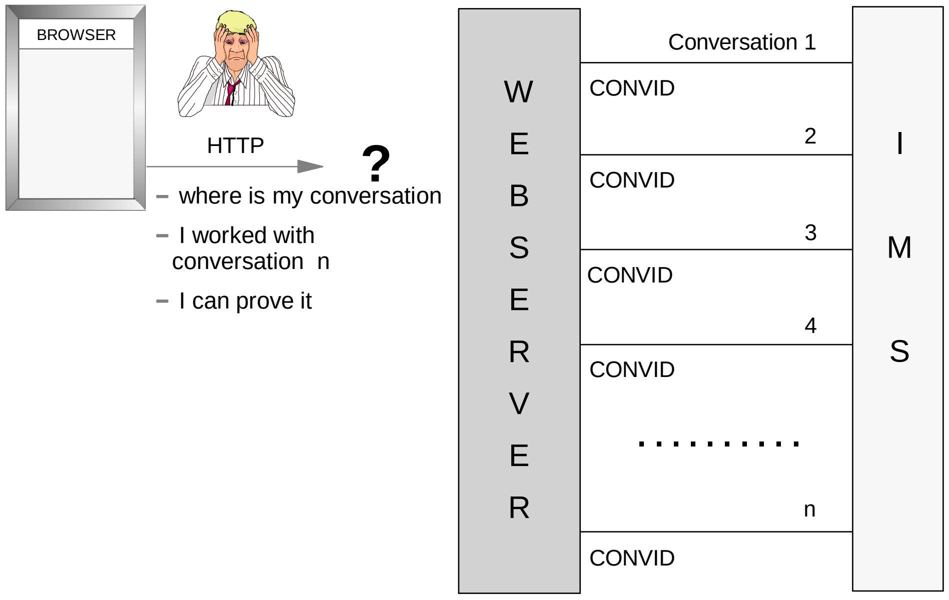


Figure 115. Conversational Transactions and HTTP

The number of sessions that can exist between two LUs is limited by VTAM definitions. Once a conversational protocol is started and *not* de-allocated, it is characterized by a conversational ID and reserves a session. The session can only be used by one conversation at the time.

This “convID” must be preserved on the client side and reused for consequent interactions. Here lies the problem for the browser interaction. The browser talks to the Webserver over the HTTP protocol, which is interrupted and reinstated, while the APPC protocol between the Webserver and APPC is kept and tokenized by the convID. When the browser returns to the Webserver and wants to continue the initiated conversation with IMS, then the convID must be found again reclaim the APPC tunnel.

The only way to achieve this is to keep session information for each active browser and allow the browser to find it again by presenting a token. This token can be sent back and forth between browser and Webserver, for instance via a “hidden value” or a “cookie,” the latter being certainly the best.

11.1.4.1 How to Set Up a Cookie

A *cookie* is a block of ASCII text that can be passed to the browser by the Webserver. The cookie is sent back to the Webserver when it is addressed by the browser. The cookie is sent to the browser as part of the response header and returned as part of the request header. To send the cookie, the Set-Cookie HTTP variable has to be set, with a value containing a series of variable/value pairs. Some of those will be user-defined, while others are special values. The following shows the cookie which could be set up as the contents of the HTTP_set-cookie variable. In the GWAPI environment this can be done with the HTTPD_set API.

```
convnr=1; adate=19121998; expires=...; path=...;
```

These explanations relate to the preceding example:

Notes:

- convnr and adate are user names.
- expires and path are cookie-related.

When returned from the browser, the value of the cookie can be obtained by retrieving the HTTP_COOKIE environment variable. In the GWAPI environment we can do it with the HTTPD_extract API.

We could deal with the session concept in the following way. Two solutions are presented. Both are based on cookies:

1. No servlet, but GWAPI

   A table is kept in the Webserver, where we reserve a number of slots significantly greater than the maximum of conversations we could have. Each slot has space for information like user information, timestamps, security information and the convID of APPC.

   When a certain browser works with a conversational transaction, a slot is allocated for it and the “slotnr” is returned via a cookie to the browser. This allows us to find the slot again and continue the conversation. This is the base concept.

   Other information that could be included in the cookie and verified at each resumption of the session is, for instance, the conversation step, slotcode and a Webserver identification. A Webserver could be restarted between two browser interactions, in which case the cookie token is not valid anymore and has to be renewed. The APPC conversations will have to be aborted.

2. Servlets with WebSphere

   The ServletExpress function in WebSphere supports, via its Java framework, the session concept. Objects can be preserved in the session. As objects are opaque and can be devised and materialized by your design, whatever you want to be preserved can be saved (for instance, in a `TMState` object). To make an object known to the `session` object, it has to be registered by code similar to the following excerpt:

   ```
   /*  -  TMState   object                  */
   tmstate = (TMState)session.getValue("TMSTATE");
   if (tmstate == null) {
      tmstate = new TMState();
      session.putValue("TMSTATE",tmstate);
   }
   ```

   Here, the session state should be registered under the name `TMSTATE`. When we retrieve the TMSTATE and the result is null, we instanciate a new one, and register it with a `putValue`. Next time, the `getValue` will return it. The session manipulation is handled by WebSphere, which will build, send and handle the cookie, containing appropriate information to keep up the session concept.

   You may note that in our implementation, we put some complete connection objects in `TMstate`. Keep in mind that, in practice, this means only the class variables, but this is a good use of the object concept. The concept discussed so far regarding the handling of the session state can also be applied to solutions using CICS EXCI, MQM or OTMA access for servlets running in ServletExpress.

   Objects registered in the session may implement the `valueBound` and `valueUnbound` methods and, as such, will be informed when the bounding occurs or when the object is unbound as a result of a release of the session.

11.2 Connecting to IMS Based on APPC

In this section we look at the possibilities for accessing IMS using the APPC protocol as follows:

- Using a servlet
- Using templates

Attention

In the following sections we refer to various pieces of sample code. You can find the code on the disk enclosed in this book. Refer to Appendix A, “CD-ROM” on page 299 for details.

11.2.1 IMS Access with Servlet and Via APPC

Advanced Program-to-Program Communication (APPC) is a useful, fast and reliable protocol that can be used in a Wide Area Network (WAN) or local (on the same platform) environment. Depending on the environment, although externally the same interface (for instance Common Program Interface Communication (CPIC)), the transport mechanism will differ. This means that on the same system, between a Webserver and IMS, storage-based transport will be used, omitting all VTAM address space support.

We designed a sample application accessing IMS via APPC, containing several Java classes, Java Native Interface (JNI) “glue” and C routines to do the actual interaction with APPC. See Figure 116.

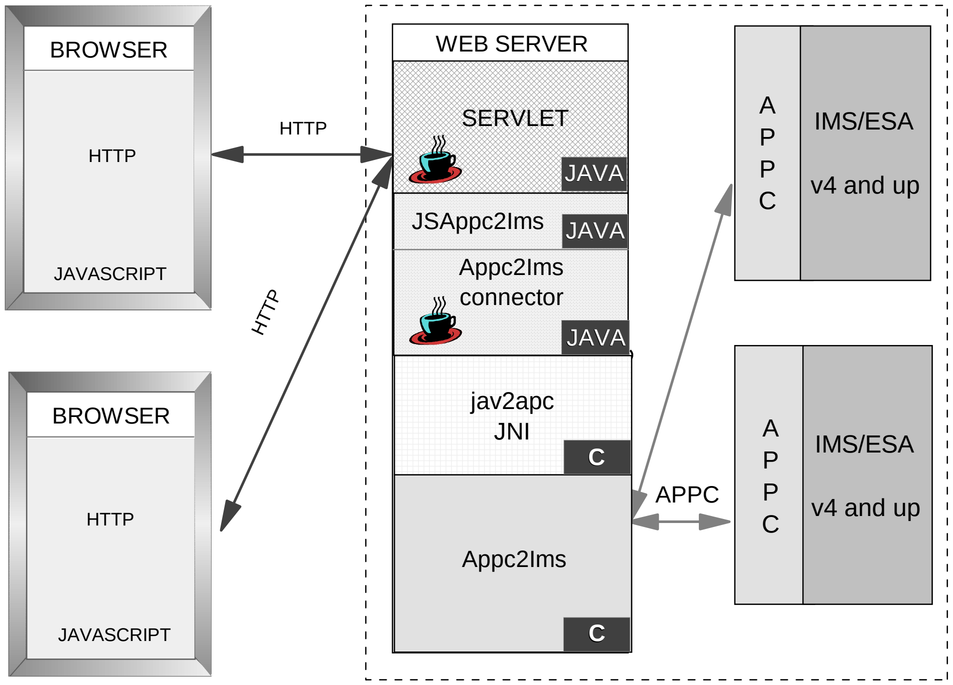


Figure 116. IMS Access from a Servlet Via APPC

Some of the classes are part of the overall design, while others are specific to the called transactions and have to be written for each specific TRAN and transaction result characterized by the layout of the reply. If the ISRT of the transaction result was done with a “ImsModName,” this modname will be the token for the output processing.

Following is a list of Java classes and C routines used in our sample. Each item is discussed in more detail in the next section.

parent class to instanciate the APPC framework and invoke it

itso/ibm/ims/JSAppc2Ims class

itso/ibm/ims/Appc2Ims class

itso/ibm/ims/IMSInputOutput class

itso/ibm/ims/IMSException class

itso/ibm/ims/TMState class

itso/ibm/ims/BuildParse interface

itso/ibm/ims/[IMSTRAN][ImsModName]edit classes implementing BuildParse

hjav2apc	JNI C routine (MVS/DLL libhjav2apc.so)
hapc2ims	C routine called by JNI routine hjav2apc
hhprintf	For trace printing (solves Tprintf reference)
hjgetref	JNI C routine (MVS/DLL libhjgetref.so)
hpRefDEf	C routine called by JNI routine hjgetref
hJNIRtns	C routine with JNI helper functions called by the JNI DLLs.
DFSLUEE0	IMS exit in assembler
HCMRCV	Assembler assistance routine for hapc2ims

11.2.1.1 The parent Class in the APPC Solution

This class is responsible for directing and starting the APPC communication. This is done from the Java servlet code, but for testing purposes this could also be done in a Java "application" with a MAIN entry. The following is a code excerpt of the parent class:

```
TMState              tmstate = null;  1
JSAppc2Ims           jac;  2
...
Hashtable            reqht; (in Batch, otherwise passed)  3
....
jac = new JSAppc2Ims();  4
jac.setReqht(reqht);  5
jac.setTmstate(tmstate);  6
jac.setTrclvl(trclvl);  7
try {
   System.err.println("ImServlet_doget calling JAppc2Ims_doIms\n");
   rc = jac.doIms();  8
} catch(IMSException e) {
   System.err.println("ImServlet IMSException" +e );
   return;
}
```

These explanations relate to the preceding example:

Notes:

1 `TMState` is the session state object for APPC/MQI. In a servlet environment we obtain it from the Session object. It is not mandatory to pass it, or it can be set to "null." It is mainly meant for conversational IMS transactions.

2 Reference to the `JSAppc2Ims` object.

3 "reqht" is the reference to the hashtable containing the varvals passed with the URL. Eventually other installation-related values could be added as in the following line:

```
    reqht.put("PACKAGE,"
"itso.ibm.imsEdit");
```

4 `JSAppc2Ims` is instanciated.

5 The connection request hashtable is set into the `JSAppc2Ims` object.

6 The `TMState` is set into the `JSAppc2Ims` object (this could be omitted).

7 The `trclvl` is propagated (this could be omitted); the default is 0.

8 The method `doIms()` is invoked. This will start the APPC communication preparation, as an exception could be thrown. The statement is surrounded by a TRY/CATCH sequence.

11.2.1.2 The JSAppc2Ims Class

The `doIms()` method instanciates immediately an `IMSInputOutput` object. This object will contain the input data and the result data. It is connection-related (not session-related), and is released at the end.

We also instanciate an `appc2Ims` object if not existing already, or recuperate the existing one from `TMState` if we were already in session and the object exists. We can follow the main excerpt here (the complete `JSAppc2Ims` code is included on the CD-ROM packaged with this book).

```
appc2ims = getAppc2ims();  // acquire  conversation related   1
inOut = getInOut();        // acquire  connection related    2
buildObjectFromHash();   3
try {
   ((BuildParse)inOut.getClassToLoad())).buildIn(reqht,inOut);   4
   appc2ims.setInOut(inOut);
   appc2ims.setTrclvl(trclvl);
   if (trclvl > 0)
      System.out.println("++JSAppc2Ims call Appc2Ims");
   apcrc = appc2ims.doIms();   5
   buildObjectToHash();   6
   ((BuildParse)inOut.getClassToLoad())).parseOut(reqht,inOut);   7
}
catch (Exception e)
{
   IMSException ex = new IMSException("JSAppc2Ims_doIms
      Error in calling APPC routines " +e);
   throw ex;
}
/*  inquiry the Transtate of Appc2Ims,
   if the state is conversational save it Transtate (Session) */
transtate = appc2ims.getTranstate();   8
if (transtate == CONVERSATIONAL)
   tmstate.setAppc2ims(appc2ims);
else {
   appc2ims = null;                   // free the Appc2Ims object
   tmstate.setAppc2ims(null);    // set it to null in TMState
}
inOut = null;   9
```

These explanations relate to the preceding example:

Notes:

1 The Appc2Ims object is recuperated or instanciated.

2 An IMSInPutOutput object is instanciated.

3 The local buildObjectFromHash propagates a group of properties into the Appc2Ims object. The propagated values come from the passed Hash container or from defaults stored as "final" values in the J2IMS class. A method called inOut.loadInClassFromHash is also invoked in the IMSInputOutput object to set connection-oriented properties, and to find the class which will build the input String.

4 This is a quite interesting statement. Read it from right to left. We will invoke a buildIn method on the object established by inOut.getClassToLoad(). This is an Edit class responding to a generic implementation. As such, the result object is cast with the implementation BuildParse. The InputString is now built. The build class relates to the ImsTran.

5 The IMSInputOutput object is set into the Appc2Ims object and the doIms() method is invoked on this object.

6 On return, basically the opposite will occur. The existing connection hashtable will be populated with the results (errors) arriving from IMS, for instance, the ImsModname. The class responsible for the parsing of the result segments is determined. This happens by invoking the loadOutClassToHash on the InputOutput object.

7 We invoke here a doParse method on the object established by inOut.getClassToLoad(). This is an Edit class responding to a generic implementation. As such, the result object is cast with the implementation BuildParse. This invocation works on the resulting OutputVector. The parse class is a function of the returned ImsModName if not null or equal to ImsTran.

8 The conversational status of the Appc2Ims object is inquired and, if we have a session with IMS, the entire Appc2Ims object is saved in tmstate, if tmstate is valid.

9 The IMSInputOutput object is released. It is connection-related. Remember that the Appc2Ims is conserved in the tmstate object, and this one is stored in the session.

11.2.1.3 The IMSInputOutput Class

This object is volatile and exists only during the connection. It contains mainly the elements that are required for this exchange with IMS. This includes not only the related fields, but also the methods to look up the classes that will be required to build the input datastring and to parse the outputvector with the segments.

```
public  int returnCode = 0;   1
public  int reasonCode = 0;
public  Vector outputVector = null; // output vector with result segments
String  imsTran = "********";       // IMS transaction is TpName
boolean debugOn = false;
String  imsModName = null;          // IMS Modname of result
```

```
Object  classToLoad = null;         // input/output build/parse classes
String  inputString = "";           // input data
public  int trclvl = 0;
char    comflag = 'S';

....
loadInClassFromHash(..)
....
  setClassToLoad(Class.forName((String)aHashTable.get("PACKAGE")+   2
     "."+(String)aHashTable.get("IMSTRAN")+ "edit").newInstance());
....
loadOutClassToHash(..)
  setClassToLoad(Class.forName((String)aHashTable.get("PACKAGE")+   2
     "."+name+"edit").newInstance());
```

These explanations relate to the preceding example:

Notes:

1 All fields that are defined here belong to one connection, and do not have to be kept over connections for the “Pseudo Session” concept.

2 These two methods will establish the Edit classes, which are required to build the input datastring and to parse the Output segment. Watch in particular the use of the `Class.forName` function, and the dynamic build of the classname by concatenating PACKAGE, IMSTRAN/ImsModName and “edit.” This class, which is an implementation of the `BuildParse` implementation, is instanciated and stored in this class to be used by the `JSAppc2Ims`.

11.2.1.4 The Appc2Ims Class

This is the object which, in collaboration with JNI, does the real client/server work. This is also the object which contains the APPC conversation status materialized in the transtate and in the related ConvID, which is the APPC session token. We will preserve this object in the `TMState`, which in turn is conserved in the servlets session object. This will allow us to pick up the `Appc2Ims` and the associated state to continue the interaction with IMS.

The main method here is `doIms`. This method will find out about the transtate and project a CMINIT or CMSEND as required by the status. This is stored in iaction. The `doApc` method invokes the JNI code. Do not consider `doapcIWT`, which is of use in other instances.

We now show a code excerpt:

```
public int doIms() {
   /*-------------------------------------------------------------*/
   /* determine whether we are in conversation                    */
   /*-------------------------------------------------------------*/
   if (transtate != CONVERSATIONAL) {
      iaction = MCMINIT;
   } else {
      iaction = MCMSEND;
   }
```

```
      if (inOut != null)
         comflag = inOut.getComflag();
         apcrc = selectapc();
         return apcrc;
      }

      public native int doapc(String luName, String tpName,IMSInputOutput inOut,
         String Modeent, int iaction, int timeout,char comflag,
         int transtate,int ipRef,int trclvl);
```

11.2.1.5 The IMSException Class

```
   public class IMSException extends Exception {
       public IMSException() {
   super();
       }
       public IMSException(String s) {
   super("IMSException :" + s);
       }
   }
```

11.2.1.6 The TMState Class

`TMState` conserves the state of the `Appc2Ims` object during the HTTP session, which encompasses many HTTP interactions. Basically, we only find properties in this class. As it is used in common with MQI, we also find in this class properties belonging to a MQI session.

```
   Appc2Ims   appc2ims = null;       // preserve the Appc2Ims object
   ....
   Hashtable  sessht = null;
   int        ipRef = 0;             // int casted pointer to refdata
   int        trclvl = 0;

   ....
   /*---------------------------*/
   public void setAppc2ims(Appc2Ims appc2ims) {
      this.appc2ims = appc2ims;
      return;
   }
   /*---------------------------*/
   public Appc2Ims getAppc2ims() {
      return appc2ims;
   }
```

11.2.1.7 The BuildParse Interface

```
 public interface BuildParse )

   public int buildIn(Hashtable aHashtable, IMSInputOutput inOut);
   public int parseOut(Hashtable aHashtable, IMSInputOutput inOut);
```

11.2.1.8 [IMSTRAN][ImsModName]edit Classes Implementing BuildParse

We present here one class as an example of implementing `BuildParse`:

```
public class DFSMO1edit implements BuildParse
{
   int     rc;

   public int buildIn(Hashtable  h, IMSInputOutput inout)
   {
      rc = 0;
      return rc;
   }
   /*-------------------------------------------------------------------*/
   public int parseOut(Hashtable  h, IMSInputOutput inout)
   {
      int    segnr;
      String segment;

      Enumeration e = inout.getOutputVector().elements();
      segnr = 0;
      rc = 0;
      while (e.hasMoreElements()) {
         segment = (String)e.nextElement();
         if (segnr == 0) {
            h.put("MESSAGE",segment);
         }
         segnr++;
      } // end while
      h.put("NRSEGM", ""+segnr);
      return rc;
   }
}
```

This class handles IMS output which was written out with the MOD DFSMO1, mostly system messages. Here we find only a `doParse` implementation. The buildIn part is dummy, but must be there as the implementation has to complete.

Look at the name of the class: on input to IMS, we use the transaction name as the base for the classname, whereas on output, the ImsModName is the base. If the program inserted the output message without a Modname, or DFSLUEE0 is not implemented, we use also the transaction name on output.

11.2.1.9 The hjav2apc.c Routine

This routine is the “glue” code between Java and the next program. It uses the JNI interface. We pass as many parameters as possible via the call. This avoids having to use too many JNI API calls. Using this technique, remember that there could be a difference between the way data is stored in Java and the format it is expected to be in C.

For instance, character data (and strings) is stored in ASCII format in Java. As we pass directly the character or the string object, we have to take into account the required builds or conversions in the C program. Mostly we do this with helper routines located in hJNIRtns.c. For this discussion, only consider the doapc entry point.

```
JNIEXPORT jint JNICALL Java_itso_ibm_ims_Appc2Ims_doapc
   (JNIEnv *jne,jobject jcaller,jstring jluName,jstring jtpName,
   jobject jacinout,jstring jmodeEnt,jint aciaction,jint timeout,
   jchar jcomflag,jint transtate,jint ipRef,jint actrclvl)
```

The first two parameters are always passed by the JNI call:

- The jne parameter represents the Java Native Environment. This reference will be required in all coming JNI calls.
- The jobject parameter is always a reference to the calling object.
- The other parameters are specified in the native method of the caller class.

Integer parameters (jint) can be used immediately. They are passed in big endian format.

Characters (jchar) are in ACII (UTF) and have to be converted to EBCDIC on an S/390 platform.

Strings also have to be converted to EBCDIC (C array). For this purpose we developed a help function, as follows:

```
s = getCFromJString(jne,jluName,buffer,buflen,pRef,actrclvl)
```

For details about this function, refer to the code. The helper routine hJNIRtns contains many other functions which can be very useful in extracting data from a Java class or in setting data in a class.

In general, to get data from or put data to a class, or to invoke a method from C, an absolute reference is required to the class instance. Within this class, a relative ID to the required property or method is required.

The code presented here shows many applications of the JNI API. However, we do not discuss JNI any further as it is beyond the scope of this redbook.

This routine, together with a few other routines, is linked into libhjav2apc, a DLL which is defined in the Appc2Ims class, as follows:

```
static
{
   System.loadLibrary("hjav2apc");
}
```

11.2.1.10 The hapc2ims.c Routine

This routine is the “work horse” of the APPC-to-IMS connection. It executes the required sequence of APPC calls as a function of the conversation state, request data and the return code of earlier calls. The program can be entered for the following reasons:

- CMINIT

This starts a fresh APPC conversation. Depending on the selected transaction defined as (CONVERSATIONAL), the APPC flow terminates by a CMRCV, in which case the next interaction will go on with a new CMSEND. The token used to continue the ongoing conversation is convID, and had to be saved in one way or another. If the called transaction was *not conversational*, the APPC flow terminated with a CMDEAL and a new IMS would have to restart with a CMINIT.

- CMSEND

  This option continues a previous started transaction, which had to be conversational. An ongoing transaction can always be deallocated by the transaction, in which case we return to a non-conversational state.

- CMDEAL

  This option is only to be used when an existing conversation has to be deallocated in a "forced" way.

The result of this call is a vector with output segments, a return code (which can be a pseudo in case of a timeout), a format indication (ImsModName), if the DFSLUEE0 exit is installed, and a convID *NOT* equal to blank, if the conversation has to be continued.

11.2.1.11 The hhprintf.c Routine

The hhprintf.c routine is part of all .so modules and contains two external reference entrypoints:

```
int  Hprintf(REFDATA *pRef, const char *fmt, ...)
...
int  Tprintf(REFDATA *pRef, const char *fmt, ...)
```

It has to be linked with the DLL modules. This routine does not execute essential functions, except that it is responsible for directing output printed via Hprintf or Tprintf to an adequate/available target. This decision is based on the information and references set in the REFDATA vector passed by the Hprintf and Tprintf request.

The pRef is a structure reference that is used to determine in which environment the program is running. It will help us to direct the output to the appropriate destination.

The routine is also compiled conditionally depending on the fact whether it is part of JNI code or not.

11.2.1.12 hjgetref JNI C Routine (MVS/DLL libhjgetref.so)

This DLL, attached to the `TMState class`, is called to prime the REFDATA vector, which, as mentioned before, is passed among the different C routines through the pRef pointer. To fulfill this task, the next C routine is called. The call of this routine is based upon the availability of a shared area, pointed by the first LE pointer of the enclave. This is verified in the following way:

```
//  get anchor to Global zone
function_code = QUERY;
field_number = 1;
CEE3USR( &function_code, &field_number, &field_value, &fc);
```

```
if ( _FBCHECK (fc, CEE000 ) != 0 ) ??<
   printf("++SVS CEE3USR failed with message nr %d\n",fc.tok_msgno);
   .....  default dummy  shared area  ...
??>  else ??<
   .....  call hpRefDef   .....
??>
gshr_ptr = (THREAD_SHARED *)field_value;
```

11.2.1.13 hpRefDef C Routine

The hpRefDEf C routine is called by JNI module hjgetref and can only operate in an IWT environment. In other words, the IWT init code located in gwapex01, was setting up the required shared area. If this environment was *not* set up, then this routine will not be called.

11.2.1.14 hJNIRtns C Routine

This file regroups a collection of functions that are used in a JNI environment. As we developed a lot of JNI code, we found it useful to put them together in one deck, and eventually extend when required.

The routines regrouped here allow for access to Java resources. This includes access to aggregates (objects, primitives) directly specified in a class, but also access to methods and consecutively the invocation of those. In general, to find elements in an instanciated Java class, we need three elements:

- A reference to the class object
- The ID of the class
- The ID of the element (object, primitive or method) within the class

To find the ID of an element, a signature has to be provided. As in Java, everything is stored and registered in ASCII(UTF), and signatures have to be provided in ASCII. Thus, in an OS/390 environment, we have to provide translation facilities. When dealing with strings and character fields, JNI always wants and returns it in ASCII.

For further details about writing JNI, refer to the literature. A good explanation can be found in *Java Secrets* by Eliotte Rusty Harold (IDG Books) or at the following URL: `http://www.ibm.com/java/education`

11.2.1.15 DFSLUEE0

For output messages, IMS calls the LU 6.2 Edit exit routine for each message segment before the message segment is sent to the LU 6.2 program. The exit routine can intercept the data sent by the application program and edit it for the particular destination. It is also called if a message is inserted from an alternate PCB destined for an LU 6.2 destination. This exit routine is for use with standard IMS and modified IMS application programs. You can write the LU 6.2 Edit exit routine to do the following:

- View the contents of a message segment and continue processing
- Change the contents of a message segment and continue processing
- Discard a message segment

For output messages, IMS calls the LU 6.2 Edit exit routine for each message segment before the message segment is sent to the LU 6.2 program. The exit routine can intercept the data sent by the application program and edit it for the particular destination. In our sample code we use the Exit to add the ImsModName in front of the output datastream. The ImsModName can be considered as a transaction code on the client side; it expresses how the datastream has to be processed on the client side (Webserver client code, or Applet code). Our implementation always prefixes the ImsModName to the message if it is available, but the prefixing could be conditional on tags available in the control blocks available passed in by the Exit.

The exit is fully explained in *IMS/ESA V6 Customization Guide*, SC26-8732. To detect whether prefixing has been done or not, a small `X'4141'` token precedes the 8-character ImsModName.

11.2.1.16 HCMRCV

This routine, used in conjunction with hapc2ims, will handle the first receive call after a transaction (apparently successful) has been sent to IMS. It handles a timeout. Normally, for exceptional situations, a client program is immediately informed by a return code about the malfunction of the IMS server or the connection, and can then handle things accordingly.

In a few conditions, IMS will not reply: for instance, when the transaction is queued in IMS, but we have a long waiting line, or no Message Processing Region has been started for this transaction class. A transaction could also be looping.

In such conditions, we have to abort after a certain delay. HCMRCV takes care of this timeout; two threads are started, and one waits on the IMS reply, while the other waits for a time delay. Whichever one times out first the one which timeouts first will present the result.

11.2.2 IMS Access from a Servlet Using APPC and Templates

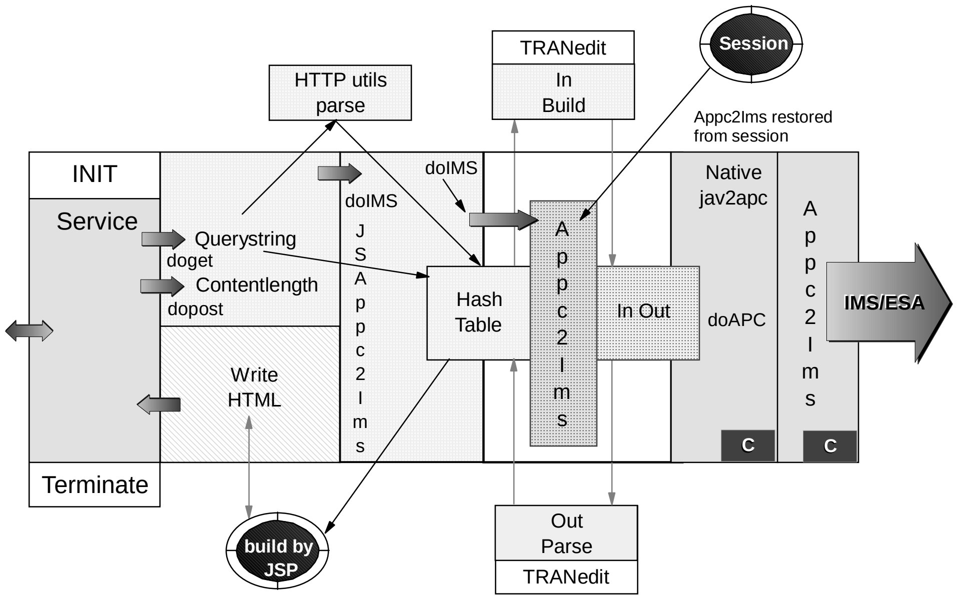


Figure 117. IMS Web Templates

Why would we want to use IBM Web Templates (IWT)? What could a design like IWT do for us? JSP solves the dynamic building of the HTML pages in a certain way and we adopted this solution in all servlet implementations. The build of the input message for IMS, and the parsing of the output segments from IMS, has still to be done.

In the previous servlet cases, we did this with the `TRANedit` and the `MODNAMEedit` classes of the `itso.ibm.imsEdit` package[4] The aim of this implementation is to do this task with the templates, the same ones as used in the pure IWT implementation. From a practical point of view, the differences are the following:

- The IWT linked list CONTAINER is complemented with a hashtable.
- The `JSAppc2Ims` class is replaced by the `IwtBase` class.

  This class will instanciate a `AddLookHash` class, which, in combination with the JNI hjreadmc, and using the same IWT modules with some extensions, can build the input message from the browser input, and parse the output result into the hashtable, where it will be available for JSP processing.

The strength of this approach is that it is very simple, quickly implemented and we use the session framework of the servlets. The code that we used is not meant to be a product, but it shows a way to build the IMS_IN message from information in a container.

The IO templates have an MFS “look.” On output, an IMS_OUT template allows for the parsing of the output segments into the hashtable, and all this collected information can be used to build in a dynamic way the browser output.

The APPC solution with templates is composed of the following:

parent class to instanciate the IWT framework and invoke it

itso/ibm/servlets/IwtBase class

itso/ibm/ims/AddLookHash class

hreadmac	JNI C routine (MVS/DLL libhjreadmc.so)

itso/ibm/ims/Appc2Ims class

itso/ibm/ims/TMState class

hjav2apc	JNI C routine (MVS/DLL libhjav2apc.so)
hapc2ims	C routine called by JNI routine hjav2apc
hhprintf	For trace printing (solves Tprintf reference)
hjgetref	JNI C routine (MVS/DLL libhjgetref.so)
hpRefDEf	C routine called by JNI routine hjgetref
hJNIRtns	C routine with JNI helper functions called by JNI DLLs
DFSLUEE0	IMS exit in assembler
HCMRCV	Assembler assistance routine for hapc2ims

[4] Refer to the enclosed disk for the code.

Some components are specific for the APPC/IWT solution, while others are the same as in the solution without IWT, as discussed in 11.2.1, “IMS Access with Servlet and Via APPC” on page 217. In the following sections we discuss the IWT-specific components and refer to other sections for the components that are the same as in the solution without IWT.

11.2.2.1 The parent Class

This servlet code is responsible for the instanciation of the IWT/APPC solution. The following is a code excerpt related to this task:

```
IwtBase              iwtb;   1
...

tmstate.setTrclvl(trclvl);
iwtb = new IwtBase();        2
iwtb.setReqht(reqht);        3
iwtb.setTmstate(tmstate);    4
iwtb.setHwriter(hwriter);    5
iwtb.setTrclvl(trclvl);
if (trclvl > 4) {
   System.out.println("++ImServlet3_performTask IWT2 ++");
}
rc = iwtb.doIwt();           6
```

These explanations related to the preceding example:

Notes:

1 Reference to the `IwtBase` object.

2 `IwtBase` is instanciated.

3 reqht is the reference to the hashtable containing the varvals passed with the URL. Eventually other installation-related values were added.

4 The `TMState` object is set into `IwtBase`.

5 The `hwriter` object is set so that eventually we will have the possibility to write directly to the browser without JSP.

6 The method `doIwt()` is invoked. This will start the all process.

11.2.2.2 The IwtBase Class

This class will instanciate AddLookHash, which is responsible for the “templates” interface. The class will be called first for building the input message to IMS. Remember that this message has to be built from information available in the container which came from the browser.

```
AddLookHash        al = null;   1
....
process = (String)reqht.get("PROCESS");   2

/*----------------------------------------------------*/
/*   create an AddLookHash object                      */
```

```
/*------------------------------------------------*/
....
....
al = new AddLookHash (reqht,macropath,hwriter,trclvl);   3
....

if (ipRef == 0) {   4
   ipRef = al.getIpRef();
   System.out.println("++IwtBase_doIwt ipRef was NUL is now set to
} else
   al.setIpRef(ipRef);
....
if ( ImsCics == IMS)   5
   impart = "%IMS_IN";
else
   impart = "%CICS_IN";
....

if (trclvl > 0) {
   System.out.println
      ("++IwtBase_doIwt_doInputBuild with skeleton");
}
inprc = al.doInputBuild (process, impart, in_data);   6
....
```

The input is built in a C structure and passed between the different C JNI modules via the Refdata structure pointed by pRef. This structure was obtained earlier.

Notes:

1 Reference to the AddLookHash object.

2 The name of the "Process" is obtained. It is the name of the template or DLL module which will be used for the build of the input message to IMS.

3 AddLookHash is instanciated and we pass in some parameters:

1. reqht is the container with the "varvals."
2. macropath is the path where the templates are located.
3. Via hwriter with a println function, we could print to the browser:

```
hwriter.println("<br>++IwtBase_doIwt with trclvl " + trclvl);
```

4. Via trclvl is self-explanatory; this can be used for testing purposes.

4 ipRef is the integer value of a pointer to a REFDATA shared structure which points to a group of other used resources. This pointer is passed from module to module, from class to class. Here, this shared area is built (if it was not previously built).

5 This design is also used with CICS and EXCI. The parameter TM indicates whether we use CICS or IMS.

6 Here we call the InputBuild function in AddLookHash, which in turn will invoke through a native method the appropriate JNI code located in the libhreadmc.so DLL.

The input message is kept in C-managed storage pointed by a pointer in the refdata structure. In Java, we contain the pointer to this structure in an integer, and as such we can pass it to other instanciated classes. That is what we are doing with the Appc2Ims class, which is the same as the one used in the previous case.

```
ipRef = al.getIpRef();   1
...
if ( ImsCics == IMS) {
/* =========================== IMS   access =====================
   appc2ims = getAppc2ims();   2
   if (appc2ims.getTranstate() == NON_CONVERSATIONAL) {   3
      appc2ims.setTimeout((String)reqht.get("TIMEOUT"));
      appc2ims.setModeent((String)reqht.get("MODE_ENT"));
      appc2ims.setPluName((String)reqht.get("PLU_NAME"));
      appc2ims.setTpName((String)reqht.get("IMSTRAN"));
   } /* endif */
   ....
   appc2ims.setIpRef(ipRef);   4
   aprc = appc2ims.doIms();   5
```

These explanations relate to the preceding example:

Notes:

1 We get the integer value of the pointer to the reference structure.

2 We try to get an Appc2Ims object. This will be a new one or one that was preserved in the TMState object.

3 Several properties are set in the Appc2Ims object.

4 We pass the reference structure to Appc2Ims.

5 Here we invoke APPC.

The result of this operation consists of two important elements:

1. A response structure with pointers to segments returned by IMS.
2. An ImsModName returned by IMS if the DFSLUEE0 exit is installed. We will call this element HTMLMOD also because it characterizes what we received (a good or bad response) and indicates how it has to be parsed and later on presented.

```
htmlmod = appc2ims.getImsModName();   1
if (htmlmod == null)
   htmlmod = procsave;   2
transtate = appc2ims.getTranstate();
if (tmstate == null) {
   if (transtate == CONVERSATIONAL)
      System.out.println ("++IwtBase_do
} else {   3
   if (transtate == CONVERSATIONAL)
      tmstate.setAppc2ims(appc2ims);
   else {
      appc2ims = null;              //
```

```
            tmstate.setAppc2ims(null);      //
         }
      }
```

These explanations relate to the preceding example:

Notes:

1 We try to get the ImsModName.

2 If this object is null, we use the original process name instead.

3 If the transaction is in a conversational state, we save the `Appc2Ims` object in `TMState`; otherwise, we assure that the reference is null.

```
if ( ImsCics == IMS)   1
   impart = "%IMS_OUT";
else
   impart = "%CICS_OUT";
reqht.put("HTMLMOD",htmlmod);
....
if (ijsp == YES)   2
   al.setHashput('Y');               // we set output in the HASH
else
   al.setHashput('N');               // we will write HTML via IWT
....
outrc =al.doOutputParse(htmlmod, impart);   3
```

These explanations relate to the preceding example:

Notes:

1 As we prepare for parsing the result from IMS/CICS, we select the appropriate option.

2 The parse basically means the creation of additional varvals in the container. The container will be the source for the dynamic HTML. This build can be done by JSP or by templates. Depending on the selected option, the parsing results will be written to the hashtable or to a Linked list.

3 Finally, we activate the parsing and indicate which skeleton (htmlmod) to use.

If the HTML presentation has to be built with the templates (JSP = NO), we will invoke `AddLookHash`, once more for this and exit with rc=1, indicating that no JSP processing has to be done by the servlet.

```
if (ijsp == YES) {
   reqht.remove("JSP");
   rc = 0;       // HTML has to be done by JSP   1
   }
} else {
   rc = al.doSkelHtml(htmlmod,"%HTML_OUT", hwriter);   2
   .....
   rc = 1;       // HTML has been done   3
}
```

These explanations relate to the preceding example:

Notes:

1 Dynamic page by JSP, return with 0.

2 Dynamic page by templates file, so we invoke the `doSkelHtml` method.

3 We return with 1 to indicate that no further processing has to be done.

11.2.2.3 The AddLookHash Class

This class, with the collaboration of the JNI DLL libhjreadmc, is the template services provider that we used from class `IwtBase`. It contains several references to entry points in the DLL module. We can look at some definitions:

```
public native int dohtml(
  int ipRef,String impnm,String sklnm,char hpt,.......
public native int doinBuild(
  int ipRef,String impnm,String sklnm,String in_data,int trclvl);
public native int dooutParse
  int ipRef,String impnm,String sklnm,int out_p,......
.....
```

You can recognize the methods which were called by class `IwtBase` to the three major required functions:

1. Build input for IMS from the container.
2. Parse output from IMS into the container.
3. Use the container contents to build the dynamic HTML page.

This Java class is a wrapper for a lot of native code. Besides the three major methods, other related help functions are provided. The DLL containing all the required entry points is named in this method by the following statement:

```
static
{
   System.loadLibrary("hjreadmc");
}
```

11.2.2.4 The hreadmac.c Routine

This routine is the template processor. We do not describe it in detail here and only provide you a sample of what could be done with this approach. We show the templates required to make the input for one of the sample programs MANUFAIL, and to parse the output.

```
%IMS_IN
%# For IMS_IN Enter elements in following sequence
%# ---------------------------------------------------
%# pass the part name with trailing null
%# ---------------------------------------------------
MFLD=in_man LENGTH=30 FILLUP=BLANK   1
%}
```

```
%IMS_OUT
%# -------------------------------------------------
%# -------------------------------------------------
SEGM=0
MFLD=ca_rc       LENGTH=1   2
MFLD=ca_sqlcode  LENGTH=9
MFLD=ca_message  LENGTH=20
MFLD=ca_cnt      LENGTH=9
%DO  (I)  0 $(ca_cnt)
MFLD=out_man$(I) LENGTH=30
%}
%}
```

These explanations relate to the preceding example:

Notes:

> **1** The input is just one field, besides of course the IMS transaction code. The field was received from the browser under the name in_man. It is placed in the input from pos 0-30.
>
> **2** The output contains a prefix with some header fields. They are put by their corresponding names into the container as new varvals. The last field is repetitive and is pulled from the IMS ouput via a do loop. The upper limit of the do was determined by the ca_cnt. The resulting manufactors are put in the containers with the following keyname:
>
> ```
> out_man_n    (n =  0...ca_cnt-1)
> ```
>
> Once in the container, they can pulled out for the building of the dynamic HTML pages.

The hreadmac is the main file for the template processing, but in reality many of other routines are called. These are located in other files of the IWT code:

hvarlook.c	Routines to fetch values from the container
hadrfval.c	Routines to put values in the container
hgetoken.c	Routines to find tokens in input
hfilerd.c	Routine for reading templates
hgdllent.c	Routine load DLL routines
hmvsdump.c	Routine to print storage in dump format
hutilrtn.c	Several helper routines

11.2.2.5 The Appc2Ims Class

Refer to 11.2.1.4, “The Appc2Ims Class” on page 221 for a description of this class.

11.2.2.6 The hjav2apc.c Routine

Although this routine is the same as for the "non-IWT APPC," we use here a different entry point due to the way the input string for IMS and the output segment from IMS are passed and handled. In "non-IWT" the build and parse is done by Java classes, here it is done by template processing. The entry point used is:

```
JNIEXPORT jint JNICALL Java_itso_ibm_ims_Appc2Ims_doapcIWT
  (JNIEnv *jne,jobject jcaller,jstring jluName,jstring jtpName,
  jint in_p,
  jstring jmodeEnt,jint aciaction,jint timeout,jchar jcomflag,
  jint transtate,jint ipRef,jint trclvl)
```

11.2.2.7 The hapc2ims.c Routine

This routine is exactly the same as in the APPC case; refer to 11.2.1.10, "The hapc2ims.c Routine" on page 224 for any explanation.

11.2.2.8 The hhprintf.c Routine

Refer to 11.2.1.11, "The hhprintf.c Routine" on page 225 for details.

11.2.2.9 hjgetref JNI C Routine (MVS/DLL libhjgetref.so)

Refer to 11.2.1.12, "hjgetref JNI C Routine (MVS/DLL libhjgetref.so)" on page 225 for details.

11.2.2.10 hpRefDEf C Routine

Refer to 11.2.1.13, "hpRefDef C Routine" on page 226 for details.

11.2.2.11 hJNIRtns C Routine

Refer to 11.2.1.14, "hJNIRtns C Routine" on page 226 for details.

11.2.2.12 DFSLUEE0

Refer to 11.2.1.15, "DFSLUEE0" on page 226 for details.

11.2.2.13 HCMRCV

Refer to 11.2.1.16, "HCMRCV" on page 227 for details.

11.2.2.14 Conclusion

IBM Web templates (IWT) is *not* a product, it is an "as is" bundle of routines, which shows how easy it can be to build solutions for accessing existing IMS and CICS transactions from the Web. JSP brings the build of dynamic HTML pages and templates help to build and parse the messages to/from the TM systems. The basic concept element is the *container*, which is referenced and populated in almost all modules. It is the information vehicle throughout the design. The container is considered to be a concept. The implementation, depending on the usage, can be in a linked list or in a hashtable.

11.3 Access to IMS Using a Servlet/MQI

Attention

In the following sections we refer to various pieces of sample code. You can find the code on the disk enclosed in this book. Refer to Appendix A, "CD-ROM" on page 299 for details.

Client programs can connect to the Message Queueing Manager (MQM) in two ways:

1. They can connect via a direct attachment if they are running as an adjacent process on the same platform. This will be called the MQM binding.
2. Remote clients can connect to the MQM via a bidirectional client channel. This is the client support.

In both cases, all queues are located in the MQM. MQSeries Client for Java and MQSeries Bindings for Java, provide support to enable Java applets and applications to use MQSeries applications.

The two Java packages involved are:

1. MQSeries Client for Java

   This is an MQSeries client written in the Java programming language for communicating via TCP/IP. It enables Web browsers with Java applets and applications to issue calls and queries to MQSeries, giving access to mainframe and legacy applications over the Internet without the need for any other MQSeries code stored on the client machine.

2. MQSeries Bindings for Java

   This package enables you to write server-side MQSeries applications using the Java programming language. These applications communicate directly with MQSeries queue managers to provide a high-productivity and high-performance development option.

Both packages are available for download from URL:
`http://www.ibm.com/software/ts/mqseries`

Figure 118 on page 237 shows clearly both options. The client option is explicitly used and explained in *Integrating Java with Existing Data and Applications*, SG24-5142.

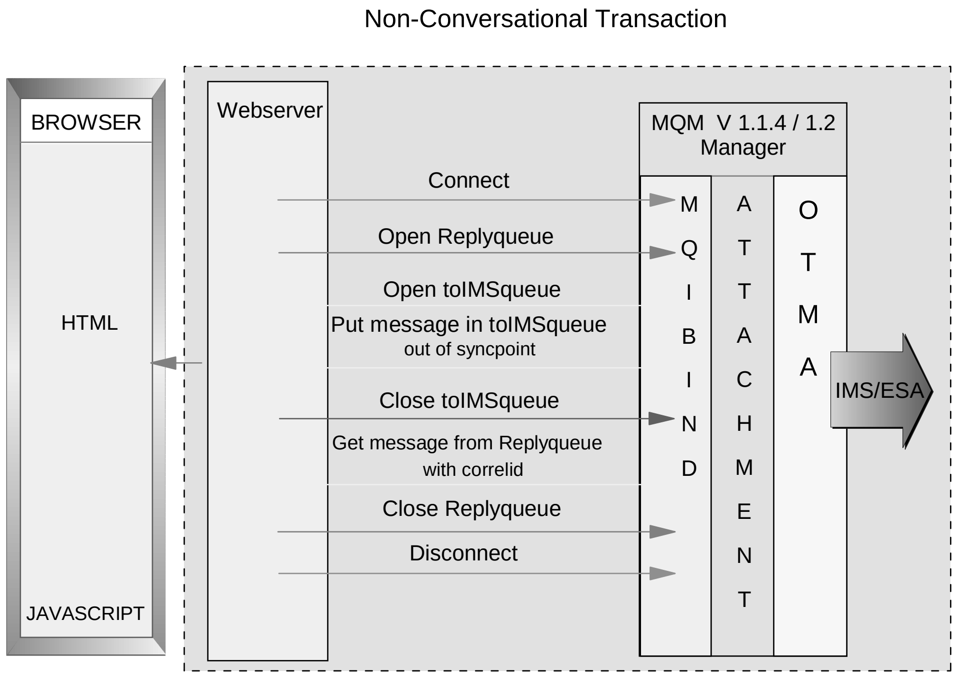

Non-Conversational Transaction
BROWSER
HTML
JAVASCRIPT
Webserver
MQM V 1.1.4 / 1.2 Manager
Connect
Open Replyqueue
Open toIMSqueue
Put message in toIMSqueue out of syncpoint
Close toIMSqueue
Get message from Replyqueue with correlid
Close Replyqueue
Disconnect
MQIBIND
ATTACHMENT
OTMA
IMS/ESA

Figure 118. MQM Bridge to IMS Protocol

In both cases, the MQI/IMS bridge is used to reach IMS. As such, the command flow used in both cases is about the same. The MQI/IMS bridge uses two channels through a special storage class defined as communicator with IMS. One channel will be used as a “To Channel,” the other one is the “From Channel.”

Following are the definitions of the channels as they were used during our testing.

```
 DEFINE STGCLASS(TOIMSB) PSID(01) DESCR('OTMA TO IMSB')  +  1
    REPLACE   XCFGNAME(ITSOIMS) XCFMNAME(APIMSA58)  2
******
 DEFINE QLOCAL( 'MQS2.TOIMSB' ) +  3
          REPLACE +
* COMMON QUEUE ATTRIBUTES
          DESCR( 'OTMA TO IMSB' ) +
          SHARE +
          NOTRIGGER +
          DEFSOPT(SHARED) +
          DEFPSIST(NO) +
* LOCAL QUEUE ATTRIBUTES
          GET( ENABLED ) +
          STGCLASS(TOIMSB) +
          INDXTYPE( NONE )
******
 DEFINE QLOCAL( 'MQS2.FROMIMSB' ) +  4
          REPLACE +
* COMMON QUEUE ATTRIBUTES
          DESCR( 'REPLYQ FROM IMSB' ) +
          PUT( ENABLED ) +
          DEFPRTY( 5 ) +
          DEFSOPT(SHARED) +
          DEFPSIST(NO) +
* LOCAL QUEUE ATTRIBUTES
```

```
               GET( ENABLED ) +
               SHARE
```

These explanations are related to the preceding example:

Notes:

1 The TOIMS queue is defined within the special storage class, which binds it logically with the target IMS.

2 The communication between IMS and MQM uses the Extended Coupling Facility (XCF), which can be used within a sysplex. IMS and MQM systems wanting to communicate through the bridge must belong to the same XCF group, in this case ITSOIMS. The storage class also defines the destination.

3 The TOIMS queue is linked with the special for IMS defined Storage Class.

4 The FROMIMS queue is a normal local queue whose name will be specified in the header in the request to IMS, so that the response from IMS can be directed to this queue.

The communication between MQM and IMS is done over the Open Transaction Management Adapter (OTMA) protocol. This protocol is based on XCF. The server implementation is part of IMS while MQM is the client. The client implementation is done in a "queue" way: one queue is the "to IMS" channel to send messages to IMS, and another channel acts as the "from IMS" channel where we will try to get the response. The In-Message (request) and response are correlated by a correlationID, which is returned by the send of the request so that it can be used as a parameter when issuing the receive tentative.

Important points to keep in mind are the following:

- The connection to an MQM manager, being a rather costly operation, should only be done once and existing connections should be preserved and reused.
- The preceding remark, in a minor way, is also valid for the "queue opens" and we should look for reuse of the queue handles as well.
- The main difference between the APPC convID and the MQI references to connection and queues is that the convID is session-related, while MQI references are thread-related. As the consecutive interactions from a browser to a Webserver can be backed by different threads, the MQI references have to be saved on a thread base and *not* on a session base. The latter is currently not supported and a solution is expected in a "Connection framework."

From a programming point of view, the visible differences between a Java client program and a Java bindings program are the following:

1. When using the MQSeries bindings, a different "package" is imported:

   ```
   import com.ibm.mqbind.*;
   ```

2. The `MQEnvironment` object cannot and does not have to be primed with TCP/IP and channel/port information.

3. As MQM and the attached process both run on the same platform, codepage translation is not of a direct concern as in the client case, where the client runs in an ASCII "little endian" mode, while MVS runs in EBCDIC "big endian" mode.
4. Used in a servlet environment, we should try to reuse the connection and the queue handles. The fact that servlet invocations belonging to the same session can be dispatched on a different threads will require a different approach to achieve this goal.

The list of Java classes and C routines involved follows. Each component is discussed later.

- parent class to instanciate the MQI framework and invoke it.
- `itso/ibm/ims/CMqi2Ims` class
- `itso/ibm/ims/Mqi2Ims` class
- `itso/ibm/ims/IMSInputOutput` class
- `itso/ibm/ims/IMSException` class
- `itso/ibm/ims/TMState` class
- `itso/ibm/ims/BuildParse` interface
- `itso/ibm/ims/[IMSTRAN][IMSModName]edit` classes implementing `BuildParse`
- libwmqjbind.so provided with the `com.ibm.mqbind` package

11.3.1 The parent Class in the MQ Solution

The parent class is part of the servlet. Following is an excerpt of the code:

```
Hashtable         thrdht = null;   1
....
if (thrdht == null) {
   thrdht = new Hashtable();
}
....
tmstate = new TMState();
jmq = new CMqi2Ims();
jmq.setReqht(reqht);
jmq.setThrdht(thrdht);
jmq.setTmstate(tmstate);
jmq.setTrclvl(trclvl);
jmq.setNodisc('Y');
try i
   jmq.doIms();
} catch(IMSException e) {
   System.err.println("TSmqic_main IMSException " +e );
}
```

These explanations relate to the preceding example:

Notes:

1 One additional member variable, a hashtable, has been declared to contain all thread-related references.

The first time we use this servlet, the hashtable is instanciated. A hashtable can contain all kinds of objects. Originally being empty, it will be populated with MQI references. This hashtable, besides other elements, is passed to the `CMqi2Ims` object, which then will be able to populate it after activating MQI references, so that next time it can reused. In our implementation, we put the `Qmgr` object into the hashtable.

11.3.2 The CMqi2Ims Class

This class, instead of instanciating `Mqi2Ims`, inherits from that class. Consequently, all variable/method members of `Mqi2Ims` are reachable by `CMqi2Ims`. This class concentrates on the building of the Input segment for IMS. After the build is complete, the class invokes the inherited `domqi` method of `Mqi2Ims`. On return from IMS, this class directs the parsing of the segments into the request hashtable. The classes used for the IBM "Input-Build" and the "Output-Parse" are dynamically determined at runtime, but must respond to the same Java implementation. In this class the same classes and methods are used as in the APPC case. For further explanation, refer to 11.2, "Connecting to IMS Based on APPC" on page 216.

11.3.3 The Mqi2Ims Class

This class contains the MQI calls wrapped in one `domqi` method. Internally, the method works via a Decision Logic Table (DLT) principle. At the end of one particular action the result is evaluated and the next action is determined.

```
/*-------------------MQCONNECT------------------------------*/
/**********************************************************/
/*                                                        */
/**********************************************************/
case MMQCONN:
   ....
   qMgr = new MQQueueManager(mqMgr);
   ....
/*--------------OPEN  REPLY QUEUE----------------------------*/
/**********************************************************/
/*                                                        */
/**********************************************************/
case MMQOPENRQ:
   ....
   int replyq_openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_INPUT_AS_Q_DEF |
   MQC.MQOO_INQUIRE | MQC.MQOO_SET;
   ....
   r_queue = qMgr.accessQueue(mqReplyQueue,
   replyq_openOptions,
   mqMgr,           //
   null,           // no dynamic qname
   null);          // no alternate user id
   ....
/*--------------OPEN IMSTO  QUEUE----------------------------*/
/**********************************************************/
/*                                                        */
/**********************************************************/
case MMQOPENIQ:
   ....
   int putq_openOptions = MQC.MQOO_OUTPUT;
   ....
   ims_queue = qMgr.accessQueue(mqImsToQueue,
   putq_openOptions,
   null,           //
   null,           // no dynamic q name
   null);          // no alternate user id
   ....
/*--------------PUT ON IMSTO QUEUE---------------------------*/
/**********************************************************/
/*                                                        */
/**********************************************************/
case MMQPUTIQ:
   /*------------------------------------------------------*/
   toims = new MQMessage();
   /*  build the message descriptor */
   toims.messageType = MQC.MQMT_REQUEST;
   toims.replyToQueueManagerName = mqReplyMgr;
   toims.replyToQueueName = mqReplyQueue;
   toims.encoding = MQC.MQENC_NATIVE;
   toims.format = "MQIMS   ";                // data in
   toims.putApplicationType = MQC.MQAT_MVS;
   toims.characterSet = mqcharset;
   toims.messageId = MQC.MQMI_NONE;
   toims.report = MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID;
   toims.persistence = MQC.MQPER_PERSISTENT;
   /*-------------------------------------------------------*/
   /*  build MQIIH header  */
   toims.writeString("IIH ");           // 0   structure identifier l
   toims.writeInt(MQIIH_VERSION_1);     // 4   version
   toims.writeInt(MQIIH_LENGTH_1);      // 8   length = 84
   toims.writeInt(MQC.MQENC_NATIVE);    // 12  Encoding
   toims.writeInt(0);                   // 16  Code Characterset
   toims.writeString(MQFMT_NONE_ARRAY); // 20  format input data 8
   toims.writeInt(MQIIH_NONE);          // 28  flags
   toims.writeString("        ");       // 32  LtermOverride
   toims.writeString("        ");       // 40  MfsMap
   toims.writeString(MQFMT_NONE_ARRAY); // 48  Reply2Format
   toims.writeString(MQFMT_NONE_ARRAY); // 56  Authentication
   toims.write(transactionId,0,16);     // 64  TransactionID
   toims.writeString(transtate);        // 80 transtate
   toims.writeString(MQICM_COMMIT_THEN_SEND);  // 81
   toims.writeString(MQISS_CHECK);      // 82
   toims.writeString(" ");              // 83
   /*-------------------------------------------------------*/
   imsTran = inOut.getImsTran();
   input = inOut.getInputString();
   ll = (short)(12 + input.length());
   /*  build LL ZZ         */
   toims.writeShort(ll);
   toims.writeShort(0);
   imstran8 = inOut.stringPad(imsTran,' ',8);
   sendString = imstran8 + input;
   toims.writeString(sendString);
   .....
/*--------------GET FROM REPLY QUEUE-------------------------*/
/**********************************************************/
/*                                                        */
/**********************************************************/
case MMQGETRQ:
   MQMessage fromims = new MQMessage();
   String  msgId = new String(toims.messageId);
   fromims.correlationId = toims.messageId;
   fromims.characterSet=mqcharset;
   // Set the get message options..
   MQGetMessageOptions gmo = new MQGetMessageOptions();  // accept the defaults
   gmo.waitInterval = timeout * 1000;
   gmo.options = MQC.MQGMO_NO_SYNCPOINT | MQC.MQGMO_WAIT |
  MQC.MQGMO_ACCEPT_TRUNCATED_MSG;
   //------------------------------
   r_queue.get(fromims, gmo, 4096);
   //------------------------------
   String corelId = new String(fromims.correlationId);
   String rmsgId = new String(fromims.messageId);
   /* message is OK  */
   dataLength = fromims.getDataLength();
   imsRecvLength = dataLength - MQIIH_LENGTH_1;
   outputSegNum = 0;
   /*  analyze MQIIH header  */
   fromims.setDataOffset(40);  // skip TO MFS ModName
   inOut.setImsModName(fromims.readString(8));
   fromims.setDataOffset(64);  // skip TO transactionId
   fromims.readFully(transactionId,0,16);
   fromims.setDataOffset(80);  // skip TO transtate
   transtate = fromims.readString(1);
   commitmode = fromims.readString(1);  // commit mode
   secscope = fromims.readString(1);    // security scope
   currOffset = 0;
   maxOffset = imsRecvLength;
   fromims.setDataOffset(84);  // skip TO messagedata
   while (currOffset  < maxOffset) {
   ll = fromims.readShort();
   zz = fromims.readShort();
   rcvstring = fromims.readString(ll-4);
   inOut.getOutputVector().addElement(rcvstring);
   outputSegNum++;
   currOffset += ll;
/* endwhile */
/*--------------CLOSE IMSTO  QUEUE---------------------------*/
/***********************************************************/
/*                                                         */
/***********************************************************/
case MMQCLOSIQ:
   ....
   ims_queue.close();
   ....
/*--------------CLOSE REPLY  QUEUE---------------------------*/
/***********************************************************/
/*                                                         */
/***********************************************************/
case MMQCLOSRQ:
   ....
   r_queue.close(); //
   ....
   break;
/*--------------DISCONNECT FROM MQM--------------------------*/
/***********************************************************/
/*                                                         */
/***********************************************************/
case MMQDISC:
   ....
```

```
    qMgr.disconnect();
    ....
```

You may recognize the steps that are required to call a response mode transaction.

Two groups of elements could be saved for the next pass from the user's browser with the Webserver:

1. Thread-related elements (Qmgr and Queue handles);
   this is a performance issue.
2. Session-related elements (TransactionID, transtate);
   this is required for conversational transactions.

A session has more a conceptual meaning than just being physical connection-related. All interactions between a user and his/her browser, till it is switched off, are considered to belong to the session.

As an example, we show the flow for a “response mode” transaction in Figure 119.

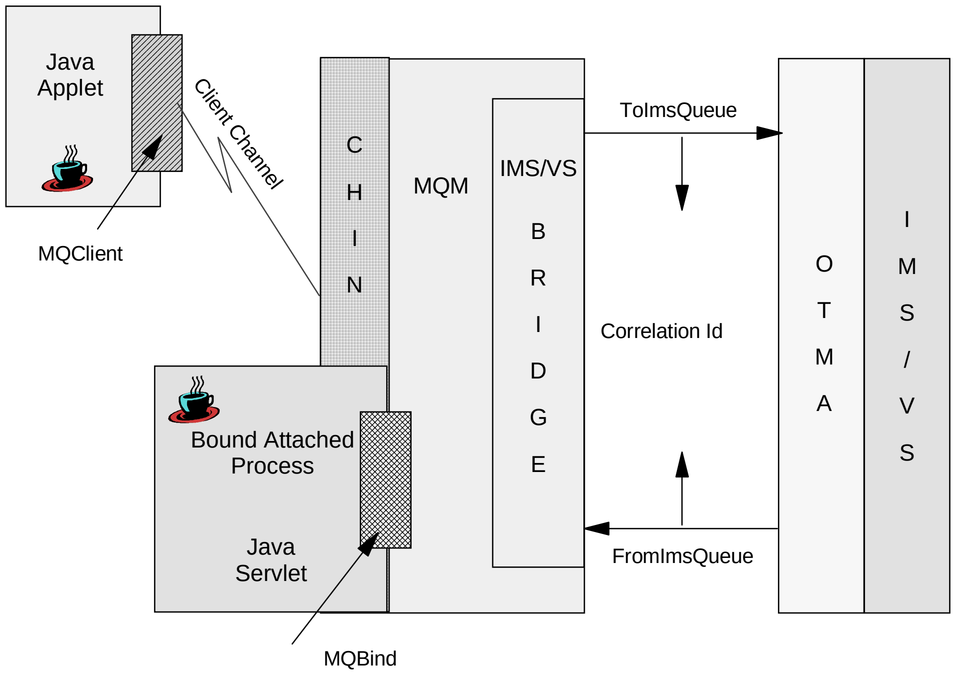


Figure 119. Message Queue Manager Connections

The reasoning behind the fact that both queues (To, From) are opened at the beginning, is that as a response mode transaction it requires an answer. This implies that if the From queue cannot be opened successfully, it is useless to send the transaction. The command/data flow is self-explanatory. The matching between the input and output message is based on the correlation ID.

The receive call supports a timeout, which is required in the case of IMS. Usually a timeout is not related to a malfunction, but is due to a queueing problem which could result from a Message Processing Region (MPR) being not available, or from long queueing delays.

11.3.4 The IMSInputOutput Class

Refer to 11.2, "Connecting to IMS Based on APPC" on page 216 for details about this class.

11.3.5 The TMState Class

```
int        mtranstate = 0;
byte[]     mtransactionId = new byte[16];
/*---------------------------*/
public void valueUnbound(HttpSessionBindingEvent be) {
   System.out.println("++TMState_valueUnbound");
   ....
   if (mqi2ims != null) {
   } else {
      mqi2ims = new Mqi2Ims();
      mqi2ims.setItranstate(mtranstate);
      mqi2ims.setTransactionId(mtransactionId);
   }
   mqi2ims.runDown();
}
/*---------------------------*/
```

The `TMState` class contains specific variables and methods for the `Mqi2Ims` class and the IMS/OTMA protocol linked to the `MqiBridge`.

The `ValueUnbound` method will be scheduled when the logical session between the browser and the Webserver is terminated, for instance after a 30-minute inactivity timeout. It may be required to schedule a rundown of the session-related MQM activity. This will certainly be required when we have an outstanding IMS conversation. The mtranstate allows us to determine whether or not we are confronted with this case.

To terminate an IMS conversation, we should send the message `/EXIT` through the queue, together with the state information that was preserved in the `TMState` object.

11.3.6 The BuildParse Interface

Refer to 11.2, "Connecting to IMS Based on APPC" on page 216 for details about this class.

11.3.7 [IMSTRAN][IMSModName]edit Classes Implementing BuildParse

Refer to 11.2, "Connecting to IMS Based on APPC" on page 216 for details about this class.

11.3.8 libwmqjbind.so

The MQBind solution is composed of classes that are used from the Java code, but the Java code itself refers to a DLL (.so) module written with the Java Native Interface (JNI). This module has to be accessible via the LIBPATH.

Part 4. Using Servlets and JavaServer Pages on OS/390

Server-side logic, controlled by a Webserver, has been popular since the introduction of the first Webserver on OS/390. In the past few years, we have seen a shift from scripting-based server-side logic (CGI) via C/C++ plugins to Java. In Domino Go Webserver Version 4.6.1 for OS/390, Java servlet support was introduced. In Lotus Domino Go Webserver Release 5.0, the servlet support was extended with functionalities, such as session management, and an applet-based browser application was added to make the configuration and management of servlets easier.

WebSphere Application Server for OS/390 V1.1 also supports JavaServer Pages, another interesting variation of performing Java server-side logic. The JavaServer Page eliminates the need to have the HTML mark-up inside the Java (business logic) programs. The JavaServer Page in itself is the mark-up (HTML) and it contains just a tiny interface to the real Java logic.

It is hard to say if the use of JSPs will be a final solution for server-side logic, but at least we can say that it is a very smooth way of developing applications and we can certainly recommend it.

In this part of the book we explain JavaServer Pages in detail and make a brief comparison with Java servlets. We also include JSP examples for your consideration. One chapter describes how to create an entire Web site including server-side logic using NetObjects Fusion. We will not talk about CGIs or C/C++ plugins, as we only focus on Java solutions.

Chapter 12. Introduction

WebAS for OS/390, included in Domino Go Webserver 5.0 for OS/390, supports a new powerful approach for dynamic Web pages: JavaServer Pages (JSP). The JSP function is based on an early version of the Sun Microsystems JSP Specification 0.91. It should be noted that future releases of the WebSphere Application Server will support 0.92 and 1.0 Sun Microsystem JSP specifications. Major changes were introduced with 0.92 and will require changes to 0.91 JSP code.

JSP will dynamically generate a servlet at the server side by inputting JSP code and automatically compile the JSP (only once per update or load) into a servlet. JSP code is simply straight HTML with imbedded Java Code. The best use of JSP is to handle the presentation logic. JSP can call JavaBeans to handle the business or data access logic. JavaBeans can access various resources on the server, including files, DBMS, CICS, IMS and so on, especially through Java connectors provided by IBM. As a result, JSP is an easy-to-use and powerful solution for generating HTML pages with dynamic content.

Besides HTML tags, Java code and JavaBeans objects, a JSP file can contains NCSA tags (special tags that were the first method of implementing server-side includes), <SERVLET> tags, and other JSP syntax.

The following is an example of a JSP code accessing a Java bean to list a directory. We are using IBM JRIO classes to perform directory listing.

```
<bean name="dirBean" type="ListDirectoryBean"
      introspect="no" scope="request">
</bean>
<% String dirName = "";
   String[] files = new String[1000] ;
 if (request.getParameter("listdir") != null)
    dirName = request.getParameter("listdir") ;
%>
<html>
<BODY  BACKGROUND="./javaback.jpg" LINK="#0000FF" VLINK="#9966FF" TEXT="#000000" >
<h2>Enter File Directory or PDS to list: </h2>
<form method=get action="ListDir.jsp" >
<input name=listdir size=50 maxlength=50 value="<%=dirName%>">
<br>  <h3> Note: for non hfs files, prefix with // </h3>
<%
 if (request.getParameter("listdir") != null)
   {
    %><h1>Directory for <%=dirName%> </h1><%
    try {
      files = dirBean.getDir(dirName);
    }
    catch (Exception e)
    { %><br><h3> No Directory file found </h3><% }
    for (int i = 0; i < files.length; i++)
      {
       %><BR><a href="../servlet/Showfile?file=<%=dirName+"/"+files[i]%>">
      <%=files[i]%></a><%
      }
    }
%>
</form>
</html>
```

Here is the `ListDirectoryBean` that uses IBM JRIO classes.

```
import java.io.*;
import com.ibm.recordio.*;

public class ListDirectoryBean {
 public String[] getDir(String directoryName)
 {  IDirectory dir = Directory.getInstanceOf(directoryName);
    if (!dir.exists())
      { throw new IllegalArgumentException(
        "com.ibm.recordio.examples.portable.ListDirectory: \"" +
        directoryName + "\" not a directory"); }
    String[] files;
    files = dir.list();
    return files;
 }
}
```

For more information on JSPs, refer to the *IBM Websphere Application Server Guide*, downloadable from or URL:
`http://www.ibm.com/software/webservers/appserv/library.html#v1o56` .

Chapter 13. How Java Servlets Work

Java servlets are Java programs that run inside a Java-enabled Webserver. Java servlets extend the function of the Webserver beyond the standard facility offered by the Webserver. Unlike other technologies that do this, such as CGI, FastCGI, ICAPI, GWAPI, and so on, Java servlets adhere to the JavaSoft servlet model. This means that in principle, not only are servlets portable between platforms, but also between Webservers from different manufacturers.

When writing a Java servlet, the Java program uses the servlet API, as specified by Sun. The Webserver is responsible for handling the servlet environment, including the loading of a servlet, support of the servlet API and the unloading of the servlet. The available OS/390 Webservers will support the Java `HttpServlet` classes, and the servlet will use the API defined within `HttpServlet` classes.

The important part to understand when using the servlet API is the usage of the following methods:

- `init()`.

  This method runs only once, at the time the Webserver loads the servlet. The servlet will be loaded either as a preload request or the first time it is requested by a client. The servlet will remain loaded until the Webserver shuts down or until it receives a manual request by the Webserver's operator interface. In general, the servlet will remain loaded from one client request after another. This provides great performance advantages. An example of a function that you would typically code in the `init` method is to do the initial database connect. This means that the database connection is performed once instead of connecting and disconnecting for each database query.

- `destroy()`.

  This method is also performed only once. It is invoked when the Webserver is stopped. An example of a function to code in `destroy` is to do a disconnect of a database, in case the connection was performed in the `init()` method.

- `doGet()` and `doPost()`. One of these methods is called for each service request from a browser and is the heart of the servlet. The incoming method from the browser could be GET or POST. A GET request will invoke the `doGet` method of a servlet and a POST request will invoke the `doPost` method. In either case, the `doGet()` or `doPost()` will handle the main logic of the servlet.

Examples of Java servlets are shown in various sections in this book, one of them using a connection to DB2.

For full documentation on the servlet API, refer to the JavaSoft Web site at URL: `http://www.javasoft.com`

Chapter 14. How a JavaServer Page (JSP) Works

JSP technology has three key elements:

- Reusable components
- Scripting language
- Compiled-page objects

At development time, a developer can use reusable components and scripting language with HTML. At runtime, written JSP code is translated to other executable language, and then executed as a compiled page object.

In ServletExpress, JSP supports JavaBeans as reusuable components, Java as scripting language and servlets as a compiled page object. After all, the JSP is scripting language, but it works as a servlet internally.

14.1 Execution Process of JSP Code in ServletExpress

This section decribes how the ServletExpress "engine" handles JSP. Understanding this process helps you solve JSP-related problems.

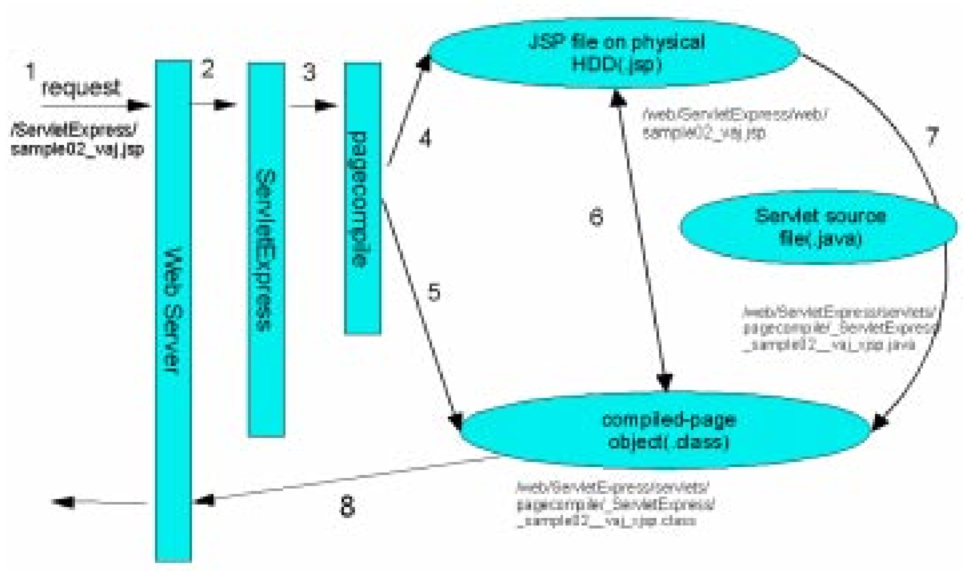


Figure 120. Execution Process of JSP

Attention

The path shown in Figure 120 may be different in your case. It depends whether you have WebSphere Application Server for OS/390 V1.1 installed or ServletExpress. Refer to 4.2.1, "Configuring WebSphere Application Server for OS/390 V1.1" on page 36 for details regarding the location of the Webserver files.

Referring to Figure 120, the following steps take place when requesting a JSP:

1. The Web browser or servlet code requests a JavaServer Page file. The JavaServer Page file is identified to the server by a `.jsp` extension.
2. The Webserver routes this request to ServletExpress.

   Note: In order to enable the Webserver to do so, the following statement should be in the httpd.conf file.

   ```
   Service    /*.jsp /usr/lib/libadpter.so:AdapterService
   ```

3. ServletExpress routes this request to the pagecompile servlet.

   Note: ServletExpress will create a directory called pagecompile in the `/usr/lpp/ServletExpress/servlets` directory and WebAS will do so in `/usr/lpp/WebSphere/AppServer/servlets`. We found that we had to change the permission bits for the `/usr/lpp/ServletExpress/servlets` directory to 777 to allow for the creation of the pagecompile directory. If you do not change the permission bits of the servlets directory, you may receive the following message:

   ```
   Error getting compiled page.
   Cannot create directory /usr/lpp/ServletExpress/servlets/pagecompile
   ```

4. The pagecompile servlet looks for the corresponding physical file to be URL-requested. At this time, the pagecompile servlet uses a request routing rule directed by the httpd.conf file.

   Note: The following statement instructs what configuration file is used in the jvm.properties file.

   ```
   # Properties for Domino Go
   ncf.native.httpd.cnf.path=/web/java14/httpd.conf
   ```

5. The pagecompile servlet checks whether the corresponding compiled page object (servlet bytecode) exists or not.
   When the URL is:

   ```
   /ServletExpress/sample02_vaj.jsp
   ```

   then the corresponding servlet will be:

   ```
   <SEroot>/servlets/pagecompile/_ServletExpress/_sample02__vaj_xjsp.class
   ```

   Note: The following two parameters for the pagecompile servlet specify which directory is used in order to store the compiled page object.

workingDir	By default, <SEroot>/servlets
packagePrefix	By default, pagecompile

   By default, all compiled page objects (servlets) belong to the subpackage pagecompile, and the root directory for these classes is <SEroot>/servlets. Therefore compiled page objects are placed in a subdirectory of `<SEroot>/servlets/pagecompile/`.

6. At this moment, the servlet code contains a qualified name and a written date/time of the corresponding JSP.
7. If the servlet code does not exist or has a different date/time than the JSP file, the pagecompile servlet translates the JSP to servlet code, and then invokes javac in order to compile the servlet to bytecode.
8. The pagecompile servlet invokes the servlet code corresponding to the JSP file and sends the results to the Web browser or servlet through the Webserver.

Chapter 15. Designing a Server_side Plugin

There are several ways to develop Java Server Pages. You can edit the files using any editor, or an application designed specifically for JSPs.

In our examples, we use a combination of JSP, Servlets, JavaBeans and HTML.

NetObjects ScriptBuilder was designed to assist with the development of JSPs. NetObjects ScriptBuilder has many features and benefits.

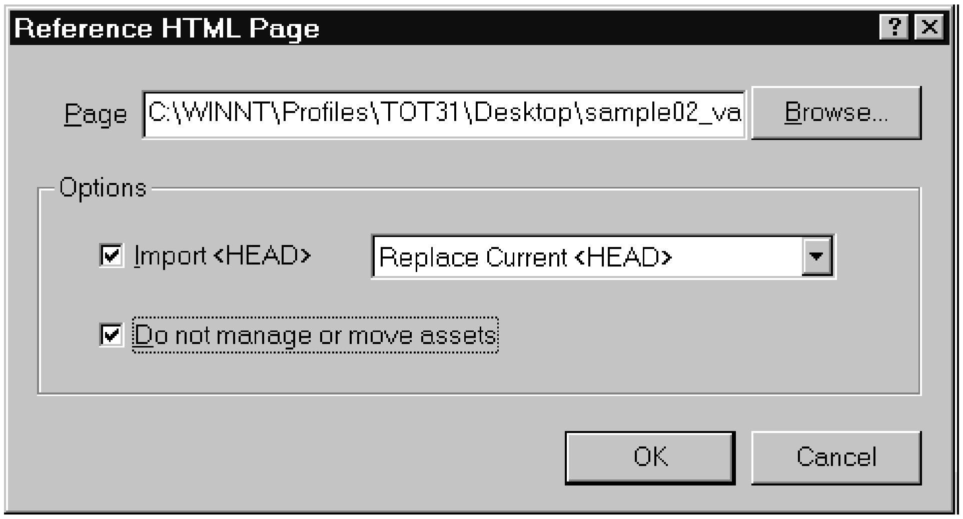
- Color syntax highlighting and syntax checking: NetObjects ScriptBuilder visually identifies pieces of code to speed and ease development and debugging.
- Automated tasks: NetObjects ScriptBuilder speeds development and reduces errors through one click access to AutoScripting, automatic tag insertion, and user-definable code templates. Document/Object Map: Using NetObjects ScriptBuilder developers can simply navigate complex Web pages or object code by visually browsing embedded functions and scriptable objects.
- Support for multiple scripting languages: NetObjects ScriptBuilder allows scripters to develop in the language most familiar to them, while having the flexibility to use other technologies without changing their development environment.

The CD that comes with this redbook also contains a 30-day trial version of NetObjects ScriptBuilder; see the CD for additional details.

You can also integrate the development of JSPs in NetObjects Fusion. If you already have JSP code, you can attach it to a page in the site as external HTML. Once this is set up and if you have NetObjects ScriptBuilder installed, NetObjects Fusion will call ScriptBuilder to edit the JSPs. The process of adding existing JSPs is very easy, but there are two ways to do this:

- You can select **File**, then select **Reference HTML** from menu.
- Or, you can click the **external HTML** button from the left toolbar.

These options lead to a similar pop-up menu. Figure 121 on page 254 shows the resulting screen.


Reference HTML Page

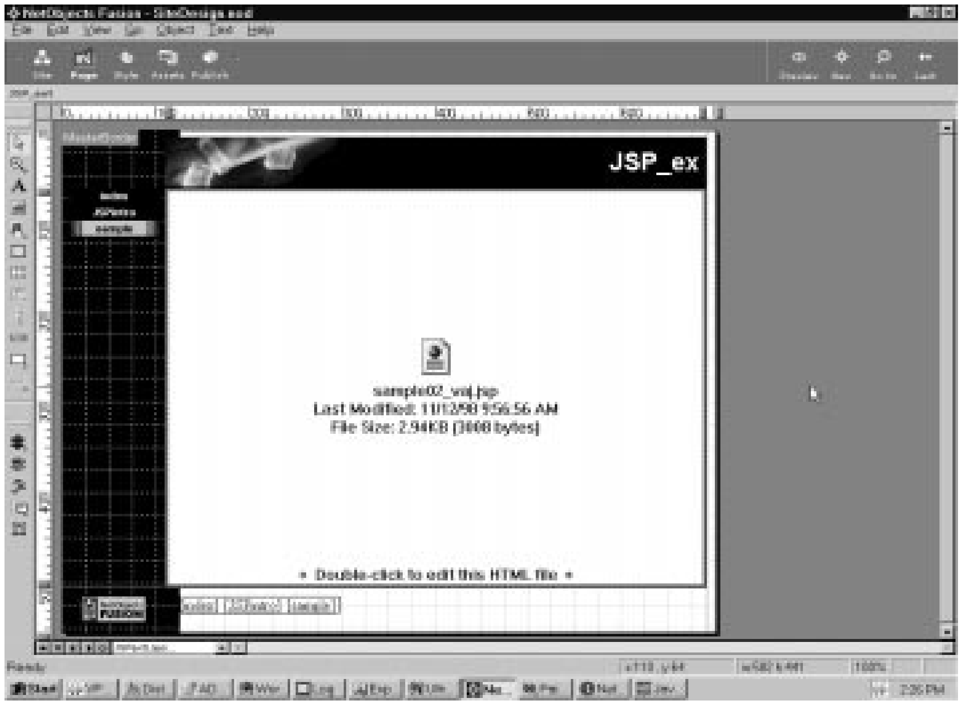
Page: C:\WINNT\Profiles\TOT31\Desktop\sample02_va

Options

[✓] Import <HEAD> Replace Current <HEAD>

[✓] Do not manage or move assets

Figure 121. Reference HTML Page Window in NetObjects Fusion


sample02_vaj.jsp
Last Modified: 11/12/98 9:56:56 AM
File Size: 2.94KB (3008 bytes)

• Double-click to edit this HTML file •

Figure 122. Result Screen of Import External JSP in NetObjects Fusion

In this case, NetObjects Fusion will generate a new JSP file combined of two pages at publishing. Of course, the extension of this file should be set to .jsp and the name of the new file (not the one of the original external JSP code), should be used when the code is called.

If we already have servlet code, we can include it in a page by means of a servlet tag. In this case, the extension of the file should be .shtml or .jsp. But if the file contains just HTML tags and servlet tags, the extension .shtml is recommended. The reason is that some problems may occur when the servlet tag calls a complex servlet in JSP files. NetObjects ScriptBuilder can assist with adding the servlet tag, or you can add it manually.

In case of writing the Java code in the page, add a text object to the place where the servlet will be located, right-click the object, and select **Layout HTML**. After the page HTML dialog appears, click **Beginning of Body** tab and write the required servlet tag.

```
<SERVLET NAME="JDBCServlet2" CODE="JDBCServlet2.class" CODEBASE="/servlet">
</SERVLET>
```

15.1 Design of Each Page

How can we construct each page in a Web application? Both JSPs and servlets are able to execute presentation and business logic. Both can call and use Java classes or JavaBeans. In addition, servlets can call JSPs. Therefore, there can be various combinations for implementing Web applications.

15.1.1 Things to Consider

We all want to be able to write code easier and faster. We all also want the code to be easy to read. In addition, we want to be able to continuously change the code. For those purposes, we need a separation of logic in the Web application. This is where JSP can help.

Using JSP, we can concentrate at one thing at a time, be it either the presentation logic or the business logic. At development time, you may start with only writing HTML, eventually in a prototyping approach with the end-users. Then, we can extend the application with Java code to perform the business logic. At the end, we can add a small amount of Java code to the HTML in order to connect the two kinds of logic.

Because separation of logic is the most important merit of JSP, we need to maximize this when we write a Web application. There are several models that can be used. The following are three that were looked at during this residency.

15.1.2 Model 1 - JSP and JavaBean

A request from the Web browser can be sent to the Webserver, requesting a JSP using a URL exactly like a normal HTML page. The JSP code contains static HTML tags as presentation logic and calls Java classes containing business logic. (This JSP code itself should not have business logic. To perform business logic, many components having elementary functions are needed. These may be for DB access, CICS, IMS or other back-end resources).

This is the recommended model to use. In this way you can assign the development of the physical page to a site designer and the bean or servlet to an application programmer. This will also allow you to create Beans and servlets that can be used by several JSPs.

Here is an example of how a JSP can be created using this model.

1. First you build a base HTML page that matches the site style and contains the elements that you want displayed. This example mssearch2.jsp generates a list box with search results and a form allowing you to repeat the search with additional information. The base HTML looks like the following.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 FINAL//EN">
<bean name="jdbcBean" type="JDBCBean" introspect="no" scope="request">
</bean>
<HTML>
<HEAD>
  <TITLE>Search</TITLE>
  <META NAME="Author" CONTENT="ITSO Residency">
</HEAD>
<BODY  BACKGROUND="../javaback.jpg" LINK="#0000FF" VLINK="#9966FF" TEXT="#000000" TOPMARGIN=0
LEFTMARGIN=0 MARGINWIDTH=0 MARGINHEIGHT=0>
<table border=1 cellmargins=0><tr><td>
<FORM NAME="Search" action="../mssearch2.jsp" method="get">
<font size=+2><b>Search</b></font>
<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0  WIDTH=190>
<TR><TD ALIGN="LEFT"><FONT COLOR="000000">Manufacturer</FONT></TD><TD ALIGN="LEFT">
<INPUT NAME= "manlist" VALUE="" SIZE=10 MAXLENGTH=30 onBlur="upper_case(this)"></TD></TR>
<tr><td colspan=2 ALIGN="right"><INPUT TYPE="SUBMIT" NAME="dbaction" VALUE="List" ></form></td></tr>
</table>
</table>
<FORM NAME="detail" action="/msdetail2.jsp" method="POST" target="detail">
<center>
<SELECT NAME=manlist size=10 onChange="launch(this)">
<!-- You want to dynamically generate the select options -->
</select>
</TABLE>
</center>
Select a Manufacturer from the list for details.<br>
<FORM NAME="Add" action="../msdetail.jsp method="get"><INPUT TYPE="SUBMIT" NAME="dbaction" VALUE="New" >
</form>
</FORM>
<p><font size=-1>JDBC Bean from a JSP Sample</font>
</BODY>
</HTML>
```

2. Next you add the definition of the JavaBean and the method invocations to get the data. The methods used are defined by the application programmer. In this case they are connect, list, and disconnect. You also need to define some local variables to get the passed parameter and a vector to hold the results. You could also add two Javascript functions to the HTML page to preprocess the data before another request is made.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 FINAL//EN">
2 <bean name="dataBean" type="dataBean" introspect="no" scope="request">
</bean>
<bean name="jdbcBean" type="JDBCBean" introspect="no" scope="request">
</bean>
<HTML>
<HEAD>
<%
String manlist = request.getParameter("manlist");
String temp;
Vector ManufacturerList;
if (manlist == null) {
  manlist = "@";
  }
manlist = manlist.toUpperCase();
3 jdbcBean.connect();
ManufacturerList = jdbcBean.list_Manufacturer(manlist);
jdbcBean.disconnect();

%>
  <TITLE>Search</TITLE>
  <META NAME="Author" CONTENT="ITSO Residency">
</HEAD>
<BODY  BACKGROUND="../javaback.jpg" LINK="#0000FF" VLINK="#9966FF" TEXT="#000000" TOPMARGIN=0
LEFTMARGIN=0 MARGINWIDTH=0 MARGINHEIGHT=0>
<script language="JavaScript">
<!--
function upper_case(field){
  f1=field.value.toUpperCase();
  field.value=f1;
  return(true);}

function launch(list) {
 var urlstring = "../msdetail2.jsp?dbaction=VIEW&manlist="+list.optionsfflist.selectedIndex“.value;
 window.open(urlstring,"detail");}
//-->
</script>
<table border=1 cellmargins=0><tr><td>
<FORM NAME="Search" action="../mssearch2.jsp" method="get">
<font size=+2><b>Search</b></font>
<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0  WIDTH=190>
<TR><TD ALIGN="LEFT"><FONT COLOR="000000">Manufacturer</FONT></TD><TD ALIGN="LEFT">
<INPUT NAME= "manlist" VALUE="" SIZE=10 MAXLENGTH=30 onBlur="upper_case(this)"></TD></TR>
<tr><td colspan=2 ALIGN="right"><INPUT TYPE="SUBMIT" NAME="dbaction" VALUE="List" ></form></td></tr>
</table>
</table>
rest is identical
```

3. The last piece to add is the Java code to generate the selects option tags.

```
<SELECT NAME=manlist size=10 onChange="launch(this)">
<%
   if (ManufacturerList.isEmpty()) {
        %><option selected>No search string entered
          </select>
        <% }
   else {
       for (int i = 0; i < ManufacturerList.size(); i++) {
          %><option value="<%=ManufacturerList.elementAt(i)%>">
         <%=ManufacturerList.elementAt(i)%>
         <% } }
     %>

</select>
```

15.1.3 Model 2 - Combination of JSP and Servlet

Another model is also possible, in which the servlet performs business logic and calls a JSP or HTML as presentation logic. If the JSP is called, it can retrieve data from the servlet through a JavaBean or an attribute of the `HttpRequest` object. The servlet can also call multiple JSPs or HTML sequentially to assemble one HTML response. This is done by coding a callpage method in the servlet.

15.1.4 Model 3 - JSP Only

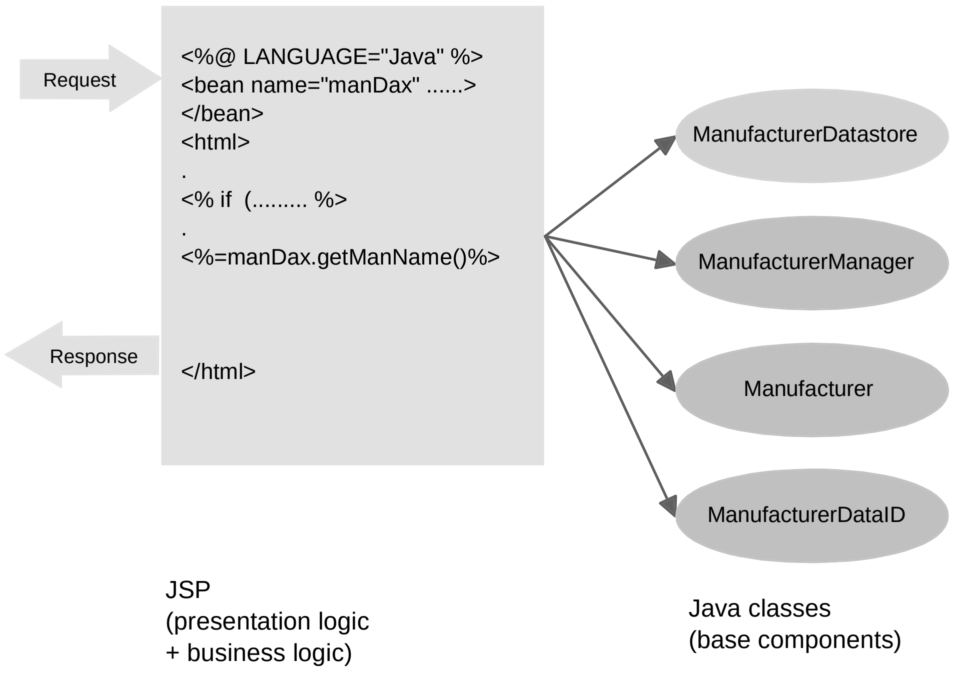


Figure 123. JSP Only

This model is not good from the view of separation of logic, because JSP code has both presentation and business logic. But it is more flexible than the previous two models, so it is useful for the quick development of simple code.

15.2 Writing JSP Code and Servlets

In the following sections, we discuss various considerations to keep in mind when writing JSP code and servlets.

15.2.1 NullPointer Exception

When we write a JSP or a servlet, we should be very aware of the “NullPoint Exception” in advance. If we encounter this problem, it may take a long time to debug. The NullPoint Exception happens when a method or property of an object with a null value is referenced. In case of an array, just referencing the index may cause the exception.

In any case, we should not let the code reference null values. This is very important from a productivity viewpoint.

We usually use `getParameter()` or `getParameterValues()` methods in order to get parameters from the Web request. At this time, we should always keep in mind that these methods may return null. Therefore, we either should write code that checks whether return values are null or not, and then does not use the result, or replaces it with another value if it is null.

Also note that the following code in JSP may cause NullPoint Exception:

```
<SELECT><%=getXXX().trim()%>
```

If the `getXXX()` method returns a null value, the code throws NullPoint Exception because it cannot reference the `trim()` method.

Similarly, we should mention the case of data access beans generated by the VisualAge for Java Data Access Builder. When we use these beans, we usually use `setXXX(...)` in order to set the property. If we use these methods with "" (an empty) value, an SQL Exception may happen when executing the SQL statement. Therefore, make sure not to set a property with an empty value, usually leaving it as null.

For the `getXXX()` method in JSP, we recommend you use the <insert> tag, if possible. Because this tag is able to prevent NullPoint Exception for itself, it is safer to use.

15.2.2 OutOfMemory Exception

If you get the OutOfMemory Exception, you need to check the <repeat> tag in the JSP. If you do not set an “end” attribute for the <repeat> tag, the code between <repeat> and </repeat> is executed until the index reaches 2,147,483,647 or `IndexOutOfBoundsException` happens. If `IndexOutOfBoundsException` does not happen, the code falls into a situation similar to an infinite loop, finally causing the OutOfMemory Exception.

In this case, examine the Java code to which the JSP is translated by the Webserver and make sure the `IndexOutOfBoundsException` is generated in time.

If this is not the case, increase the memory size for the JVM. You can specify this with the ncf.jvm.mx attribute in the jvm.properties file, as follows:

```
..........
# Max Java Heap Size
ncf.jvm.mx=350108864
..........
```

15.2.3 Debugging JSP Code

If there are any problems with your JSP code, you may see the following message in your browser:

```
Error getting compiled page.
Unable to compile /usr/lpp/ServletExpress/servlets/pagecompile/_IWT/_PARThtml_xjsp.java
```

All compiler errors will be written to the ncf.log file if you have ncf.log tracing turned on in your jvm.properties file. You can also debug your servlet by manually compiling the java code that was produced by the Web server in the /pagecompile/* directory.

15.2.4 The <insert> Tag

Important

At the time of writing, the JSP specification was still undergoing changes. At all times, we refer you to the URL of the JSP specification to verify your code. We also refer you to Chapter 12, “Introduction” on page 247 for the levels of JSP supported by WebSphere Application Server for OS/390 V1.1.

The Webserver supports very useful tags for JSP. One of them is the <insert> tag we mentioned in 15.2.1, “NullPointer Exception” on page 258. Following is the syntax of this tag:

```
<insert requestparm=pvalue requestattr=avalue bean=name
property=property_name(optional_index).subproperty_name(optional_index)
default=value_when_null>
</insert>
```

We should use this in requestparm (for parameter), requestattr (attribute of request) and bean. When the servlet calls JSP code, we can set the attribute of request and bean in the servlet in order to transfer some data to the JSP using setAttribute().

There are two things to consider when using this tag:

1. We cannot reference the property of attributes with the <insert> tag. For example, if an attribute is an array of String, we cannot use its element with the <insert> tag. (Of course, if it is just String, we can.) Therefore, if we need to reference the property of an object transferred from a servlet in a JSP with the <insert> tag, the object is declared by a <bean> tag in JSP.

2. The object should be “JavaBeans” in order to reference its property with the <insert> tag. In other words, the Java class should have some methods for the property, and the signature of the method should abide by the JavaBeans specification. For example, the following tags work correctly when manList bean has `String getManName(int)` and `String getManListName()`:

   ```
   <insert bean=manList propety=manName(1)></insert>
   <insert bean=manList propety=manListName></insert>
   ```

Chapter 16. Samples of JSP/Servlet

In this chapter we provide JSP and servlet examples.

16.1 JSP and JavaBean

Important

At the time of writing, the JSP specification was still undergoing changes. At all times, we refer you to the URL of the JSP specification to verify your code. We also refer you to Chapter 12, “Introduction” on page 247 for the levels of JSP supported by WebSphere Application Server for OS/390 V1.1.

This code is an example of a JSP accessing a DBMS. Instead of using a servlet, this example uses a JavaBean `JDBCBean` that has most of the business logic. The JSP code just gets the result from the bean and then puts it between HTML tags. The sample is started as ms2.html. It creates two frames. The left frame is for search requests and results, and the right is for individual details.

The following example is the search frame. The file is mssearch2.jsp.

```
1 <%@ LANGUAGE="Java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 FINAL//EN">
2 <bean name="dataBean" type="dataBean" introspect="no" scope="request">
</bean>
<bean name="jdbcBean" type="JDBCBean" introspect="no" scope="request">
</bean>
<HTML>
<HEAD>
<%
String manlist = request.getParameter("manlist");
String temp;
Vector ManufacturerList;
if (manlist == null) {
  manlist = "@";
  }
manlist = manlist.toUpperCase();
3 jdbcBean.connect();
ManufacturerList = jdbcBean.list_Manufacturer(manlist);
jdbcBean.disconnect();

%>
  <TITLE>Search</TITLE>
  <META NAME="Author" CONTENT="ITSO Residency">
</HEAD>
<BODY  BACKGROUND="../javaback.jpg" LINK="#0000FF"
    VLINK="#9966FF" TEXT="#000000" TOPMARGIN=0 LEFTMARGIN=0 MARGINWIDTH=0 MARGINHEIGHT=0>
<script language="JavaScript">
<!--
function upper_case(field){
  f1=field.value.toUpperCase();
  field.value=f1;
  return(true);}
```

```
function launch(list) {
 var urlstring = "../msdetail2.jsp?dbaction=VIEW&manlist="+list.optionsfflist.selectedIndex“.value;
 window.open(urlstring,"detail");}
//-->
</script>
<table border=1 cellmargins=0><tr><td>
<FORM NAME="Search" action="../mssearch2.jsp" method="get">
<font size=+2><b>Search</b></font>
<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0  WIDTH=190>
<TR><TD ALIGN="LEFT"><FONT COLOR="000000">Manufacturer</FONT></TD><TD ALIGN="LEFT">
    <INPUT NAME= "manlist" VALUE="" SIZE=10 MAXLENGTH=30 onBlur="upper_case(this)"></TD></TR>
<tr><td colspan=2 ALIGN="right"><INPUT TYPE="SUBMIT" NAME="dbaction" VALUE="List" ></form></td></tr>
</table>
</table>

<FORM NAME="detail" action="/msdetail2.jsp" method="POST" target="detail">
<center>
<SELECT NAME=manlist size=10 onChange="launch(this)">
4 <%
   if (ManufacturerList.isEmpty()) {
        %><option selected>No search string entered
          </select>
        <% }
   else {
       for (int i = 0; i < ManufacturerList.size(); i++) {
          %><option value="<%=ManufacturerList.elementAt(i)%>">
    5 <%=ManufacturerList.elementAt(i)%>
         <% } }
     %>

</select>
</TABLE>
</center>
Select a Manufacturer from the list for details.<br>
<FORM NAME="Add" action="../msdetail.jsp method="get">
<INPUT TYPE="SUBMIT" NAME="dbaction" VALUE="New" ></form>
</FORM>
<p><font size=-1>JDBC Bean from a JSP Sample</font>
<% System.out.println("We made it to the end"); %>
</BODY>
</HTML>
```

1 Specify the script language. At this time, only Java is supported. The <%@ %> tag is used to direct a property that affects the entire JSP code.

2 Declare the JavaBean (=java class) used by the JSP. The name of the tag and attribute have changed in the JSP specification. This code is based on WebSphere Application Server 1.1.

3 The connect(), list_Manufacturer(manlist), and disconnect() methods are used to call the javabean and perform the data access passing it a qualifier that is used for the search. The <% %> tag is used to identify Java code inserted between HTML tags.

4 In Java Server Pages, Java code is inserted between HTML tags; if you looked at the Java servlet generated from the JSP, you see that HTML is transformed to corresponding Java code and then inserted between the other Java code. In order to understand this part, you need to examine the Java code first, and then imagine the other HTML code in between.

5 One result value is inserted between HTML. The <%= %> tag is used to insert just one value. Note that you should not use a semi-colon (;) at the end, since the <%@ %> tag does not require it.

16.2 Servlet and JSP

In this example, a servlet receives a request from a Web browser, processes business logic, and returns the result using JSP code.

This servlet has the same role as `JDBCBean` in the previous example.

Note: Part of the sample code is omitted.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
1 import com.sun.server.http.*;
import java.sql.*;
2 import com.ibm.ivj.eab.dab.*;
2 import com.ibm.itso.ls3604.jsp.dbAccess.*;

public class Sample_man_db extends HttpServlet {
 private Manufacturer ivjman = null;
 private ManufacturerDatastore ivjmanDatastore = null;
 private ManufacturerManager ivjmanManager = null;

 private boolean manListRequest = false;
 private String requestType = null;

 private IVector vec = new IVector();

 private String manNameList[] = null;
 private String manAttr[] = null;

public void service(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException
{

  3 connect();
  4 setResHeader(response);
  5 setFromRequest(request);
  6 execute();
  7 genBO();
  8 genHTML(request, response);
  9 disconnect();
}
public void genBO() throws ServletException {
 try {
  // ManListRequest("", LIST, UPDATE, SAVE)
  if (isManListRequest()) {
   manNameList = new String [vec.size()];
   for (int i = 0 ; i < vec.size() ; i++) {
       manNameList[i] = ((Manufacturer)vec.elementAt(i)).getMan_name();
   }
  }
  // SEARCH(SELECT ONE ROW)
  else if (requestType.equals("SEARCH")) manAttr = getman().getAttributeStrings();
  } catch (java.lang.Throwable ivjExc) {
   handleException(ivjExc);
  }
}
public void genHTML(HttpServletRequest request,HttpServletResponse response)
                throws ServletException, IOException {

 if (isManListRequest()) {
             ((HttpServiceRequest)request).setAttribute("manNameList", manNameList);
             ((HttpServiceResponse)response).callPage("/cheong/ManList.jsp", request);
 }

 if (isManSearchRequest()) {
  ((HttpServiceRequest)request).setAttribute("manAttr", manAttr);
  ((HttpServiceResponse)response).callPage("/cheong/ManSearch.jsp", request);
 }
 if (isManAddRequest()) {
  ((com.sun.server.http.HttpServiceResponse)response).callPage("/cheong/ManAdd.jsp", request);
 }
}
}
```

1 This is needed in order to use `com.sun.server.http.HttpServiceResponse.callPage()`..

2 These are packages of base components.

3 **4** **5** **6** **7** **8** **9** These are methods that carry out part of the task handling request.

3 Connect to DB2, using the method of the base component.

4 Set the HTTP header. This method uses `HttpServletResponse.setContentType()`, `setHeader()`, and `SetDateHeader()`. It prevents the Web browser from caching.

5 This method sets properties of base components that are required for processing the request from its parameters.

6 Execute the SQL statement.

7 In order to pass the result of the execution, prepare the data structure.

8 Call the JSP and pass the data structure of the result.

9 Disconnect from DB.

16.2.1 JSP

```
1 <%@ language=JAVA%>
2 <% String manNameList [] = (String [])request.getAttribute("manNameList"); %>

<HTML>
<HEAD>
<TITLE>Manufacturer Information</TITLE>
</HEAD>

<BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000FF" VLINK="#663366"
ALINK="#000088" BACKGROUND="/redfade.gif" NOSAVE>
<spacer type=block width=100 height=100 align=left>
<center><B><font size=+3>Manufacturer Information</font></center></B>
<br><br><br>
<blockquote>
<form name=dman method=get action="/servlet/Sample_man_db">
<table cols=2 width=90% ><tr>
<td align=right width=100>Manufacturer Name:</td>
<td>
<select name=manlist>
3   <% for (int i = 0 ; i < manNameList.length ; i++) {%>
  <option selected><%= manNameList [i]%>
  <% } %>
</select>
<input type=submit value=SEARCH name=button>
</td></tr>
<tr><td align=right>Address:</td>
<td><input name=address size=30 &nmaxlength=30 value="">

-----------  omitted ------------

</tr>
</table>
<center>
&nbsp;
</center>
<center>
```

```
<input type=submit value=LIST name=button>
<input type=submit value=ADD name=button>
</form>
</blockquote>
</BODY>
</HTML>
```

1 Declare Java as the script language for JSP.

1 Receive the attribute transferred from the servlet. String elements cannot be referenced with the <insert> tag. To use the insert tag, you should create a new JavaBean having this attribute as a property in the servlet and declare it with the <bean> tag in the JSP.

3 Insert information received from the servlet between HTML tags.

Part 5. Using VisualAge for Java ET/390 and HPJ/390

For serious Java application development, an Integrated Development Environment (IDE) has become a *must*. When developing Java you must be able to control the source. Also, productivity-increasing features, such as an intelligent editor and a Visual Composition editor, can make life much easier. IBM's premier IDE for Java is VisualAge for Java. Today, VisualAge for Java Version 2.0 is available and in October 1998, IBM released the specific version for OS/390 developers, called VisualAge for Java, Enterprise edition for OS/390.

VisualAge for Java, Enterprise edition for OS/390 offers a full cross-platform Integrated Development Environment (IDE), giving you the option of performing activities on OS/390 remotely from the workstation by means of GUIs. The specific OS/390 facilities are grouped under the name Enterprise Toolkit/390 (ET/390). Also, part of VisualAge for Java, Enterprise Edition for OS/390 is an OS/390 compiler and runtime.

In this part of the book, we provide you with an overview and examples of the specific OS/390-related features. This part is not meant as a full description of VisualAge for Java, but it will help you to get started with ET/390.

Attention

In this redbook we use the term HPJ for the native code compiler for Java on OS/390 because HPJ was IBM's internal code name.

However, the official product name is VisualAge for Java, Enterprise Edition for OS/390.

Chapter 17. Introduction

Attention

HPJ/390, ET/390 or Native Code Compiler; what is what?

In this book and other literature you may find different terms regarding the new OS/390 functionalities of VisualAge for Java: HPJ/390, ET/390, Native Code Compiler.

- HPJ/390 is known as the OS/390 component that you will need to support the ET/390 to perform its cross-platform activities. It includes a compiler and a runtime.
- ET/390 is the specific “plugin” toolset that gives you the specific OS/390 features on top of the VisualAge for Java Enterprise edition.
- A Native Code Compiler is a generic term for all platforms and actually means a compiler that turns source (also Java) into object code.

Before describing the details of using ET/390 or HPJ/390, we can compare the purpose of each product.

HPJ/390 is a compiler. It is supported by IBM on a number of IBM platforms. Its purpose is to take Java bytecode and produce machine code for the specific platform. Machine code can execute much more quickly than interpreted bytecode. Of course, the machine code, unlike the bytecode, is not portable to other platforms. This is not a problem if the bytecode is not discarded.

ET/390 is a component of the IBM VisualAge for Java Integrated Development Environment (IDE). VisualAge for Java runs on a workstation (for example, Windows 95/NT). To develop a Java application for OS/390, it is possible to use the VisualAge Integrated Development Environment (IDE) on the workstation. The ET/390 component will transparently send the Java source and/or class files to OS/390. Once on OS/390, there are ET/390 options that allow the application to run in the OS/390 JVM or to be built into native code using HPJ/390. Figure 124 on page 270 illustrates the integration of the VisualAge for Java IDE and the OS/390 runtime environment.

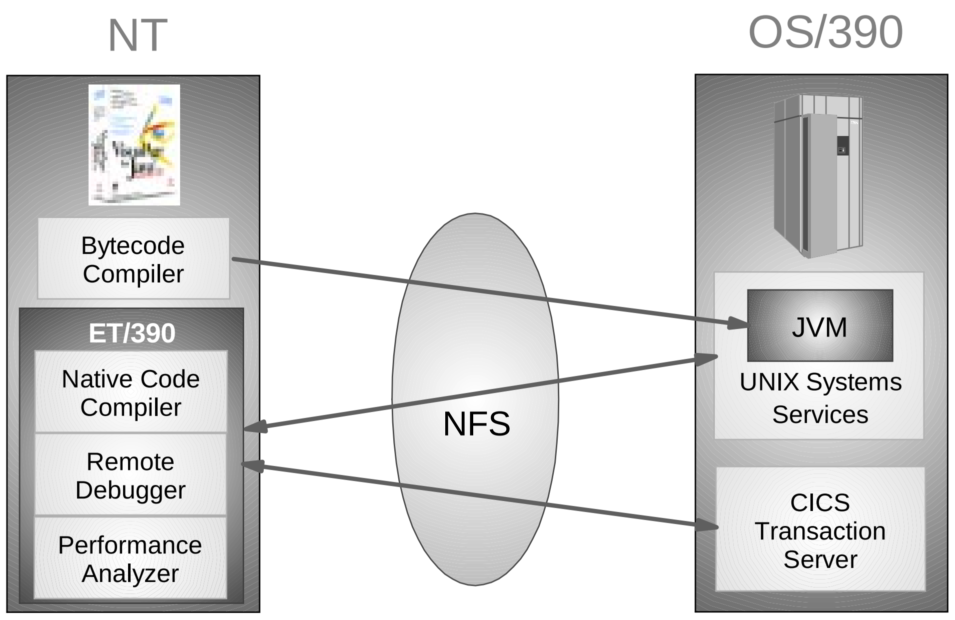


Figure 124. Integration of the VisualAge for Java IDE and OS/390

You may choose not to use the facilities provided by ET/390. Here is a list of some possible development scenarios:

1. Develop directly on OS/390.

   In this case, you would typically use a terminal emulator supported by OS/390 and use a command line editor to develop your code[5]. The techniques are:

   a. Using telnet

      Having established the telnet connection, use the vi editor to create/edit the source and use `javac` or `hpj` to compile the source.

   b. Using OMVS to start the UNIX shell on the 390

      Within the OMVS shell, use `oedit` to create/edit the source and use `javac` or HPJ to compile the source.

2. Develop on a workstation (for example, Windows 95/NT) and port to OS/390.

   The techniques are:

   a. Using ftp

      Use any PC-based text editor to develop the source. Maybe use javac even on the workstation to produce the class files. Then use ftp to transfer the files from the PC to the 390.

   b. Using NFS

      When using NFS, you can work directly with OS/390 directories and files on your PC as if they would be stored locally on your hard disk. When using NFS, you have to mount the OS/390 HFS to be used first before you can access it.

[5] In OS/390 UNIX System Services, the ISPF editor is supported by the `oedit` command

Important

With OS/390, an NFS mountpoint coincides with the mountpoint of *one* HFS (Hierarchical File System). For *each* HFS to be used, you need to do an NFS mount on the workstation. When looking at the directory in an HFS mounted to your workstation, you cannot “jump” to a directory in another HFS that is not mounted.

When an HFS and its directories and files have been mounted, you will be able to operate on the files with a PC editor.

Note: The use of both FTP and NFS requires that their matching servers be running on OS/390. Care must be taken that ASCII-to-EBCDIC translation occurs for source files, but not for class files.

The ET/390 extensions to VisualAge for Java combine and encapsulate most of these scenarios into a seamless environment. The result is a sophisticated Integrated Development Environment using the powerful features of the GUI development. Applications can be built using the visual editing tools and deployed onto the 390 from within the IDE with the click of a button or two.

Chapter 18. Using HPJ/390 - Scenarios

In this publication, we use High Performance Java (HPJ) as the name for the Native Code Compiler for Java on OS/390. A native code compiler is also available on other platforms (for example, Windows NT).

The input to HPJ/390 is a bytecode file, like a file that is output from javac and has the .class file extension. The output from HPJ/390 is either a dynamic load library (DLL) or a program object (executable).

Using the HPJ/390 compiler is very similar to using any other compiler. It has a simple command line interface with a wide range of options. Many of the options are not commonly used.

Here is an example of a simple command to produce a program object from a Java class file:

```
hpj Example1
```

Notes:

- This command will produce a program object with the name a.out.
- The input file must be named Example1.class.
- The directory containing Example1.class must be in your CLASSPATH environment variable.
- The class Example1 is not part of a package.

The final three rules are applicable to all Java compiling.

A second example is a little more realistic:

```
hpj -o=eg2 example.Example2
```

Notes:

- This command will produce a program object with the name eg2.
- The input file must be named Example2.class.
- Example2.class must be located in a directory named example.
- The directory named example must be in your CLASSPATH environment variable.

The following are more complex scenarios using HPJ/390.

18.1 Scenario I

Imagine that you want to build a Java executable from a package that has two classes, as follows:

- `MyExe.Main` contains the main method, and it references `MyExe.refClass`.
- `MyExe.refClass` does not reference any other user Java class.

To build the HFS Java executable, use the following `hpj` command:

```
hpj -make -o=MyExe MyExe.Main
```

Both classes are bound because the -follow option is the default. When the application is built for the first time, the -make option is not useful, but there is no harm in specifying it. The default is -exe and the Java executable MyExe is built.

If you subsequently modify `MyExe.refClass`, you can use the same `hpj` command to rebuild MyExe. This time, the -make option is very useful because it tells the bytecode binder to automatically regenerate any outdated ".o" files. If a class extends or uses the outdated class, it too will be rebuilt.

In this example, the bytecode binder regenerates both .o files because `MyExe.refClass` has been updated since the last build, and `MyExe.Main` references `MyExe.refClass`.

18.2 Scenario II

You want to build a Java DLL with many classes that are not necessarily related. The classes are as follows:

- `MyPackage.Read`
- `MyPackage.Write`
- `MyPackage.Utilities`

Specifying one class with the `hpj` command and using `-follow` will not work in this build scenario because these classes do not reference each other. You can, of course, specify all three classes on the command line, but a simpler `hpj` command can be used as follows:

```
hpj -jll -nofollow -o=MyPackage.jll
     -classpath=MyPackage.zip
          -include=MyPackage.* -make
```

You can use multiple -include and -exclude options to specify the classes that you want to bind without using a ZIP file, so long as the -classpath is correctly set to find the classes. In this example the three classes are zipped into MyPackage.zip. The `hpj` command builds MyPackage.jll with the three classes. Because you do not want referenced classes to be included in the DLL, the `-nofollow` option is required.

If subsequently you updated one or two of the classes and you want to rebuild MyPackage.jll with all the updated code, you can use the same `hpj` command. However, if you want to rebuild MyPackage and only update one of the classes, you should use the `-partial` option, as follows:

```
hpj -jll -o=MyPackage.jll -include=MyPackage.*
     -nofollow -classpath=MyPackage.zip
          -partial=MyPackage.Write
```

In this example, MyPackage.jll is rebuilt, and `MyPackage.Write` class is updated in the Java DLL. In rebuild, you simply replace the `-make` option in the initial build with the `-partial` option. You cannot specify both `-make` and `-partial` options in the same `hpj` command. You can specify the `-partial` option multiple times in a single `hpj` command. Note that `-partial` forces a bytecode rebinding of that class, and it is also more efficient than the `-make` option when you know ahead of time the classes that you want to rebind. You can also specify the wildcard character (*) in the `-partial` option.

18.3 Scenario III

You can specify ZIP or JAR input files in the `hpj` command. This is appropriate when you want to include all the classes in the ZIP or JAR file into the build. For example, app.jar contains two Java packages: `abc.staff.queryDB` and `abc.staff.verifyDB`. To build a Java DLL in the PDSE member MYPACK for all the classes in app.jar, use the `hpj` command as follows:

```
hpj app.jar -jll -nofollow
     -o="//'FRED.PDSE.LOAD(MYPACK)'"
          -alias=abc/staff/queryDB.jll
               -alias=abc/staff/verifyDB.jll
```

Alias names are required for Java DLLs written to PDSE members. One alias name is specified for each Java package that is in the Java DLL. All the classes in app.jar are included in the Java DLL. All resource files in the JAR file are ignored because the `-resource` option is not specified. The Java DLL is written to MYPACK in the FRED.PDSE.LOAD dataset.

If you only want to include a subset of the classes in the ZIP or JAR file, you should use the `-include` and `-exclude` options and specify the ZIP or JAR file, not as an input file, but as a file in the `-classpath` option, as follows:

```
hpj -jll -nofollow -o="//'FRED.PDSE.LOAD(MYDLL)'"
     -alias=abc/staff/queryDB.jll
          -alias=abc/staff/verifyDB.jll
               -classpath=app.zip:$CLASSPATH
                    -include=abc.staff.*
                         -exclude=abc.staff.verifyDB.r*
```

In this example, a subset of the classes in the ZIP file is used in the build. For example, if there is an `abc.staff.verifyDB.read` class in the ZIP file, it would not be included in the MYDLL PDSE member. An alias name is specified for each Java package.

18.4 Using Java DLLs

If you have created a Java DLL, you will want to be able to reference classes and methods that are within the DLL.

Here is a simple example that shows the procedure.

1. Create the DLL.

   A file named Adll.java contains the following:

   ```
   public class Adll {
       public void aMethod() {
           System.out.println( "This is Adll.aMethod" );
       }
   }
   ```

   a. Use `javac` to create a file named Adll.class:

   ```
   javac Adll.java
   ```

   b. Now, convert the bytecode to a DLL using `hpj`:

   ```
   hpj -jll -o=Adll.jll Adll
   ```

   This will create a file names Adll.jll.

Note: Do not delete the file named Adll.class.

2. Create the executable that will call method `aMethod` in class `Adll` in the DLL Adll.jll.

   A file named TestDll.java contains the following:

   ```
   public class TestDll {
       TestDll() {
           Adll ad = new Adll();    // this class is in the dll
           ad.out;
       }
        public static void main( String[] args ) {
           TestDll td = new TestDll();
       }
   }
   ```

   a. Use `javac` to create a file named TestDll.class:

      ```
      javac TestDll.java
      ```

   b. Now convert the bytecode to an executable using `hpj`:

      ```
      hpj -o=Test TestDll -nofollow
      ```

      The `-nofollow` option is used to prevent the `hpj` from binding Adll.class into the Test executable.

   To have successfully compiled these two files, you must have the current directory in your CLASSPATH. Now you can test that the TestDll executable runs and that it calls the class in the DLL just by typing:

   ```
   Test
   ```

   To prove that the DLL is located and loaded at runtime, you can rename or delete the DLL and rerun the executable.

Chapter 19. HPJ Performance on OS/390

The HPJ/390 compiler produces an executable file from Java bytecode. The main reason for using HPJ/390 is to produce an executable that runs faster than interpreted bytecode. But how much faster is the executable? Is it worthwhile creating a native compiled version of your application?

This section attempts to answer these questions.

Attention

It is important that the results not be considered benchmark figures. This is just one sample program that has been used in one specific environment. The results may not be the same in your own environment.

19.1 The Sample Code

```
public class Trial implements Runnable{
   int    anInteger = 0;
   double aDouble   = 0.0;

   int numberOfLoops = 1;
   int threadNumber  = 0;

   public Trial( int i, int t ) {
      super();

      numberOfLoops = i;
      threadNumber  = t;
   }

   public static void main(String args[]) {
      int numberOfLoops   = Integer.parseInt( args[1] );
      int numberOfThreads = Integer.parseInt( args[0] );

      long t0 = System.currentTimeMillis();
      for ( int i = 0; i < numberOfThreads; i++ )
         new Thread(new Trial( numberOfLoops, i ) ).start();

      // wait until all threads are completed
      while ( Thread.currentThread().activeCount() > 1 )
        ;

      long t1 = System.currentTimeMillis();
      System.out.println( "Time for all threads: "+(t1-t0));
   }

   public void run() {
      long t0,t1,t2,t3,t4,t5,t6;

      t0 = System.currentTimeMillis();
         for ( int i = 0; i < numberOfLoops; i++ )
            setADouble( i );
         t1 = System.currentTimeMillis();

         for ( int i = 0; i < numberOfLoops; i++ )
            setAnInteger( i );
         t2 = System.currentTimeMillis();

         for ( int i = 0; i < numberOfLoops; i++ )
            syncSetADouble( i );
         t3 = System.currentTimeMillis();

         for ( int i = 0; i < numberOfLoops; i++ )
            syncSetAnInteger( i );
         t4 = System.currentTimeMillis();

         for ( int i = 0; i < numberOfLoops; i++ )
            somethingMathematical( i );
         t5 = System.currentTimeMillis();

         float l = numberOfLoops / (float)1000;

         System.out.println(
           "elapsed for setDouble            : " + (t1-t0)/l );
         System.out.println(
           "elapsed for setInteger           : " + (t2-t1)/l );
         System.out.println(
           "elapsed for syncSetADouble       : " + (t3-t2)/l );
         System.out.println(
           "elapsed for syncSetAnInteger     : " + (t4-t3)/l );
         System.out.println(
           "elapsed for somethingMathematical: " + (t5-t4)/l);
         System.out.println(
            "elapsed time for thread " + threadNumber+": "+(t5-t0));
   }

   public  void setADouble( double d ) {
      aDouble = d;
   }

   public void setAnInteger(  int i ) {
      anInteger = i;
   }

   public void somethingMathematical( int i ) {
      double d =  (double)i;
      d = Math.sqrt( d );
      d = Math.tan( d );
   }

   public   synchronized void syncSetADouble( double d ) {
      aDouble = d;
   }

   public  synchronized void syncSetAnInteger(  int i ) {
      anInteger = i;
   }
}
```

The program takes two parameters on the command line:

1. The first parameter is the number of threads.
2. The second parameter is the number of times each method is called.

19.2 Summary Results

Rather than get into the controversial territory of absolute performance, we confine the results to relative performance.

For this sample, using a wide range of input values, it appears that the HPJ version consistently performed about 4.5 times faster than the bytecode equivalent. This held true even when the number of threads was significantly increased.

Chapter 20. Remote Debugger on OS/390

Part of the ET/390 component for VisualAge Java Enterprise Edition is the Remote Debugger.

20.1 Why to Use a Remote Debugger

If you are using the ET/390 component for VisualAge Java, you will be developing code within the VisualAge IDE on your workstation. You will use the Export and bind option of the ET/390 to transport Java files to the 390 and have them compiled there.

Without leaving the IDE you can use the ET/390 Remote Debugging tool to debug your Java code *as it executes on the 390*.

You can use all the features of a GUI debugger to step through your code, examine the stack, look at your program's variables and much more.

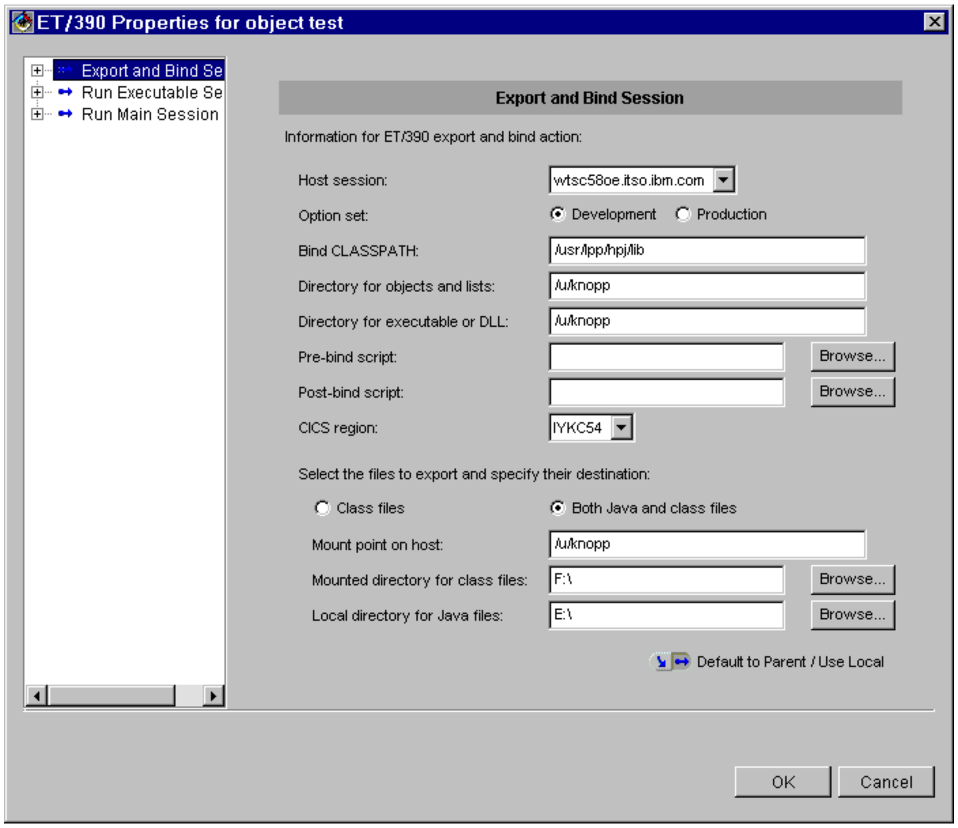
20.2 Getting Started with the Remote Debugger

Your OS/390 Systems Administrator will have to ensure that the appropriate runtime libraries are installed. You can read more about this in 5.2, “VisualAge for Java, Enterprise Edition for OS/390” on page 58.

You will also have to set some project properties to turn on debugging. Complete the following steps:

1. Right-click on your project.
2. Select **Tools** from the pop-up menu.
3. Select **ET/390** from the next pop-up menu.
4. Select **Properties** from the next pop-up menu.
5. A window will appear. The left side shows a graphical tree.
    a. Click **Export** and **Bind Session** to expand the branch.
    b. Click **Bind options**.
    c. Click the **Build a Java executable** in the main window.
    d. Click the **Data for debug and trace** checkbox in the main window.

The window as displayed in Figure 125 on page 282 illustrates this procedure.


ET/390 Properties for object test

Export and Bind Session

Information for ET/390 export and bind action:

Host session:	wtsc58oe.itso.ibm.com
Option set:	[✓ Development] Production
Bind CLASSPATH:	/usr/lpp/hpj/lib
Directory for objects and lists:	/u/knopp
Directory for executable or DLL:	/u/knopp
Pre-bind script:	
Post-bind script:	
CICS region:	IYKC54

Select the files to export and specify their destination:

Class files [✓ Both Java and class files]

Mount point on host:	/u/knopp
Mounted directory for class files:	F:\
Local directory for Java files:	E:\

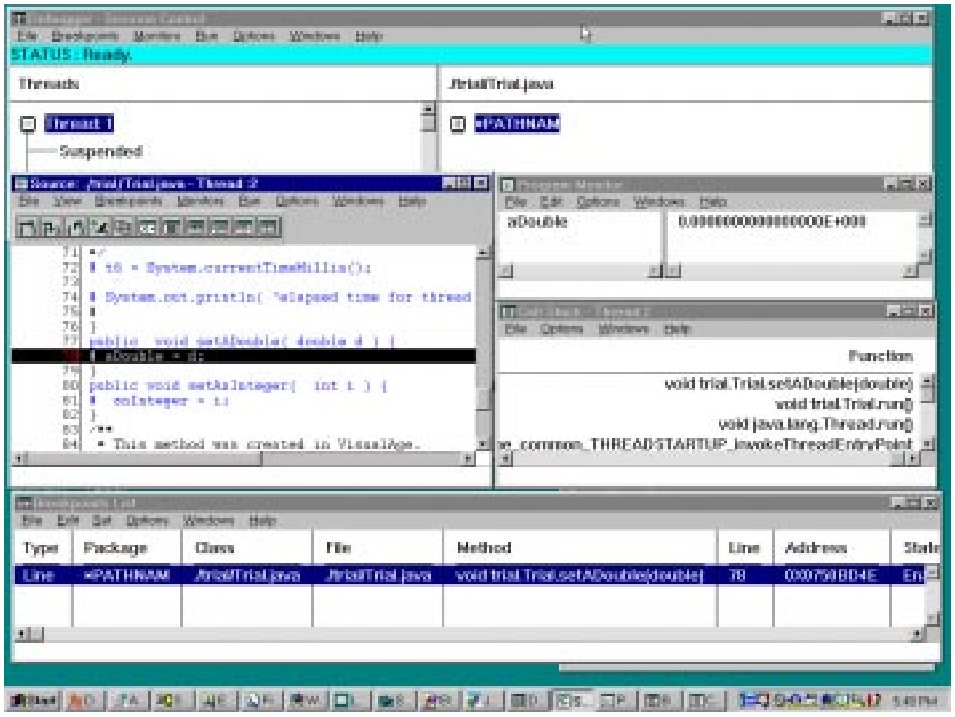
Figure 125. Turning on the Debug Option for ET/390

After you have successfully performed the Export and Bind phase of ET/390, you are ready for Remote debugging. Complete the following steps:

1. Right-click on your project.
2. Select **Tools** from the pop-up menu.
3. Select **ET/390** from the pop-up menu.
4. Select **Debug executable** from the pop-up menu.

After a short delay, this should start the Remote debugger.

Figure 126 on page 283 illustrates a sample of some possible debug activities.


File Breakpoints Monitors Run Options Windows Help
Threads
Thread 1
Suspended
/trial/Trial.java
*PATHNAM
Source: /trial/Trial.java - Thread 2
File View Breakpoints Monitors Run Options Windows Help
File Edit Options Windows Help
File Options Windows Help
Function
File Edit Set Options Windows Help

STATUS : Ready.

```
 */
 # t6 = System.currentTimeMillis();

 # System.out.println( "elapsed time for thread
 #
 }
 public  void setADouble( double d ) {
 # aDouble = d;
 }
 public void setAsInteger(  int i ) {
 #  onInteger = i;
 }
 /**
  * This method was created in VisualAge.
```

aDouble	0.0000000000000000E+000

```
                    void trial.Trial.setADouble(double)
                                  void trial.Trial.run()
                              void java.lang.Thread.run()
_common_THREADSTARTUP_invokeThreadEntryPoint
```

Type	Package	Class	File	Method	Line	Address	State
Line	*PATHNAM	/trial/Trial.java	/trial/Trial.java	void trial.Trial.setADouble(double)	78	0X0750BD4E	En

Figure 126. A Scenario Using the ET/390 Debugger

The is a very complex-looking screen. Not all these windows need to be present. But they all contribute something to your knowledge of what your program is doing. In 20.2.1, “Session Control Window” through 20.2.5, “Call Stack Window” on page 284, we briefly describe the purpose of each window.

20.2.1 Session Control Window

This window will always be present. It lets you choose a thread to debug.

20.2.2 Source Window

This window displays the current source. We have found it useful to confirm that the directory and filename displayed are the ones you expect.

The line that is currently being executed is highlighted. The icons on the toolbar give you the ability to carry out a wide variety of tasks. These include stepping into or over a method. You can set a breakpoint by clicking the line number where you want your program to break.

20.2.3 Breakpoints List Window

You can bring up this window by clicking the **Breakpoints** menu and selecting the **List All** option. This window is useful for helping you to keep track of where you have set breakpoints. It also allows you to delete a breakpoint that you no longer require.

20.2.4 Program Monitor Window

You can bring up this window by double-clicking on a variable in the source window. This window will show you the current value of the variable. It will also allow you to change that current value.

20.2.5 Call Stack Window

You can bring up this window by selecting the **Monitors** menu option and clicking on the **Display call stack** option. A call stack shows the path that the program took to get to the current line of code.

Chapter 21. Performance Analyzer for OS/390

Part of the ET/390 component for VisualAge Java Enterprise Edition is the Performance Analyzer for OS/390.

21.1 Why Use a Performance Analyzer

If you are using the ET/390 component for VisualAge Java, you will be developing code within the VisualAge IDE on your workstation. You will use the Export and bind option of the ET/390 to transport Java files to the 390 and have them compiled there.

Without leaving the IDE you can use the ET/390 Performance Analyzer to help you understand and improve the performance of your Java programs.

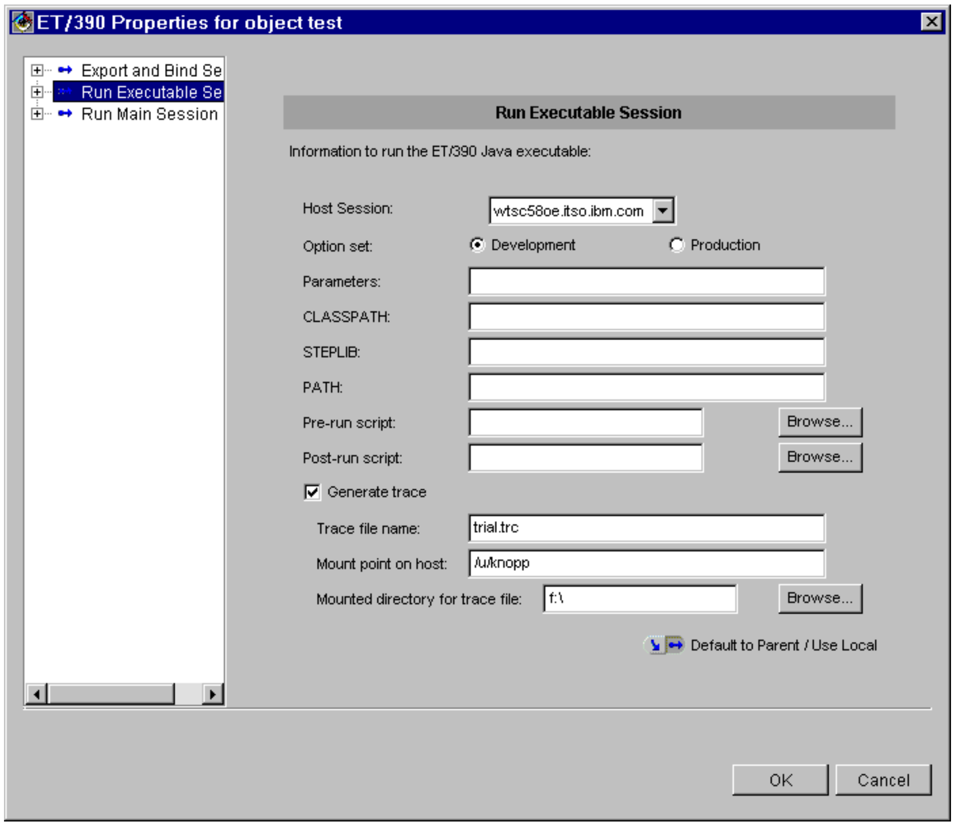
21.2 Getting Started with the Performance Analyzer

Your OS/390 Systems Administrator will have to ensure that the appropriate runtime libraries are installed. You can read more about this in 5.2, "VisualAge for Java, Enterprise Edition for OS/390" on page 58.

You will also have to set some project properties to turn on trace file logging. Complete the following steps:

1. Right-click on your project.
2. Select **Tools** from the pop-up menu.
3. Select **ET/390** from the next pop-up menu.
4. Select **Properties** from the next pop-up menu.
5. A window will appear. The left side shows a graphical tree.
    a. Click **Run Executable Session** to expand the branch.
    b. Click the **Generate trace** checkbox.
    c. Enter the "trace file name."
    d. Enter the "Mount point on host."
    e. Enter the "Mounted directory for trace file."

Figure 127 on page 286 shows the procedure.



Run Executable Session

Information to run the ET/390 Java executable:

Host Session: wtsc58oe.itso.ibm.com

Option set: [✓ Development] Production

Parameters:

CLASSPATH:

STEPLIB:

PATH:

Pre-run script:

Post-run script:

[✓] Generate trace

Trace file name: trial.trc

Mount point on host: /u/knopp

Mounted directory for trace file: f:\

Figure 127. Setting the Trace File Details

You should now perform the Export and Bind phase of ET/390. This is described in 20.2, “Getting Started with the Remote Debugger” on page 281.

Now you can run the application using ET/390. Complete the following steps:

1. Right-click on your project.
2. Select **Tools** from the pop-up menu.
3. Select **ET/390** from the next pop-up menu.
4. Select **Run executable** from the pop-up menu.

Running the application will have resulted in the creation of the trace file. The Performance Analyzer will use this trace file to calculate and display performance information.

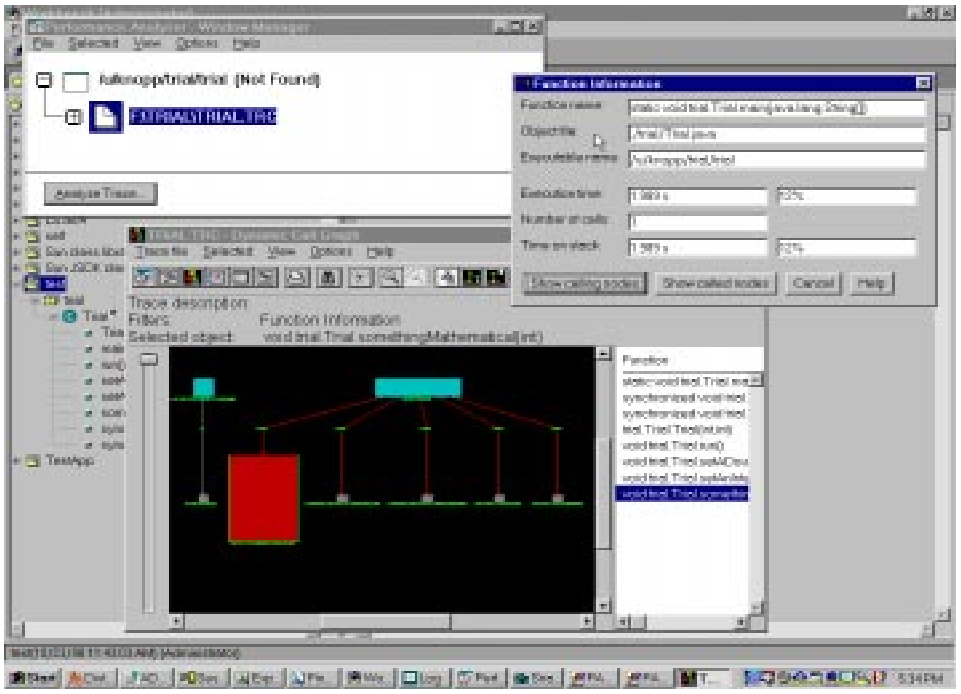
Now you are ready to run the Performance Analyzer. Complete the following steps:

1. Right-click on your project.
2. Select **Tools** from the pop-up menu.
3. Select **ET/390** from the next pop-up menu.
4. Select **Analyze Trace** from the next pop-up menu.

After a moment, the Performance Analyzer - Window Manager will appear. Click the **Analyze Trace** button. A window will appear asking you to enter the “trace file name.” You can use the browse option to find the file you specified in the Project Properties that you set earlier in this section.

There are quite a number of Function Analysis tasks that can be performed. Click the ones you want, then click **OK**. In a moment, you will see a window containing data for each task you selected.

Figure 128 shows an example of a Function Analysis. This is the Dynamic Call Graph.


Performance Analyzer - Window Manager
File Selected View Options Help
/u/lkrapp/trial/trial (Not Found)
F:\TRIAL\TRIAL.TRC
Analyze Trace...
Function Information
Show calling nodes
Show called nodes
Cancel
Help
Dynamic Call Graph
Trace file Selected View Options Help
Function

Function name: static void trial.Trial.main(java.lang.String[])
Object file: ./trial/Trial.java
Executable name: /u/lkrapp/trial/trial
Execution time: 1.589 s 92%
Number of calls: 1
Time on stack: 1.589 s 92%

Trace description
Filters: Function Information
Selected object: void trial.Trial.somethingMathematical(int)

Figure 128. The Dynamic Call Graph

You can find out more about the specifics of each of these analyses and more about the Performance Analyzer in general by using the search engine of the VisualAge for Java Integrated Development Environment.

To use the search engine, press **F1** from the VisualAge for Java IDE. Select **Information Search**. Enter “Introducing the Performance Analyzer (OS/390)” or any similar string to get more information about the Performance Analyzer.

Chapter 22. Mixing Java Bytecode and Objectcode on OS/390

At the time of writing, it was not possible to use native compiled classes or refer to methods in native compiled classes from a Java bytecode class.

The scenario where you would want to do this could be, for instance, in servlets. We found it a very useful architecture to have a "thin" servlet (as bytecode) and let it use native compiled classes.

Interpreting bytecode is not as efficient as running an executable (or DLL). So it would be convenient if only the core of the servlet was in bytecod e form. The Webserver would call this bytecode core. The bytecode would then call Java DLLs to do the major work. This would be a major performance advantage.
Currently, servlets only run as bytecode. However, it is not unlikely that WebSphere Application Server on OS/390 will also support native compiled servlets in the future.

There are various ways to get around this:

1. Consider a solution that does not use servlets.

   You may convert your servlet logic into another type of Java server-side program, like an RMI server, which will run as a native compiled program itself. In this case you could use other native compiled classes as well. However, you probably have good reasons for using servlets, so this workaround may not be a feasible option for you.

2. Use servlets and some remote object mechanism such as CORBA or RMI.

   Another workaround, and again not an ideal one in most cases, is to build your native compiled classes under a Java server program. The Java server program itself is also native compiled and is able to communicate with the servlet by means of sockets, RMI or CORBA. In that case, each request from the servlet to do the "heavy" work would result in a request to the Java server program. In this case, you would get communication overhead and you may loose some of the OO advantages if you run objects "de-coupled."

3. Convert your servlet logic into a CICS/Java transaction, which, in itself, is native compiled. Note, however, that, in this case you have to distill the HTML presentation from the servlet and implement an IIOP client.

Chapter 23. Using HPJ/390 with the Java Native Interface (JNI)

This chapter describes the steps required to use the JNI interface with HPJ on OS/390. It makes no attempt to teach you how to program using the JNI interface. We refer you to *Essential JNI: Java Native Interface*, by Rob Gordon for general information about JNI and the following URL for specific information regarding the usage of JNI on OS/390:

```
http://www.ibm.com/s390/java/jni_oe.html
```

23.1 JNI Step-By-Step

This example implements the “Hello World” program. HelloWorld has one method, a native method, that displays “Hello World,” and the implementation for the native method is provided in the C programming language.

The following steps can all be done using the OS/390 UNIX shell.

1. Write the Java Code

   Create a Java programming language class named HelloWorld that declares a native method and implements the main method:

```
class HelloWorld{
    public native void displayHello();
    static { System.load("LIBMYJNI"); }
    public static void main (String[] args) {
        new HelloWorld().displayHello();
    }
}
```

2. Convert the Java Source to Bytecode Use “javac” to produce the bytecode for the Java programming language code that you wrote in Step 1.

```
javac HelloWorld.java
```

3. Create the .h File

   Use “javah” to create a JNI-style header file (.h file) from the `HelloWorld` class. The header file provides a function prototype for the implementation of the native method `displayHelloWorld()`, which is defined in the `HelloWorld` class.

```
javah HelloWorld
```

4. Write the Native Method Implementation

   Write the implementation for the native method in a native language (such as ANSI C) source file. The implementation will be a regular function that is integrated with your Java programming language class.

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
```

```
JNIEXPORT void JNICALL Java_HelloWorld_displayHello(
                           JNIEnv *env,
                           jobject obj  )
{
  printf("\nJava_HelloWorld_displayHello gets control\n");
}
```

5. Create a Shared Library

   Use the C compiler to compile the .h file and the .c file that you created in Steps 3 and 4 into a shared library (DLL). The following makefile will automate this for you:

```
.SUFFIXES: .o .c

# Makefile to generate dll LIBMYJNI
# You MUST have OS/390 Optional Feature C/C++ installed
C_DLL       = LIBMYJNI
C_OBJS      = HelloWorld.o
C_CMD       = c89

JNI_LINK    =  $(IBMHPJ_HOME)/lib/HPJDLL.x
JNIINC      = -I $(IBMHPJ_HOME)/include -I /usr/lpp/java16/J1.1/include \
              -I /usr/lpp/java16/J1.1/include/mvs
DLL_CFLAGS = -W "c,expo,dll,noso,noexp,noshow,nolist,lang(extended)"
DLL_LFLAGS = -v -W "l,dll,dynam=DLL,map,msglevel=4"

.c.o:
   $(C_CMD) -c $(DLL_CFLAGS) -DNEEDSIEEE754 $(JNIINC) -I. $<

all: buildOE

buildOE: $(C_OBJS)
   $(C_CMD) $(DLL_LFLAGS) -o$(C_DLL) $(C_OBJS) $(JNI_LINK)> $(C_DLL).map 2>&1

clean:
   rm -f $(C_OBJS) $(C_DLL).map $(C_DLL).x $(C_DLL)
```

6. Compile the Java Bytecode Using HPJ/390

```
hpj HelloWorld -o=HelloWorld
```

   And finally, run the program.

```
HelloWorld
```

23.2 Other JNI Tips

You might consider using C++ rather than C as the native implementation, as this provides a cleaner object oriented approach. You can find out more about JNI using C++ and about JNI generally by using the search engine of the VisualAge for Java Integrated Development Environment.

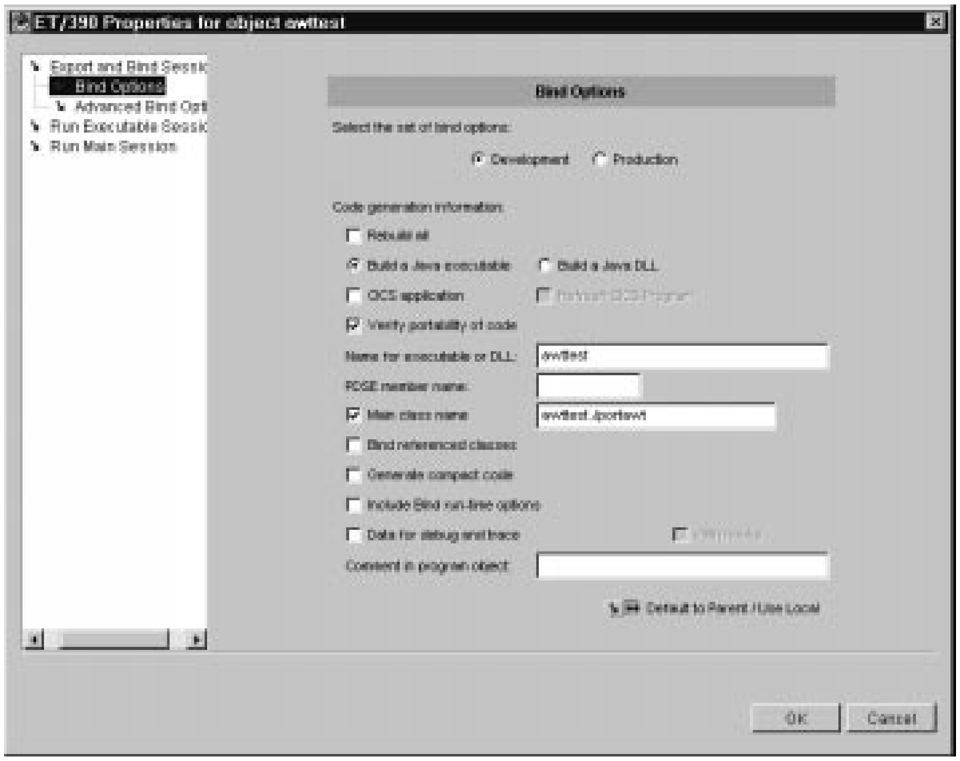
To use the search engine, press **F1** from the VisualAge for Java IDE. Select **Information Search**. Enter `OS/390 JNI` to get more information about the Java Native Interface for OS/390.

If you are using the Enterprise Edition of VisualAge for Java, you may want to explore the C++ Access Builder. This will automate many of the previous steps, leaving you to write just the C++ implementation and, of course, the Java calling code.

Once again, to get more information, use the VisualAge for Java IDE search engine. Search using “C++ Access Builder.”

Chapter 24. The Jport Utility

The Jport Utility tests packages or class files during the export and bind of a package to OS/390, if in the setup of the properties for the bind (under the export and bind properties) the checkbox for verify portability is checked. Figure 129 illustrates this.



ET/390 Properties for object awttest

- Export and Bind Sessic
  - Bind Options
  - Advanced Bind Opti
- Run Executable Sessic
- Run Main Session

Bind Options

Select the set of bind options:

[✓ Development] Production

Code generation information:

- [ ] Rebuild all
- [✓ Build a Java executable] Build a Java DLL
- [ ] CICS application
- [✓] Verify portability of code
- Name for executable or DLL: awttest
- PDSE member name:
- [✓] Main class name: awttest.Jportawt
- [ ] Bind referenced classes
- [ ] Generate compact code
- [ ] Include Bind run-time options
- [ ] Data for debug and trace
- Comment in program object:

Default to Parent / Use Local

OK Cancel

Figure 129. Verify Portability Option in Bind Options

The Jport Utility checks to insure that all referenced packages and classes are available on OS/390. One of the packages, for example, that is not provided on OS/390 is the awt (it does provide the remote awt).

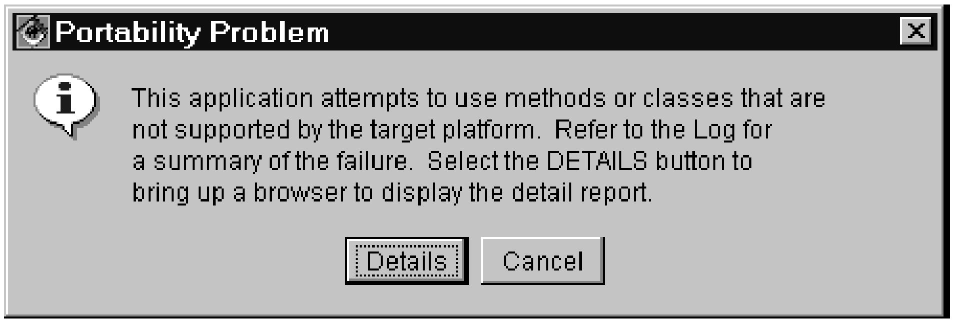
The jport utility was exercised by first doing an export and bind on a package that only used supported packages. No log messages were received and the export and bind proceeded to a normal conclusion. Then a package was put together that used the `java.awt` package. During the export and bind of this package, a message box appeared that indicated that an unsupported package was referenced and the jport test failed. The export of the .class file and the .java file continued to completion, but the bind was not performed. The log showed that the application was not portable.

Here we show the code that used the `java.awt` package:

```
package awttest;
import java.awt.*;
import java.awt.event.*;
/**
* This type was created in VisualAge.
*/
```

```
public class Jportawt {
Frame frame;
TextArea notes;
Jportawt()
{
frame = new Frame( " MyFrame " );
notes = new TextArea( );
frame.add( "Center", notes );
frame.setSize( 200, 300 );
frame.show( );
}
/**
* Starts the application.
* @param args an array of commandiline arguments
*/
public static void main(java.lang.String[] args) {
// Insert code to start the application here.
new Jportawt();
}
```

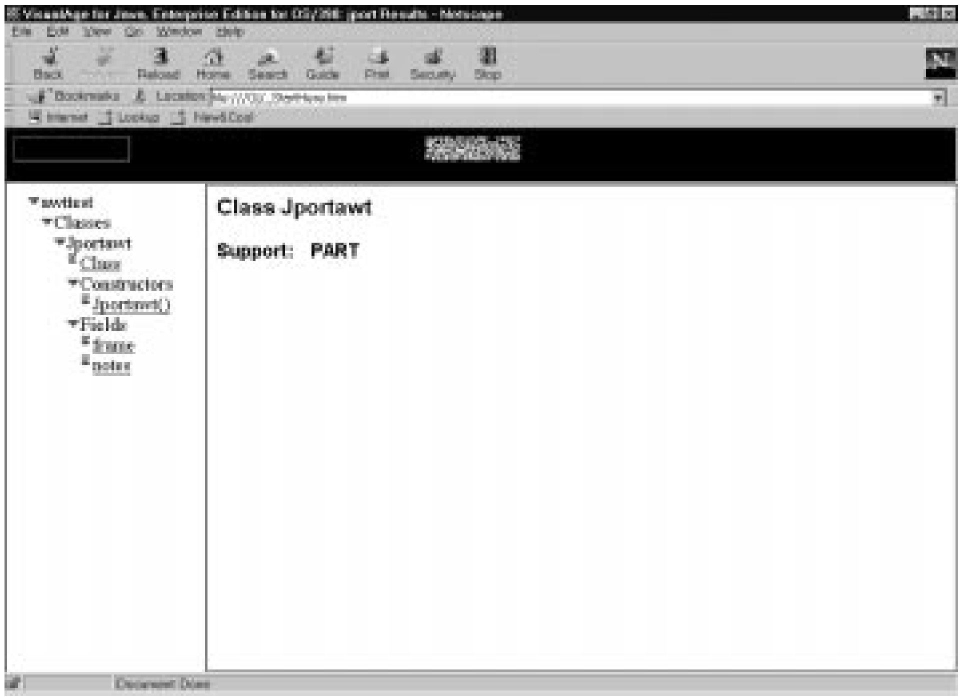
Figure 130 shows the problem report box that comes up when the package that used the `java.awt` package has an export and bind done on it.


Portability Problem

This application attempts to use methods or classes that are not supported by the target platform. Refer to the Log for a summary of the failure. Select the DETAILS button to bring up a browser to display the detail report.

Figure 130. Problem Report with Details Box

Figure 131 on page 297 shows the details of the unsupported packages and classes displayed on the browser.


VisualAge for Java, Enterprise Edition for OS/390: jport Results - Netscape
File Edit View Go Window Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location
Internet Lookup New&Cool
awttest
Classes
Jportawt
Class
Constructors
Jportawt()
Fields
frame
notes
Document: Done

Class Jportawt

Support: PART

Figure 131. Details of Unsupported Objects

Appendix A. CD-ROM

This appendix contains information regarding the contents of the enclosed CD-ROM. You may find it necessary to refer to the code on the disk to be able to understand 10.7, "A Closer Look at our Sample CICS Application" on page 198, 11.2, "Connecting to IMS Based on APPC" on page 216, and 11.3, "Access to IMS Using a Servlet/MQI" on page 235.

The CD-ROM has three parts:

- A directory structure containing sample code (sources) and documentation
- A "tar" file containing the samples (both sources and run versions) for installation on the OS/390 server
- A readme file in both PDF format and html format, containing the same text as this appendix

A.1 Directory Contents

The directory structure on the CD-ROM contains sample code, documentation and links to other resources. If you have a browser installed on your workstation, your system will automatically show the front page, `index.html`, after you have inserted the CD-ROM.

It is highly recommended to have a TCP/IP connection to the Internet, because some of the links point to URLs on the Internet.

A.2 tar File

The tar file on the CD contains basically the same components as the directory described in A.1, "Directory Contents"; however, the tar file is packaged in such a way that you can easily unpack it on your OS/390 system. If you succeed in installing the package on your OS/390 system, you can:

- Access the "content" on OS/390 from your browser
- Run the samples on the OS/390 system

A.3 Installing the Package on OS/390

In this section we describe the steps to follow to install the package on your OS/390 system. We will not describe the installation and configuration of the required subsystems, but we will refer to sources of information, if appropriate.

The following assumes some level of understanding of the UNIX System Services environment. It also assumes that you have a working knowledge of Java and the WebSphere Application Server running on OS/390.

Because selected demos use DB2 and CICS, you will need to have those subsystems customized as well. It is beyond the scope of this section to discuss how to install and configure DB2 and CICS, but we explain some important prerequisites to running the Java and JSP demos.

Note: For IMS and MQSeries we have only included source code examples to clarify the IMS and MQSeries chapters in the redbook. They are not meant to be installable demos or samples.

A.3.1 Prerequisite Software on OS/390

We used the following products in the development and testing of the samples.

Important

We mention the versions of the products that we used for the development and testing of our samples. However, you may be able to use them with higher versions of the products mentioned, but a newer version of a product may require another setup or may not even support the samples as we developed them.

- OS/390 Version 2 Release 5
- IBM WebSphere Application for OS/390 Version 1.1 (beta)
- Java for OS/390 (JDK Version 1.1.6)
- CICS Gateway for Java Version 2.0
- IBM DB2 Server for OS/390 Version 5.1
- IBM CICS Transaction Server for OS/390 Release 3 (LA)
- IBM IMS Transaction Server for OS/390 Version 6.1
- MQSeries for MVS/ESA V1.2
- MQSeries Bindings for Java for OS/390 (beta)
- DB2 for OS/390 Java Database Connectivity
- SQLJ support for DB2 Version 5 on OS/390

  Refer to the following URL for availability of SQLJ:

  `http://www.ibm.com/software/data/db2/os390/sqlj.html`

A.3.2 Installing the tar File on your OS/390 Server

The size of the tar file is approximately 28 MB. Perform the following steps to install the tar file on OS/390:

1. Logon to OS/390 and go into the `omvs` shell.
2. Create a directory to hold the tar file and its unpacked contents.

   We recommend that you create a subdirectory inside `/u`, like SG245342. Also, you may find it useful to create a separate HFS data set for the sample package. The following JCL can be used to do this:

```
//HFSALLOC JOB (POK,999),HFSALL,MSGLEVEL=(1,1),MSGCLASS=X,
//  CLASS=A,NOTIFY=userid
//*
//STEP01  EXEC PGM=IEFBR14
//HFS       DD DSN=OMVS.SG245342.HFS,SPACE=(CYL,(160,5,1)),  1
//             DSNTYPE=HFS,DCB=DSORG=PO,
//             DISP=(NEW,CATLG,DELETE),
//             STORCLAS=OPENMVS,VOL=SER=volser,UNIT=unit
```

   Notes:

1 Depending on your type of DASD, allocate sufficient space to be able to hold about 30 MB.

Once you have created the HFS data set, you can mount it with the following command from TSO:

```
MOUNT FILESYSTEM(OMVS.SG245342.HFS) MOUNTPOINT(/u/SG245342) TYPE(HFS)
```

3. Open an ftp connection between your workstation and your OS/390 server.
4. Go to the directory you just created.
5. Transfer the file in binary format into the preferred subdirectory.
6. Unpack the tar file with the command:

   ```
   tar -xvf SG245342.tar
   ```

   This command will lead to the creation of all the subdirectories and files, as follows:

   ```
   drwxr-xr-x   3 ALEX     TSO            0 Apr 23 15:12 config
   drwxr-xr-x  13 ALEX     TSO            0 Feb 16 16:36 html
   drwxr-xr-x   5 ALEX     TSO            0 Apr 23 15:27 imsmq
   drwxr-xr-x   3 ALEX     TSO            0 Feb 19 09:26 java
   drwxr-xr-x   2 ALEX     TSO            0 Apr 23 14:28 mvs
   ```

7. Once the unpacking is done, you may want to change the permission bits and the owner of the directories and files in the package. Useful commands are:
   - `chown <userid> <name>` for changing the ownership
   - `chmod xxx <name>` for changing the permission bits

The unpacking of the tar file will result in five other subdirectories to be created:

config	contains sample configuration files
html	contains the Web content
imsmq	contains sample sources on how to use IMS and MQ
java	contains the java source and class files that make up the samples
mvs	contains MVS components needed to set up and use the back-end systems.

A.3.3 Moving the MVS Components to Data Sets

The `mvs` directory contains the components that you will need to set up the MVS side of some samples. Most of them are stored in a special format called “TSO TRANSMIT OUTDATA.”

You will find the following files in the `mvs` subdirectory:

```
-rwxr-xr-x   1 ALEX     TSO         6480 Apr 26 15:10 cobol.copybook.xmit
-rwxr-xr-x   1 ALEX     TSO       112080 Apr 26 15:10 cobol.source.xmit
-rwxr-xr-x   1 ALEX     TSO       690400 Apr 26 15:10 db2.data.xmit
-rwxr-xr-x   1 ALEX     TSO        67500 Apr 26 15:12 db2.unload
-rwxr-xr-x   1 ALEX     TSO        13840 Apr 26 15:11 dbrmlib.xmit
-rwxr-xr-x   1 ALEX     TSO         2240 Apr 26 15:12 dsnaoini
-rwxr-xr-x   1 ALEX     TSO        50800 Apr 26 15:09 jcl.xmit
-rwxr-xr-x   1 ALEX     TSO        30240 Apr 26 15:14 load.xmit
```

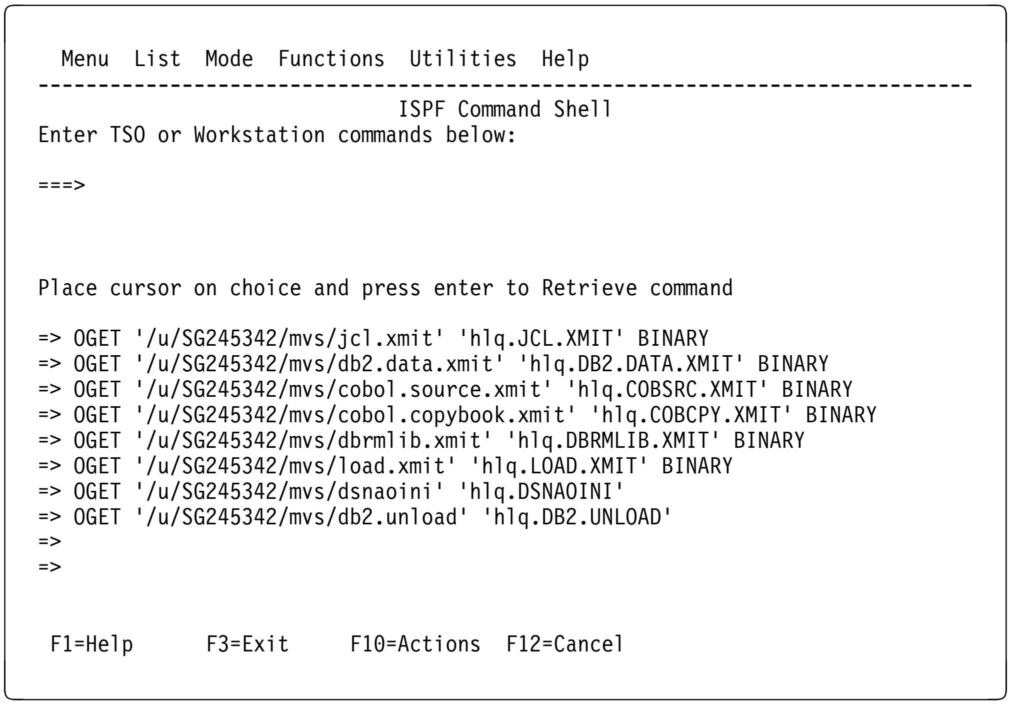
In order to work with these files, they must be copied to traditional MVS data sets. You can use the TSO OGET command to pull these files out of the HFS and put them in data sets.

You can use the following JCL to pre-allocate the data sets on OS/390.

Note: Pre-allocation is necessary to give the data sets the correct LRECL, RECFM and BLKSIZE. You must use the LRECL and RECFM as indicated in the following JCL.

```
//ALLOC  JOB (POK,999),MSGCLASS=T,NOTIFY=USERID
//*
//* Allocate the demo data sets
//*
//NCXALLO EXEC PGM=IEFBR14
//JCL     DD DISP=(NEW,CATLG),SPACE=(TRK,(10,5)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.JCL.XMIT
//DATA    DD DISP=(NEW,CATLG),SPACE=(TRK,(5,2)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.DB2.DATA.XMIT
//COBSRC  DD DISP=(NEW,CATLG),SPACE=(TRK,(10,5)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.COBSRC.XMIT
//COBCPY  DD DISP=(NEW,CATLG),SPACE=(TRK,(10,5)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.COBCPY.XMIT
//DBRMPY  DD DISP=(NEW,CATLG),SPACE=(TRK,(10,5)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.DBRMLIB.XMIT
//LOADPY  DD DISP=(NEW,CATLG),SPACE=(TRK,(20,10)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.LOAD.XMIT
//DSNAOI  DD DISP=(NEW,CATLG),SPACE=(TRK,(5,5)),UNIT=SYSDA,
//         DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DSN=hlq.DSNAOINI
//DB2UNL  DD DISP=(NEW,CATLG),SPACE=(TRK,(10,5)),UNIT=SYSDA,
//         DCB=(LRECL=4092,RECFM=VB,BLKSIZE=4096),DSN=hlq.DB2.UNLOAD
```

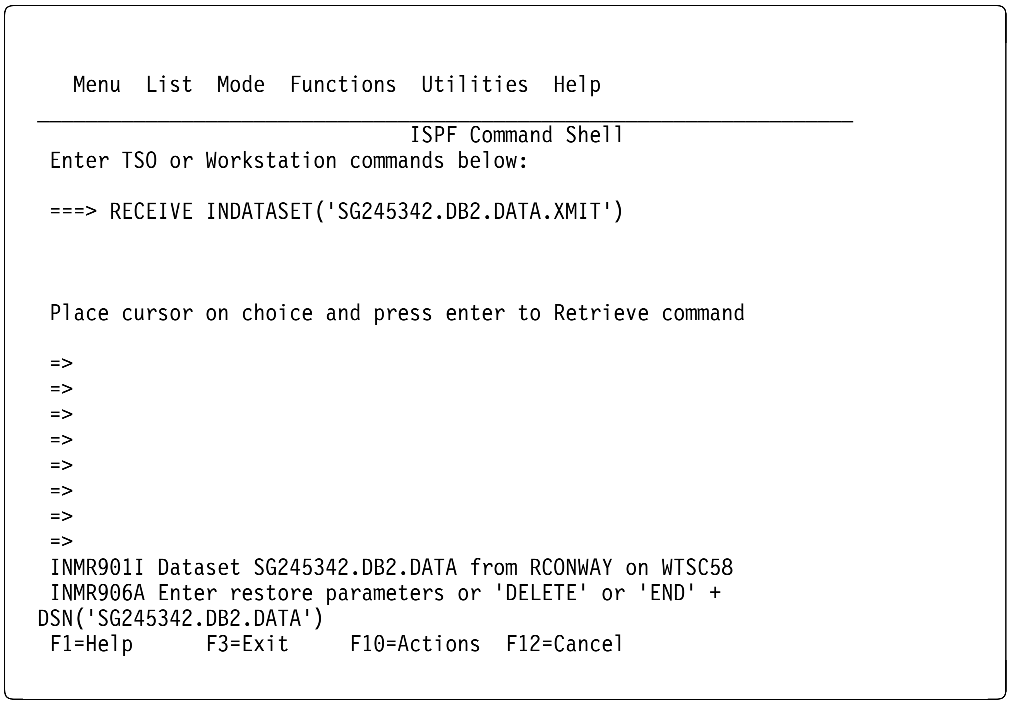
After you have allocated the data sets, use the TSO OGET command to copy the files from the HFS to MVS data sets. Following is a sample of the commands.



```
  Menu  List  Mode  Functions  Utilities  Help
-------------------------------------------------------------------------------
                              ISPF Command Shell
Enter TSO or Workstation commands below:

===>



Place cursor on choice and press enter to Retrieve command

=> OGET '/u/SG245342/mvs/jcl.xmit' 'hlq.JCL.XMIT' BINARY
=> OGET '/u/SG245342/mvs/db2.data.xmit' 'hlq.DB2.DATA.XMIT' BINARY
=> OGET '/u/SG245342/mvs/cobol.source.xmit' 'hlq.COBSRC.XMIT' BINARY
=> OGET '/u/SG245342/mvs/cobol.copybook.xmit' 'hlq.COBCPY.XMIT' BINARY
=> OGET '/u/SG245342/mvs/dbrmlib.xmit' 'hlq.DBRMLIB.XMIT' BINARY
=> OGET '/u/SG245342/mvs/load.xmit' 'hlq.LOAD.XMIT' BINARY
=> OGET '/u/SG245342/mvs/dsnaoini' 'hlq.DSNAOINI'
=> OGET '/u/SG245342/mvs/db2.unload' 'hlq.DB2.UNLOAD'
=>
=>


 F1=Help      F3=Exit     F10=Actions  F12=Cancel
```

Note: Notice the binary option on the OGET command in case of “xmit” files. The dsnaoini and db2.unload files must not be copied in binary mode.

Use the TSO RECEIVE command to unpack the XMIT data sets. These data sets were compressed using TSO TRANSMIT OUTDATASET and must be received with TSO RECEIVE INDATASET.
Enter the command `TSO RECEIVE INDATASET('<data set name>')` for example, `TSO RECEIVE INDATASET('SG245342.DB2.DATA.XMIT')`. TSO will now prompt you for the data set name to store the data set being received. If the hlq you are using for your installation is the same as your TSO PREFIX, then just press Enter to use the default values for receiving the data set.
If you are using a different hlq for the installation of the sample code, enter `DSN('hlq.DB2.DATA')`. A sample of the command is shown here.



```
  Menu  List  Mode  Functions  Utilities  Help
 ___________________________________________________________________
                            ISPF Command Shell
 Enter TSO or Workstation commands below:

 ===> RECEIVE INDATASET('SG245342.DB2.DATA.XMIT')



 Place cursor on choice and press enter to Retrieve command

 =>
 =>
 =>
 =>
 =>
 =>
 =>
 =>
 INMR901I Dataset SG245342.DB2.DATA from RCONWAY on WTSC58
 INMR906A Enter restore parameters or 'DELETE' or 'END' +
DSN('SG245342.DB2.DATA')
 F1=Help      F3=Exit     F10=Actions  F12=Cancel
```

Note: Repeat this command for every xmit data set. The dsnaoini and db2.unload files must not be received, as they are not in TSO TRANSMIT OUTDATA format.

A.3.4 Configuring the Environment

In this section we describe the steps to perform in order to be able to use the samples in your WebSphere Application Server. It is assumed that your HTTP Server and the WebSphere Application Server supporting servlets are up and running on your system. However, we give you some key additions that must be made to your configuration files.

A.3.4.1 Configuring the WebSphere Application Server

In order to run the samples, you need to make some updates in the httpd.conf, httpd.envvars and jvm.properties files and the started procedure.

1. Updates to your httpd.conf file:

   `/u/SG245342/config/webas/httpd.conf` is a sample httpd.conf file that we used for the development and testing of the samples. It contains additional PASS statements that must be added to your Webserver's httpd.conf file.

2. Updates to your httpd.envvars file:

   `/u/SG245342/config/webas/httpd.envvars` is a sample httpd.envvars file and contains paths that need to be added to your Webserver's httpd.envvars file.

3. Updates to your jvm.properties file:

   `/u/SG245342/config/webas/jvm.properties` is a sample jvm.properties file and contains paths that need to be added to your Webserver's jvm.properties file.

   The minimum to be included on top of your default WebAS class libraries in your ncf.jvm.classpath directive are the following class libraries:

   - `/usr/lpp/java16/J1.1/lib/classes.zip`

     This contains the JDK 1.1.6 classes.

- `/usr/lpp/db2/db2510/classes/db2jdbcclasses.zip`

  This is required for JDBC access to DB2 on OS/390.

- `/usr/lpp/db2/db2510/classes/db2sqljclasses.zip`

  This is required for SQLJ access to DB2 on OS/390.

- `/u/SG245342/java`

  This directory contains the Java sample classes.

  Note: In case you decided to use another subdirectory for the sample package, you need to use that name.

- `/usr/lpp/JRIO/recordio-vsam.zip`, `/usr/lpp/JRIO/recordio.zip` and `/usr/lpp/JRIO/recordio-nonvsam.zip`

  In these directories, you must have installed the JRIO classes downloadable from URL:

  ```
  http://www.ibm.com/s390/java
  ```

- `/usr/lpp/jgCICS/JGate/classes`

  This directory contains the CICS Gateway for Java classes.

Note: The names of the directories mentioned above may be different on your system.

The minimum to be included on top of your default WebAS executables libraries in your ncf.jvm.libpath directive are the following:

- `/usr/lpp/java16/J1.1/lib` and `/usr/lpp/java16/J1.1/lib/mvs/native_threads`

  These directories cntain the JDK DLLs.

- `/usr/lpp/db2/db2510/lib`

  This directory contains the native drivers for JDBC.

- `/usr/lpp/jgCICS/JGate/bin/mvs`

  This directory contains the DLLs of the CICS Gateway for Java.

- `/usr/lpp/JRIO`

  This directory contains the DLLs for MVS dataset access from Java.

The ncf.jvm.path directive should contain the DLLs of the JDK 1.1.6 as well.

A.3.4.2 SG245342.ini File

In `/u/SG245342/config` there is a file called "SG245342.ini." This file is critical for the operation of the samples. All Java servlets read this file in order to retrieve environment variables. This file needs to be in the root directory of your Webserver. This directory is normally the same directory as where you have the httpd.conf file.

Important

You need to edit the SG245342.ini file to reflect your environment.

A.3.4.3 Configuring DB2

It is beyond the scope of this section to discuss how to install and configure DB2 on your system, but there are some key prerequisites to running the Java and JSP demos.

1. DB2 prerequisite customization.

   a. Install and customize DB2 CLI support. See *DB2 for OS/390 Call Level Interface Guide and Reference*, SC26-8959 for instructions.

   b. Install and customize DB2 JDBC support. See the JDBC README file for more information.

   c. Install and customize DB2 SQLJ support. See the SQLJ README file for more information.

2. Load the DB2 database with sample data.

   Run job DB2LOAD in your unpacked JCL data set. Before running, modify the JCL to conform to your local standards.

3. Rebind the SQLJ samples and the CICS/COBOL sample.

   Jobs BINDAPP and BINMANC in your unpackd JCL data set are provided for this purpose.

4. Copy the sample DSNAOCLI file from `/u/SG245342/mvs/dsnaoini` to an MVS sequential data set, as follows:

```
   Menu  List  Mode  Functions  Utilities  Help
------------------------------------------------------------------------
                              ISPF Command Shell
Enter TSO or Workstation commands below:

 ===> OGET '/u/SG245342/mvs/dsnaoini' 'hlq.DSNAOINI'
```

5. Customize this data set by filling in your DB2 subsystem identifier and CLI planname. Add this data set to your Webserver's started procedure with the following DD name:

```
//DSNAOINI DD DSN=hlq.DSNAOINI,DISP=SHR
```

6. Allocate DB2 libraries to the Webserver's procedure. Add a STEPLIB or JOBLIB to the Webserver's procedure with the following data sets, or add them to the system link list concatenation:

```
hlq.SDSNEXIT
hlq.SDSNLOAD
```

A.3.4.4 Configuring CICS

A few of the supplied samples in this package require CICS and the CICS Java Gateway to be customized. It is beyond the scope of this section to discuss how to install and configure CICS and the CICS Java Gateway on your system, but there are some key prerequisites to running the demos with CICS.

1. Install and initialize the External CICS Interface (EXCI).

   The CICS Java Gateway requires that the External CICS Interface (EXCI) be customized. EXCI is available with CICS/ESA version 4.1 and all subsequent versions of CICS/ESA. The use of the EXCI interface is documented in *CICS TS for OS/390 V1R2 CICS Internet and External Interfaces Guide*, SC33-1944.

This publication describes the operation of the EXCI sample programs, as well as the installation steps required to use the interface. EXCI must be installed and initialized before you can use the samples.

2. Add Language Environment/370 support for CICS.

   The COBOL sample programs have been linked with the COBOL runtime library stubs that are shipped with Language Environment (LE). See *CICS Transaction Server for OS/390 V1R2 CICS System Definition Guide*, SC33-1682 for complete information on how to install Language Environment/370 support for CICS.

3. Add the CICS sample application data set to your CICS procedure. The data set in which you unpacked the LOADS can be added to the DFHRPL concatenation as follows:

   ```
   //STEPLIB  DD DSN=CICSTS12.SDFHAUTH,DISP=SHR
   //         DD DSN=SYS1.CSSLIB,DISP=SHR
   //         DD DSN=CEE.SCEECICS,DISP=SHR
   //DFHRPL   DD DSN=CICSTS12.SDFHLOAD,DISP=SHR
   //         DD DSN=SYS1.CSSLIB,DISP=SHR
   //         DD DSN=CEE.SCEERUN,DISP=SHR
   //         DD DSN=CEE.SCEECICS,DISP=SHR
   //         DD DSN=hlq.LOAD,DISP=SHR
   ```

4. Data set hlq.JCL contains an assembler module, called “DFHCNV.” This module is responsible for codepage conversion of the passed CICS COMMAREA. Each program called from “outside” needs to have an entry in DFHCNV. The example in hlq.JCL needs to be linked. You need to restart your CICS region or just refresh this particular program in order to activate your new version.

5. Add PROGRAM and TRANSACTION definitions.

   In order to use the CICS application programs that are supplied in hlq.LOAD, you have to define them to CICS. Run job hlq.JCL(ADDDEFS) to add PROGRAM and TRANSACTION definitions for the supplied CICS application programs.

6. Add a STEPLIB or JOBLIB to the WebSphere Application Server for OS/390 procedure with the following data set, or add it to the system link list concatenation:

   ```
   hlq.SDFHEXCI
   ```

7. Configure the CICS Attachment Facility.

   The sample CICS transactions that are supplied on the CD-ROM access data in a DB2 database. The CICS Attachment facility needs to be configured. See *DB2 for OS/390 V5 Installation Guide*, GC26-8970 for more information on how to configure this attachment.

8. Reinitialize the WebSphere Application Server for OS/390.

   You will need to stop and start WebAS to make the changes effective.

A.3.4.5 Configuring the VSAM Sample

In order to make the VSAM sample to work, you have to define a VSAM cluster and load it with data. Job DEFVSAM in data set hlq.JCL takes care of:

1. Deleting an already VSAM cluster with the same name
2. Defining a new cluster
3. Copying a flat file with input data into the VSAM file

   Note: The data provided is an unload of the DB2 manufacturer table.

4. Printing the contents of the file

A.3.5 MVS Datasets on the CD-ROM

Once you have unpacked the MVS datasets, you will find the following helpful datasets:

hlq.JCL This dataset contains JCL for the samples. All jobs need to be modified before they can run. The instructions are in the JCL. The following jobs are provided:

ADDDEFS	This job adds the definitions for the COBOL/CICS sample transactions in CICS.
BINDAPP	This job binds an application that uses static SQL to access DB2. Running this job is mandatory to be able to run an SQLJ program.
BINDMANC	Running this job is mandatory to be able to run the CICS samples on this CD-ROM.
DB2LOAD	This job defines the DB2 table and loads its contents for the samples.
DEFVSAM	This job defines a VSAM cluster and copies a flat file with records into it.
DFHCNV	This is a sample job to assemble the DFHCNV for CICS.
EXECEXCI	This job can be run to test the EXCI connection to your CICS system.
MANUFA*	These jobs are required to pre-compile, compile, link-edit and bind the COBOL/CICS transactions.
TIMEZONE	This job compiles the COBOL source of the TIMEZONE program.

hlq.LOAD	This library contains the CICS LOADS for the samples.
hlq.DB2.DATA	This dataset contains a member with the DDL for the DB2 sample database.
hlq.DBRMLIB	This library contains DBRMs of the sample programs that use static SQL.
hlq.COBSRC	This library contains the source of the COBOL programs used in the CICS samples.
hlq.COBCPY	This library contains the source of the COBOL copybooks used in the CICS transactions.

hlq.DSNAOINI	This data set contains the DB2 CLI initialization file.
hlq.DB2.UNLOAD	This data set contains “flat” output from the DB2 manufacturer table. This file ought to be used to load the VSAM file.

A.3.6 Using the Samples

Once you have installed the samples, as described in section A.3.4, “Configuring the Environment” on page 304, and you have all the prerequisite subsystems running, you will be able to point to the html frontpage of the sample package with your browser, as follows:

```
<tcp/ip address>:<portnumber>
```

Important

You have to make sure that in your `httpd.conf` file, you point to the `index.html` file in the html directory of the sample package.

Appendix B. Design of the Advanced Sample Application

This appendix describes the layout of the database we use in our example, the design considerations for the sample application, the class model, and the relationship between the various classes.

B.1 The Design of the Database

Our sample program deals with a MANUFACTURER table we stored in a MANFACT database. This is just a single table without any further or related tables. We want to keep it as simple as possible.

The manufacturer table defines some of the major attributes of a manufacturer. The following list shows you the defined columns.

MAN_NAME	The name and unique key
MAN_ADDRESS	All the address information we need
MAN_CITY	
MAN_STATE	
MAN_ZIP	
MAN_CON_LAST_NAME	The last name of the contact
MAN_CON_FIRST_NAME	The contact's first name
MAN_CON_PHONE_AC	The necessary phone information
MAN_CON_PHONE_EX	
MAN_CON_PHONE_NR	
MAN_CON_EXT	
MAN_CON_EMAIL	The contact's e-mail address
MAN_LAST_UPD_UID	The program identification (we use JAVAID)
MAN_LAST_UPD_DATE	Always the current timestamp

B.2 The Model View Controller Architecture

Since Java is an OO-programming language, and because we want to use the same sample for all the different connectors on OS/390 described in this book, we spend more time on the design of the class model than merely doing straightforward coding.

Our design follows the Model View Controller architecture introduced by the famous *Gang Of Four* in the mid-nineties. This design pattern defines a strong separation between the user interface (UI), considered the “view,” and the “model,” which represents the implementation of the data and the business logic.

The view and the model are connected through a “controller.” In OO theory, the term *messages* means the communication between different classes. For that reason, the controller-class does nothing except pass messages from the view to the model, and vice versa.

For further information on the model view controller design pattern and related topics, refer to:

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch, Addison-Wesley; ISBN: 0201633612.

B.3 The Class Design of the Sample Program

In this section we provide an overview of the main classes we use in our sample program.

B.3.1 The Manufacturer Business Object Class

We design a strong separation between the view and the model simply by having classes that implement the business logic and other classes that represent the data

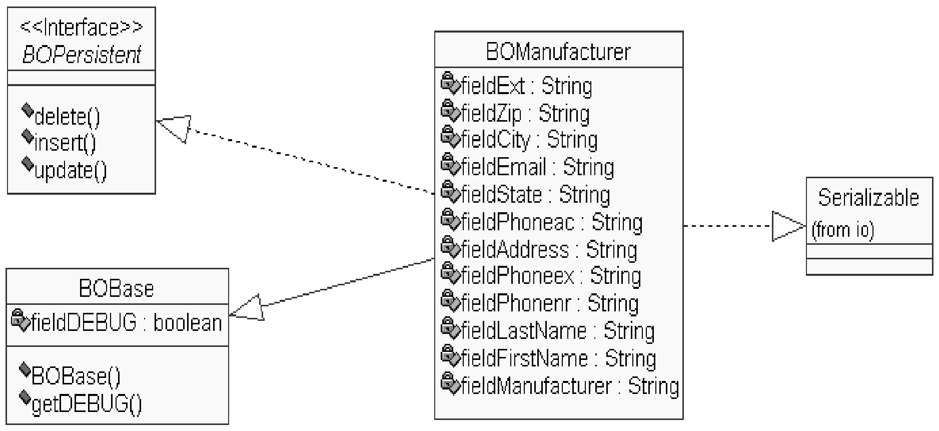


Figure 132. The Manufacturer Class Model

With the exception of `MAN_LAST_UPD_UID` and `MAN_LAST_UPD_DATE`, the `BOManufacturer` class is the exact representation of the table scheme: it represents one row of the database, with one attribute for each column. Because the manufacturer is considered to be a business object, we add the prefix `BO`.

Note that `MAN_LAST_UPD_UID` and `MAN_LAST_UPD_DATE` provide system information. They have no relationship to a manufacturer object itself, and are therefore handled by the business logic.

All properties are defined as private class members and can be accessed only through the `setter` and `getter` methods.

The Java class is compatible with the Java Beans component model; visual composition in the sense of the Visual Age Java Visual Composition Editor is possible.

We want the manufacturer data to be storable in a non-transient location. Therefore, it implements the interfaces `BOPseristent` and `java.io.Serializable`. It is derived from `BOBase`.

Implementing these interfaces ensures the Java runtime environment that the methods that are declared in the interface can be called from any other class. In other words, the object ensures that it can retrieve messages through these methods.

- `BOPseristent` declares these methods:
  - Delete
  - Insert
  - Update

The manufacturer class must provide the declared methods. Derived classes for JDBC, CICS or SQLJ implement their own behavior for these methods by simply implementing a method with the same name but a different function body. Depending on the type of object, the Java runtime environment automatically calls the correct method.

Attention

Note that `java.io.Serializable` declares no methods.

However, it guarantees that the class that implements the interface can write and read itself to and from an output stream. The implementation of RMI makes it necessary that all objects that are going to be passed between different machines or processes implement this interface.

`BOBase` has no meaning in the current system and can be extended to whatever use.

Furthermore, we wanted all the different connectors for CICS, JDBC and SQLJ implement a derived class for `BOManufacturer`.

With this approach, the `BOManufacturer` class, together with `BOPseristent` class, ensures that the representation to the business logic or the application program is always the same, no matter what connector type is being used.

B.3.2 The Datamanager Class

The `DataManager` class is the abstract representation of a connector that the application wants to talk to. However, there are connectors available. In this example we can make use of one of the following connectors:

- CICS Java Gateway
- JDBC or JDBC/ODBC drivers
- SQLJ
- IMS Java Gateway
- MQ Series
- MVS datasets

The `DataManager` class gives the interface for the application. At this time, most of the methods cannot provide a way of communicating with a specific connector. Therefore, most of the methods and the class itself are declared as `abstract`. The concrete implementation must be done in the derived classes, depending on the connector it is designed for; see Figure 133 on page 314.

ager class.

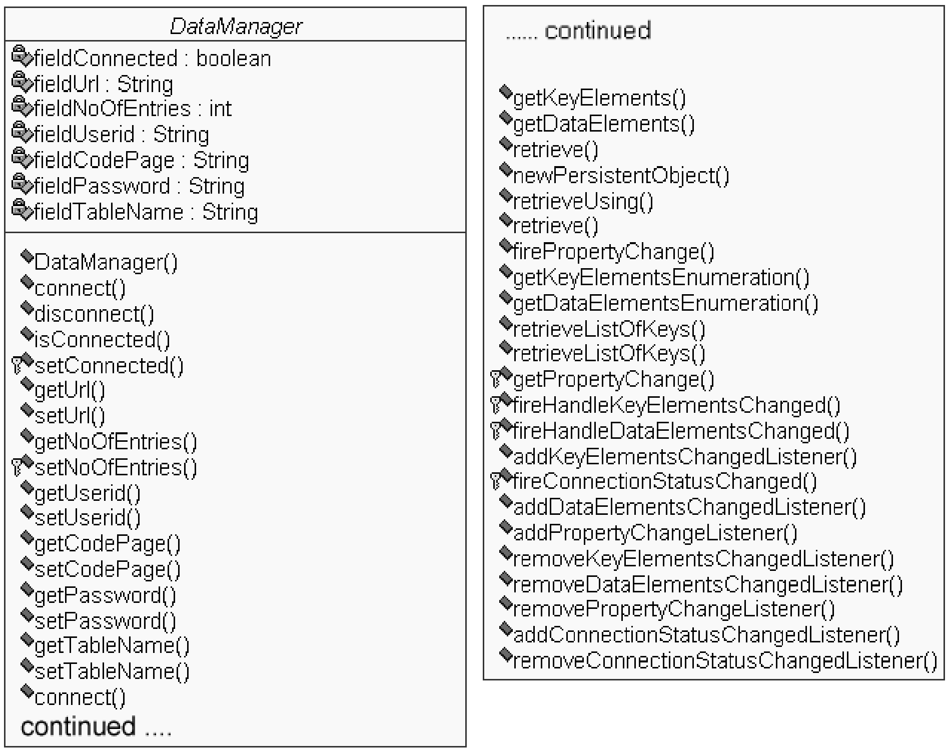


Figure 133. The DataManager Class - Attributes and Methods

Since the `DataManager` class is abstract, it cannot be instantiated. Concrete implementations can be made by simply inheriting `DataManager`, as Figure 134 illustrates.

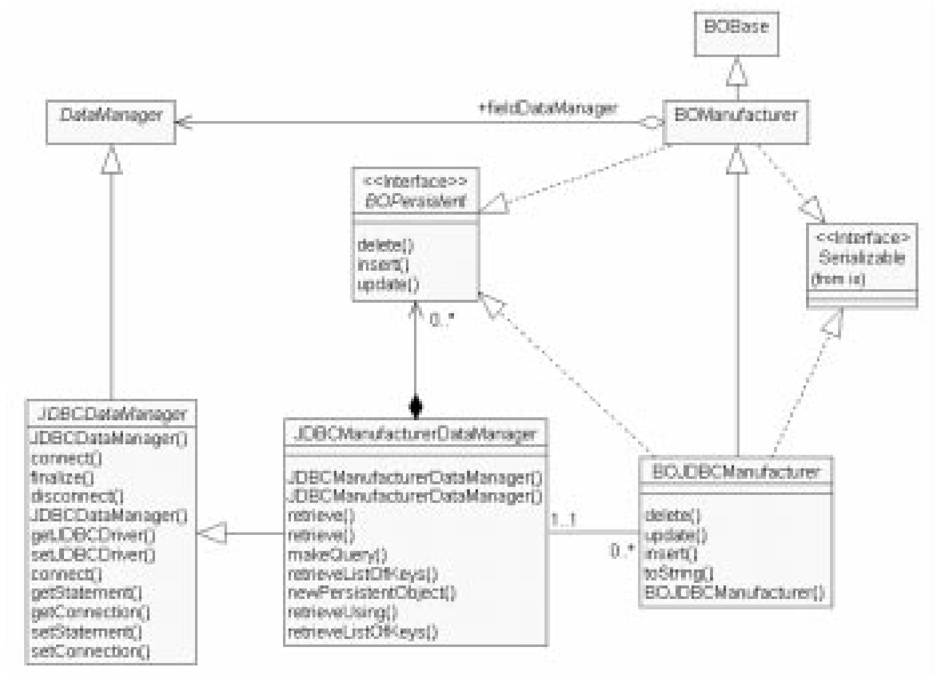


Figure 134. The Data Manager Inheritance

Figure 134 shows a concrete implementation for the JDBC connector. The class `JDBCDataManager` is specialized for JDBC access; and `JDBCManufacturerDataManager` is the concrete JDBC implementation for the manufacturer database. Moreover, the data manager can deal with a large unlimited number of rows in the database. It also has the capability of mapping table-rows into manufacturer objects.

There is another class of interest: `BOJDBCManufacturer`. This class implements the concrete JDBC-code for the methods declared in the interface `BOPersistent`. This means that any `BOJDBCManufacturer` object is able to insert, delete, and update itself in and from the database.

Because there is an association between the data manager and the business object, it does not matter what kind of connector is being used. The connector-specific code does the deflation of the business object and the communication to the connector through the referenced data manager object.

All other connectors in our sample application use the same approach. You can find specialized manufacturer business objects and data managers for JDBC, SQJL, and CICS.

B.3.3 The Servlet As a Sample for the Model View Architecture

This section shows how the servlet `ms` implements the model view controller architecture. We give you an architectural overview of the way we wired our sample application. For a better illustration, we use the Java Servlet solution. Figure 135 shows the class diagram of the final JDBC.

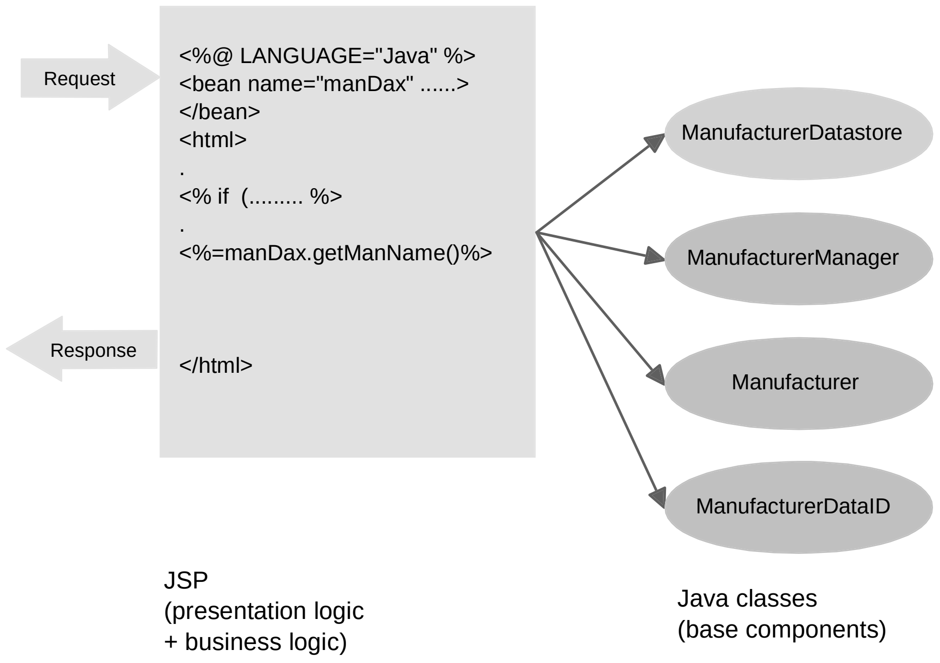


Figure 135. The JDBC/Servlet Class Diagram

The same communication and design is used for all other connectors. In our case the servlet `ms` acts as the controller and dispatches the incoming events to the view.

To keep it generic, we use factories to instantiate the selected data manager and view. This is possible because the interface for all connectors is the `DataManager`, or the `BOManufacturer`, or the `BOPersistent` interface.

After the factories create the view or the data manager, the servlet is ready to process events through its methods `doGet` or `doPost`.

As Figure 135 on page 315 shows, the connection between the view, the business object and the data manager is made through the base classes. As long as this interface stays the same for all the different connectors, the application does not have to change any of its logic. The factory creates the desired implementations and does the cast to the base class. Then the Java runtime environment ensures that the specific implementation of the methods is executed.

To give you another idea of the message flow between the different objects Figure 135 on page 315 shows an Object Interaction Diagram (OID).

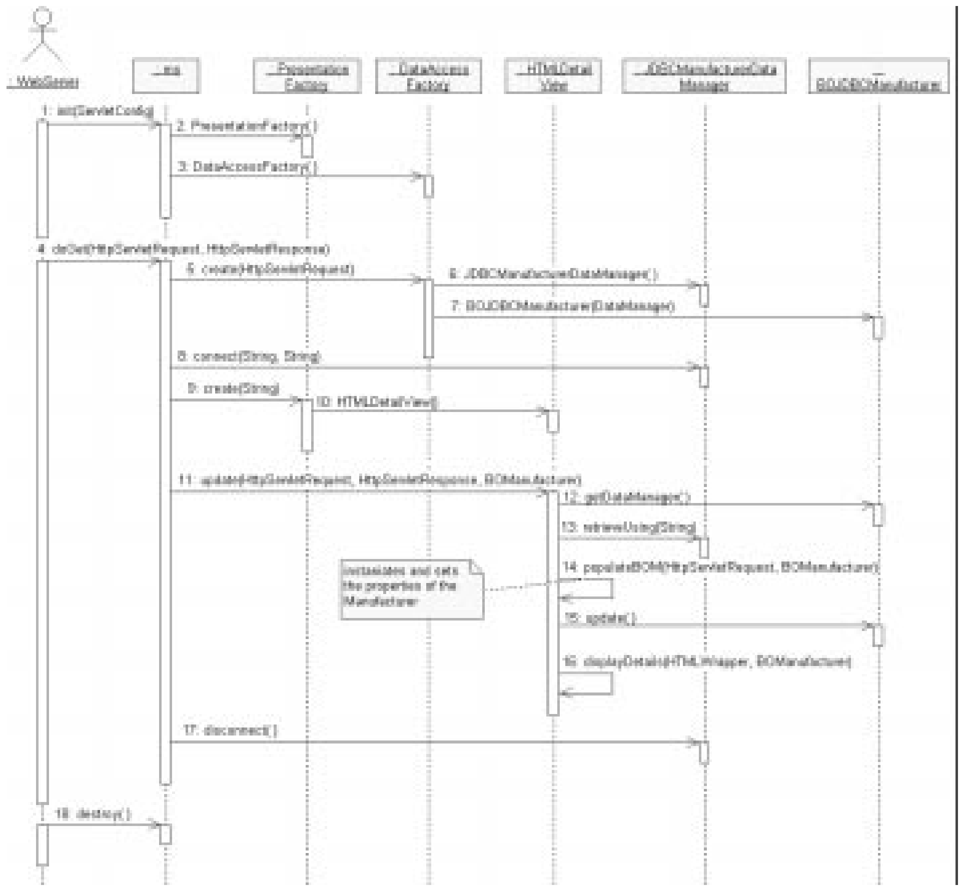

: WebServer
: ms
: PresentationFactory
: DataAccessFactory
: HTMLDetailView
: JDBCManufacturerDataManager
: BOJDBCManufacturer
1: init(ServletConfig)
2: PresentationFactory( )
3: DataAccessFactory( )
4: doGet(HttpServletRequest, HttpServletResponse)
5: create(HttpServletRequest)
6: JDBCManufacturerDataManager( )
7: BOJDBCManufacturer(DataManager)
8: connect(String, String)
9: create(String)
10: HTMLDetailView()
11: update(HttpServletRequest, HttpServletResponse, BOManufacturer)
12: getDataManager( )
13: retrieveUsing(String)
14: populateBOM(HttpServletRequest, BOManufacturer)
Instantiates and sets the properties of the Manufacturer
15: update( )
16: displayDetails(HTMLWrapper, BOManufacturer)
17: disconnect( )
18: destroy( )

Figure 136. The Servlet/JDBC Object Interaction Diagram

This demonstrates the message flow between the servlet, the data manager and the view:

As shown in this figure:

1. The Web server gets the HTML request to start a servlet. It instantiates the servlet through its default constructor and makes a call to the servlet's `init` method.

2. The servlet constructs the `PresentationFactory`.
3. The servlet constructs the `DataAccessFactory`. Finally, the control returns to the Web server.
4. The Web server makes a call to the servlet's `doGet` method.
5. The servlet calls the `DataAccessFactory` to create the `JDBCManufacturerDataManager` and gets it.
6. The servlet calls the `DataAccessFactory` to create the `JDBCManufacturer` business object.
7. The servlet calls the `JDBCManufacturerDataManager` to `connect` to the database.
8. The servlet calls the `PresentationFactory` to create the `HTMLDetailView` and gets it.
9. The servlet calls the `HTMLDetailView` to `update` the client's HTML-page.
10. The HTML-View `update` method calls `getDataManager` on `BOJDBCManufacturer`.
11. The HTML-View `update` method calls `retrieveUsing` on the data manager to return the requested manufacturer.
12. The HTML-View `update` method calls `populateBOM` on itself, rearranging the HTML output.
13. The HTML-View `update` method calls `update` on the `BOJDBCManufatcurer`.
14. The HTML-View `update` method calls `displayDetails` on itself, sending HTML to the client. The flow returns to the servlet.
15. The servlet calls the `JDBCManufacturerDataManager` to `disconnect` from the database.
16. The server calls the servlet's `destroy` when the servlet's life cycle ends.

This control flow is the same for all other connectors when using the servlet as the front end.

If the client is an applet or an application, no view factory is used. However, the communication to the connector (`DataManager`) is the same.

B.4 The Package Structure for the Sample Application

Java packages are similar to subdirectories. The sample application uses several packages, as described in the following list, to organize the classes as members of the application.

redbook	The root for the application; all member classes are stored in subpackages.
redbook.common	All classes that do not fit in one of the other packages.
redbook.data	The Business Object (BO) and its base classes.
redbook.data.cics	Business objects, derived from the one in the data package, that implement special functionality for the `CICS` connector.

redbook.data.jdbc	Business objects, derived from the one in the data package, that implement special functionality for the `JDBC` connector.
redbook.data.sqlj	Business objects, derived from the one in the data package, that implement special functionality for the `JDBC` connector.
redbook.data.management	Base package for all classes that are related to the data management as described earlier.
redbook.data.management.cics	Datamanager classes specialized for the `CICS` connector.
redbook.data.management.jdbc	Datamanager classes specialized for the `JDBC` connector.
redbook.data.management.sqlj	Datamanager classes specialized for the `SQLJ` connector.
redbook.rmi.server	All classes for the OS/390 RMI application server.
redbook.servlets	All servlets used in the sample application. Actually there is only one: the `ms.` (Manufacturer Servlet).
redbook.views	All classes that are common to views.
redbook.views.awt	Views that are implemented using Sun's `AWT` classes.
redbook.views.html	Views that are implemented in Java for generating the look and feel of the sample application using the servlet and HTML.

B.5 Connecting to DB2 on OS/390

This section discusses the sample application and its classes that are necessary for connecting to DB2.

B.5.1 Different Connections You Can Make

There are several ways to connect to DB2 on a OS/390 server machine:

1. Connect to DB2 using JDBC.
   - You can talk directly to a OS/390 DB2 by having a servlet running on a Web server on OS/390. Then the servlet and its Java classes do the connection to the database. The application must load the corresponding JDBC driver, which is:
     - `ibm.sql.DB2Driver`
   - You can talk to the OS/390 database by having DB2 Connect installed on a client workstation. Therefore, this workstation is considered to be the application server. For this kind of setup, a Web server must be installed on the application server, too. In addition to the previous setup, you can now connect to the database in two different ways:

  - Using a servlet on the application server workstation. For this scenario, the application must load the following driver:
    - `COM.ibm.db2.jdbd.app.DB2Driver`
  - Using an applet which connects from any client to the application server and talks to DB2 on OS/390. For this scenario you must start the following on the application server:
    - `db2jstart`

    The Web client running the applet must then load the following driver:
    - `COM.ibm.db2.jdbd.net.DB2Driver`

2. Connect to DB2 using SQLJ.
   - Currently there is only one way to connect to DB2 using SQLJ. The application server must reside on OS/390. The only driver available at the moment is:
     - `COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`

B.6 Explaining the Code

Most of the code in the sample application is easy to understand. Nevertheless, we explain the methods where the action takes place so that you get a better understanding of what we are doing.

For each connector, you find the code that talks to or uses the connector in the same methods.

The functionality is split: Tasks that are related to the connector itself and the functions that deal with a couple of business objects you will find in the derived `DataManager` classes. Functions that deal with just one object you will find in the derived `BOmanufacturer` classes.

Since this section deals with DB2, we explain the code in the relevant methods of the following:

- `JDBCDataManager`
- `JDBCManufacturerDataManager`
- `JDBCManufacturer`

We start at the `JDBCDataManager`:

```
/**
 * connect method comment.
 */
public final void connect() throws SQLException,
                                   ClassNotFoundException
{
      // call the overloaded 'connect' methode
      connect( getUserid(), getPassword() );
}
/**
 * connect method comment.
 */
public final synchronized void connect(String userId,
                                       String password)
                              throws SQLException,
                                     ClassNotFoundException
{
      // load the JDBC-driver and register with driver manager
      Class.forName( getJDBCDriver() );
      //userid or password ?
      if( null == userId   || userId.equals( "" ) ) &&
        ( null == password || password.equals( "" ) ))
        // no password or userid !
        // connect without
        setConnection( DriverManager.getConnection( getUrl() ) );
     // no, we have password or userid
     else
       // connect using the userid and password
       setConnection( DriverManager.getConnection( getUrl(),
                      userId,
                      password ) );
     {
         setConnected( true );
         setStatement( getConnection().createStatement() );
     }
}
/**
 * disconnect method comment.
 */
public final synchronized void disconnect() throws SQLException
{
     // do we have a statement ?
     if( null != getStatement() )
          // yes, close it
          getStatement().close();
     setStatement( null );

     // do we have a connection ?
     if( null != getConnection() )
          // yes, close it
          getConnection().close();
     setConnection( null );
}
```

As you can see, the manager does little except connect and disconnect to and from the database (although it also provides the application with attributes such as the connection, the drivername and the SQL-statement).

The work is done by the `JDBCManufacturerDataManager`; see the following methods:

```
/**
 * This method was created in VisualAge.
 * @return java.lang.StringBuffer
 */
protected StringBuffer makeQuery( String query ) {

     // create a StringBuffer with the right
     // select statement
     StringBuffer sb = new StringBuffer( "SELECT man_name,
                                                 man_address,
                                                   man_city,
                                                   man_state," ).
              append( " man_zip, man_con_last_name,
                          man_con_first_name, man_con_phone_ac," ).
              append( "man_con_phone_ex, man_con_phone_nr,
                          man_con_ext, man_con_email " ).
              append( "  FROM "  ).append( getTableName() );
     sb.append( "  WHERE man_name LIKE '" ).
         append( query ).append( "'" ).
         append( " ORDER BY MAN_NAME" );
     return sb;
}
/**
 * retreive all records that match 'query'
 * @return java.util.Enumeration
 */
public Enumeration retrieve( String query ) throws SQLException,
                                          NoSuchElementException
{
     // all elements are stored in a Vector...
     // remove, clear it
     getDataElements().removeAllElements();

     // make the SQL-query
     StringBuffer sb = makeQuery( query );

     // execute the query, get the records in a ResultSt
     ResultSet rs   = getStatement().executeQuery( sb.toString() );

     // get the next Record in the ResultSet
     while( rs.next() )
     {
          // intaniate a new business object
          BOManufacturer tmpBo = new BOJDBCManufacturer( this );

          // set each column to the corresponding
          // attribute of the business object
          tmpBo.setManufacturer( rs.getString(1) );
          tmpBo.setAddress( rs.getString(2) );
          tmpBo.setCity( rs.getString(3) );
          tmpBo.setState( rs.getString(4) );
          tmpBo.setZip( rs.getString(5) );
          tmpBo.setLastName( rs.getString(6) );
          tmpBo.setFirstName( rs.getString(7) );
          tmpBo.setPhoneac( rs.getString(8) );
          tmpBo.setPhoneex( rs.getString(9) );
          tmpBo.setPhonenr( rs.getString(10) );
          tmpBo.setExt( rs.getString(11) );
          tmpBo.setEmail( rs.getString(12) );

          // add current object reference to the Vector
          getDataElements().addElement( tmpBo );
     }

     // notify that the number of records may have changed
     fireHandleDataElementsChanged(
          new DataElementsChangedEvent( this ) );
     // return the Vector
     return getDataElements().elements();
}
/**
 * Get all records that match 'key',
 * Return just a stringlist of keys (manufacturer-name)
 * @return java.util.Enumeration
 */
public Enumeration retrieveListOfKeys( String key )
                                   throws SQLException
{
     // does the key ends with an '%' ?
     // if not, add one
     if( !key.endsWith( "%" )) key += "%";

     // get the records that match 'key'
     Enumeration e = retrieve( key );

     // remove all previously stored keys
     getKeyElements().removeAllElements();

     // loop through and get the keys into the vector
     while( e.hasMoreElements() )
         getKeyElements().addElement(
                    ((BOManufacturer )e.nextElement())
                    .getManufacturer());

     // notify that the key vector has been changed
     fireHandleKeyElementsChanged(
               new KeyElementsChangedEvent( this ) );

     // return the enumeration of the vector
     return getKeyElements().elements();
}
/**
 * Look up the on special key
 */
public BOPersistent retrieveUsing( String key ) throws Exception
{
     // did we get a key ?
     if( null == key || key.equals( "" ) )
     {
          throw new Exception( "Key not found" );
     }

     // make the query
     StringBuffer sb = new StringBuffer( "SELECT man_name,
                                          man_address,
                                          man_city,
                                          man_state," ).
     append( " man_zip, man_con_last_name, man_con_first_name,
              man_con_phone_ac," ).
     append(   "man_con_phone_ex, man_con_phone_nr, man_con_ext,
              man_con_email " ).
     append( "  FROM "  ).append( getTableName() );

     sb.append( "  WHERE man_name = '" ).
          append( key ).append( "'" );
     // look up the record
     ResultSet rs   = getStatement().
                        executeQuery( sb.toString() );
     rs.next();

     // make a new business object
     BOJDBCManufacturer tmpBo = new BOJDBCManufacturer( this );

     // get all the attributes
     tmpBo.setManufacturer( rs.getString(1) );
     tmpBo.setAddress( rs.getString(2) );
     tmpBo.setCity( rs.getString(3) );
     tmpBo.setState( rs.getString(4) );
     tmpBo.setZip( rs.getString(5) );
     tmpBo.setLastName( rs.getString(6) );
     tmpBo.setFirstName( rs.getString(7) );
     tmpBo.setPhoneac( rs.getString(8) );
     tmpBo.setPhoneex( rs.getString(9) );
     tmpBo.setPhonenr( rs.getString(10) );
     tmpBo.setExt( rs.getString(11) );
     tmpBo.setEmail( rs.getString(12) );

     return tmpBo;
}
```

You find the update, delete and insert methods in the business object (BOmanufacturer):

```
/**
 * Perform the delete method.
 */
public void delete() throws SQLException
{
     // get the jdbc-datamanager
     JDBCDataManager jdbcMgr = (JDBCDataManager )getDataManager();

     // create the statment
     StringBuffer sb = new StringBuffer( "delete from ").
          append( jdbcMgr.getTableName() ).
          append( " where man_name = '" ).
          append( getManufacturer() ).
          append( "'" );

     // execute the delete
     jdbcMgr.getStatement().executeQuery( sb.toString());
}
/**
 * Perform the insert method.
 */
public void insert() throws SQLException
{
     // get the jdbc-datamanager
     JDBCDataManager jdbcMgr = (JDBCDataManager )getDataManager();

     // make the query
     StringBuffer sb = new StringBuffer( "insert into ").
          append( jdbcMgr.getTableName() ).
```

```
            append( " (man_name, man_address, man_city, man_state,
                      man_zip, man_con_last_name," ).
            append( " man_con_first_name, man_con_phone_ac,
                     man_con_phone_ex, man_con_phone_nr," ).
            append( " man_con_ext, man_con_email, man_last_upd_uid,
                     man_last_upd_date)" );

       sb.append( " values ('" ).append( getManufacturer() ).
            append( "'," ).
            append( "'" ).
            append( getAddress() ).
            append( "',  " ).
            append( "'"  ).
            append( getCity() ).
            append( "',  " ).
            append( "'" ).
            append( getState() ).
            append( "',  " ).
            append( "'" ).
            append( getZip() ).
            append( "',  " ).
            append( "'" ).
            append( getLastName() ).
            append( "', " );

       sb.append( "'" ).append( getFirstName() ).
            append( "',  " ).
            append( "'" ).
            append( getPhoneac() ).
            append( "',  " ).
            append( "'" ).
            append( getPhoneex() ).
            append( "',  " ).
            append( "'" ).
            append( getPhonenr() ).
            append( "',  " ).
            append( "'" ).
            append( getExt() ).
            append( "',  " ).
            append( "'" ).
            append( getEmail() ).
            append("',  " ).
            append( "'" ).
            append( "JAVAID" ).
            append( "',  " ).
            append( " current timestamp)" );

       // execute the statement
       jdbcMgr.getStatement().executeQuery( sb.toString() );
  }
  /**
   * Perform the update method.
   */
  public void update() throws SQLException
  {
       // get the JDBC-datamanager
       JDBCDataManager jdbcMgr = (JDBCDataManager )getDataManager();
```

```
        // create the query
        StringBuffer sb = new StringBuffer( "update ").
             append( jdbcMgr.getTableName() ).
             append( " set man_address ='" ).append( getAddress() ).
             append( "', man_city ='" ).append( getCity() ).
             append( "', man_state ='" ).append( getState() ).
             append( "', man_zip ='" ).append( getZip() ).
             append( "', man_con_last_name ='" ).
             append( getLastName() ).
             append( "', man_con_phone_ac ='" ).
             append( getPhoneac() ).
             append( "', man_con_phone_ex ='" ).
             append( getPhoneex() ).
             append( "', man_con_phone_nr ='" ).
             append( getPhonenr() );

        sb.append("', man_con_first_name ='" ).
             append( getFirstName() ).
             append("', man_con_ext ='" ).append( getExt() ).
             append( "', man_con_email ='" ).append( getEmail() ).
             append( "', man_last_upd_uid ='JAVAID" ).
             append( "', man_last_upd_date  = current timestamp " ).
             append( "where man_name = '").
             append( getManufacturer() ).append( "'" );

        // execute the statement
        jdbcMgr.getStatement().executeQuery( sb.toString() );
   }
```

Appendix C. Special Notices

This publication is intended to help:

- Information Technology Architects understand how Network Computing applications can be deployed using Webserver and Java support on OS/390.
- Application Programmers to develop Java components for OS/390 and create solutions for connecting existing data and applications on OS/390 with Java components in an NC environment.
- System Programmers to set up the required infrastructure on OS/390 to use Lotus Domino Go Webserver Release 5.0, servlets, Java Server Pages, Java applications and Java-connectors to OS/390 subsystems.

The information in this publication is not intended as the specification of any programming interfaces that are provided by Java, VisualAge for Java, Java Database Connectivity (JDBC), SQLJ, CICS Gateway for Java, IMS TCP/IP OTMA Connectivity, MQSeries Client for Java and Java Server Pages.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

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

Advanced Application Communication System/2	AIX®
AlphaWorks	AnyNet®
Applet.Author	AS/400®
C++/MVS	C/MVS
CICS®	CICS/ESA®
CICS/MVS®	CICSPlex®
COBOL/370	Cryptolope
DATABASE 2	DB2®
DB2® Connect	DB2® Client Application Enablers
DB2® Universal Database	Distributed Relational Database Architecture
DRDA®	Encryptolope
Hummingbird	IBM®
IMS	IMS/ESA®
Language Environment®	MQ
MQSeries®	MVS® (logo)
MVS/ESA	Open Blueprint®
OS/390	Powered by S/390
RACF	S/390®
SOMobjects	S/390 Parallel Enterprise Server
VisualAge®	

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java, HotJava, JavaBeans, JDK, JDBC and Solaris
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 D. Related Publications

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

D.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 333.

- *Programming with VisualAge for Java Version 2*, SG24-5264
- *Integrating Java with Existing Data and Applications on OS/390*, SG24-5142
- *Enterprise Web Serving with the Lotus Domino Go Webserver for OS/390*, SG24-2074
- *OS/390 TCP/IP OpenEdition Implementation Guide*, SG24-2141

D.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs:

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbook	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037

D.3 Other Publications

These publications are also relevant as further information sources:

- *IBM HTTP Server for OS/390 Planning* SC31-8690
- *Domino Go Webserver 5.0 for OS/390: Webmaster's Guide*, SC31-8691
- *Domino Go Webserver 5.0 for OS/390: Messages*, SC31-8692
- *OS/390 V2R6.0 Planning for Installation*, GC28-1726
- *Programming Directory for Domino Go Webserver for OS/390*, GI10-6780
- *OS/390 V2R6.0 NFS Customization and Operation*, SC26-7253
- *OS/390 V2R4.0 MVS Programming: Resource Recovery*, GC28-1739
- *OS/390 V2R6.0 NFS User's Guide*, SC26-7254
- *OS/390 Release 6 Domino Go Webserver 5.0 Web Programming Guide*, SC34-4743 (available softcopy only)
- *TCP/IP OpenEdition User's Guide*, GC31-8305
- *DB2 for OS/390 V5 Call Level Interface Guide and Reference*, SC26-8959

- *DB2 for OS/390 V5 Command Reference*, SC26-8960
- *DB2 for OS/390 Version 5 Application Programming Guide and Reference for Java*, SC26-9547
- *Quick Beginnings Windows NT, Documentation for TXSeries V4.2*, GC33-1879
- *CICS Family: Communicating from CICS on System/390*, SC33-1697
- *CICS TS for OS/390 V1R2 CICS Internet and External Interfaces Guide*, SC33-1944
- *Software Development: Processes and Performance/The WebSphere Application Server Architecture Guide*, G321-5680
- *Open Edition MVS User's Guide*, SC23-3013
- *IMS/ESA Customization Guide*, SC26-8020

D.4 External Publications

- *Java Secrets*, by Elliotte Rusty Harold. Published by IDG Books Worldwide Inc, 1997, California. ISBN 0-7645-8008-8.
- *Essential JNI, Java Native Interface*, by Rob Gordon. Published by Prentice Hall, 1998, New Jersey. ISBN 0-13-679895-0.
- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch. Published by Addison-Wesley. ISBN 0-20-163361-2.
- *Java servlets*, by Karl Moss. Published by McGraw-Hill, 1998. ISBN 0-07-913779-2.

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** `http://www.redbooks.ibm.com/`

  Search for, view, download or order hardcopy/CD-ROMs redbooks from the redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

  Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders via e-mail including information from the redbook fax order form to:

In United States:	e-mail address: usib6fpl@ibmmail.com
Outside North America:	Contact information is in the "How to Order" section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl/

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the "How to Order" section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl/

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the "How to Order" section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl/

This information was current at the time of publication, but is continually subject to change. The latest information for customers may be found at `http://www.redbooks.ibm.com/` and for IBM employees at `http://w3.itso.ibm.com/`.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at `http://w3.itso.ibm.com/` and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may also view redbook, residency and workshop announcements at `http://inews.ibm.com/`.

IBM Redbook Fax Order Form

Please send me the following:

Title	**Order Number**	**Quantity**

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

□ Invoice to customer number

□ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

ITSO Redbook Evaluation

e-business Application Solutions on OS/390 Using Java: Volume I
SG24-5342-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and Fax it to: USA International Access Code + 1 914 432 8264 or:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
__Customer __Business Partner __Solution Developer __IBM employee
__None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____________

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

____

____

____

____

What other redbooks would you like to see published?

____

____

____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

____

____

____

____

____

e-business Application Solutions on OS/390 Using Java: Volume I

SG24-5342-00

SG24-5342-00
Printed in the U.S.A.