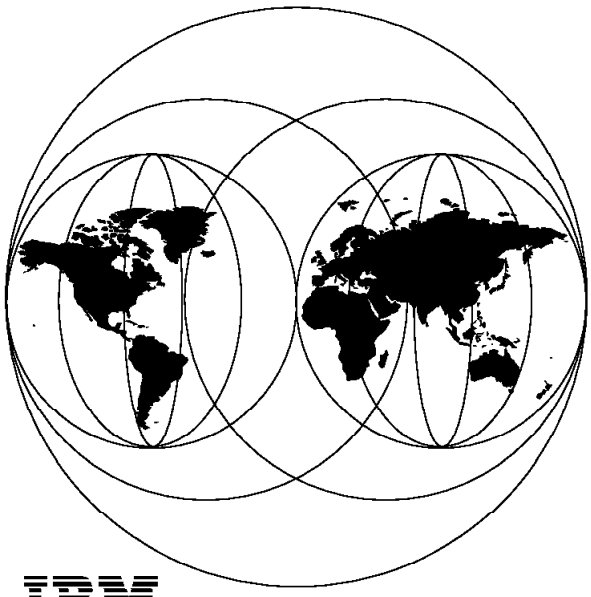
SG24-4896-00

Internet Application Development with MQSeries and Java

February 1997



IBM
International Technical Support Organization
Raleigh Center

SG24-4896-00

International Technical Support Organization

Internet Application Development with MQSeries and Java

February 1997

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special Notices” on page 151.

First Edition (February 1997)

This edition applies to:

- IBM OS/2 Warp Version 4.0
- IBM Internet Server for OS/2 Version 4.2 beta
- IBM MQseries for OS/2 Version 2.01
- IBM MQSeries Client for Java SDK, program number 5639-C34
- Netscape Navigator Version 2.02 for
- Microsoft Windows NT Version 4.0 Workstation
- Microsoft Internet Explorer Version 3.0
- Microsoft Windows 95

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

Contents

Figures

Tables

Preface

This redbook helps you integrate IBM′s award-winning middleware MQSeries and the Internet. It gives you a broad understanding of what has to be done to connect Internet/intranet users to legacy systems in your enterprise. You may gain significant advantages by using the MQSeries Client for Java when you introduce intranet-based client/server solutions, or when you want to provide Internet access to some of your enterprise applications.

The Internet technology provides low-cost easy access to global communications, while MQSeries connectivity provides high integrity with assured delivery and time independence.

Today, Java is on everyone′s mind. Java is considered to be the premier programming language for Internet applications. The MQSeries Client for Java is written in this object-oriented language. With this software, the user of an Internet terminal can become a true participant in transactions, rather than just a giver and receiver of information.

This redbook introduces you to Java, the HyperText Markup Language (HTML), MQSeries, Web browsers, and Web servers. Programmers with no knowledge of MQSeries or the Internet can use this redbook and the accompanying diskette to get started with this new technology. This publication also helps you to install and configure the various software products you need for application development.

This redbook provides an example of a demonstration application and shows what an Internet order entry system looks like. A Java applet that runs in the client machine uses MQSeries Client for Java to retrieve product lists from a server. With a graphical user interface, the user selects items from these lists and places them into order form. When completed, the applet uses MQSeries again to forward the order to a server for processing. The server program updates inventory files and responds, asynchronously, with a message to the end user indicating if the order can be filled or not.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the Systems Management and Networking ITSO Center, Raleigh.

Frank Brasch is a Systems Engineer/Consultant in Munich, Germany. There he is responsible for object-oriented technology, application development

and design of client/server applications with MQSeries on workstation platforms.

Frank has been with IBM for 26 years. Before joining the OSC three years ago, he worked for about five years in the Education and Research Marketing department in Munich as a systems engineer for high-end graphics applications on RS/6000. Prior to that he helped to design, develope and implement the Directory Assistance System for the German Telekom.

Frank is a co-writer of the redbook *SOMobjects: Management Utilities for Distributed SOM,* GG24-4479.

George T. Carey is an I/T Specialist currently with the OSC/NCC based in Atlanta, GA, USA. He has 24 years experience in the I/S world of which the last seven have been with IBM USA serving in various area-wide technical marketing support roles. He has performed a full gamut of roles for the I/S professional in both commercial and scientific projects.

A quick sampling includes being a principal and Director of Software Development in a commercial software systems house, a scientific programmer/analyst with CSC and NASA, consultant, project leader of ″OEC MegaDemo″ to IBM US Vice President et al, winner of IBM Director′s award, and progenitor of the IBM National Support offering known as the AIX Support-Line.

He is co-author of the redbook *MQSeries Three Tier, Examples for Windows Clients and AIX Servers,* SG24-4664-00.

George holds a B.A. in Mathematics with graduate work in Linear Programming and Computer Science.

Adrian Colyer majored in Computer Science at the University of Southampton, and graduated in 1992 with a first class honors degree. He joined IBM later that year, working on distributed object applications using C++, SOM and DSOM. In particular, he worked on the CICS Systems Manager for AIX product providing a distributed systems management capability for CICS/6000. In 1996 Adrian joined the MQSeries development group where he designed and developed the MQSeries Client for Java. He currently works in the MQSeries Internet team developing new applications for MQSeries and the internet.

Jin JaeWook is an I/T specialist in Korea and has been with IBM for 11 years. Jin acquired software development experience in the Korean Software Development Institute. He developed an Information Retrieval

System for HanGeul on MVS/CICS, announced this product in the Korean market, and supported IBM marketers.

Jin developed OV/CICS (OfficeVision) and BAS/400 (Decision Support System) as a worldwide product. He also participated in the biggest SI project in IBM Korea, for the Korean Mobile Telecommunication Company. After that, as a follow-on project, he applied MQSeries to the system in several areas.

Dieter Wackerow is an Advisory ITSO Specialist for MQSeries in the Systems Management and Networking ITSO Center, Raleigh, and was the leader of this project. His areas of expertise include application design and development for various industries, performance evaluations, capacity planning, and modelling of computer systems and networks. He also wrote a simulator for banking hardware and software. He taught classes and has written on performance issues, application development for the banking industry, and about MQSeries.

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

Linda Robinson
Systems Management and Networking ITSO Center, Raleigh

Wendy Ling
IBM Hursley, England

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 167 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Home Pages at the following URLs:

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

- Send us a note at the following address:

  `redbook@vnet.ibm.com`

Chapter 1. MQSeries Client for Java Positioning

MQSeries provides an excellent infrastructure for access to enterprise applications and for development of complex Web applications. A service request from a Web browser can be queued and processed when possible, thus allowing a timely response to be sent to the end user regardless of system loading. By placing this queue close to the user in network terms then the timeliness of the response is not impacted by network loading. In addition the transactional nature of MQSeries messaging means that a simple request from the browser can be expanded safely into a sequence of individual backend processes in a transactional manner.

If your enterprise fits any of the following scenarios, you can gain significant advantages by using MQSeries Client for Java:

- A medium or large enterprise that is introducing intranet-based client/server solutions. Here Internet technology provides low-cost easy access to global communications, while MQ connectivity provides high integrity with assured delivery and time independence.
- A medium or large enterprise with a need for reliable business-to-business communications with partner enterprises. Here again, the Internet provides low-cost easy access to global communications, while MQ connectivity provides high integrity with assured delivery and time independence.
- A medium or large enterprise that wishes to provide access from the public Internet to some of their enterprise applications. Here the Internet provides global reach at a low cost, while MQ connectivity provides high integrity through the queuing paradigm. In addition to low cost, the business can achieve improved customer satisfaction through 24 hour a day availability, fast response, and improved accuracy.
- An Internet Service provider, or other Value Added Network provider, such as EDI brokers. These companies can exploit the low cost and easy communications provided by the Internet and add the value of high integrity provided by MQ connectivity. An Internet Service provider who exploits MQ can immediately acknowledge receipt of input data from a Web browser, guarantee delivery, and provide an easy way for the user of the Web browser to monitor the status of the message.

For more information about the MQSeries Client for Java product refer to Chapter 7, “MQSeries Client for Java” on page 73.

Chapter 2. Introduction

This publication provides information resulting from the development of an application that exploits Internet servers and browsers, IBM′s award-winning middleware MQSeries, the HyperText Markup Language (HTML), and the product that today is in everyone′s mind, Java.

The project to write this redbook included various platforms and a variety of software and programming languages. Its purpose was to develop an example application that uses MQSeries over the Internet. This redbook documents the efforts in this project. It helps application designers and developers to get started writing programs with Java and MQSeries.

The project lasted six weeks. The programs written for this project have been tested and perform all functions required for the demonstration to work. However, the code is not intended to be state of the art. None of the authors had prior programming experience with Java.

Note: The programs developed during this project are included on a diskette supplied with this book.

2.1 About the Objectives

We based our example on a real client/server application that is used to sell cellular phones and pagers from various producers to businesses and individuals. Our application demonstrates how to buy (or sell) fruit, vegetables and other products via the Internet.

Our objective was to use messaging and queueing in a Java applet. To be more specific, we had MQSeries clients communicate with an MQSeries server program using the MQSeries Server Connection facility. An MQSeries server and Web server run in the same machine. The supporting software is MQSeries Client for Java.

Since the order-entry application is for demonstration purposes only we did not implement functions such as address verification, credit check, back order processing, and other programs that may be required to run in servers or hosts to make the application complete. However, we implemented a full-function order mechanism comprising an HTML home page, a Java applet that displays a graphical user interface (GUI), a server program written in C and a set of inventory files.

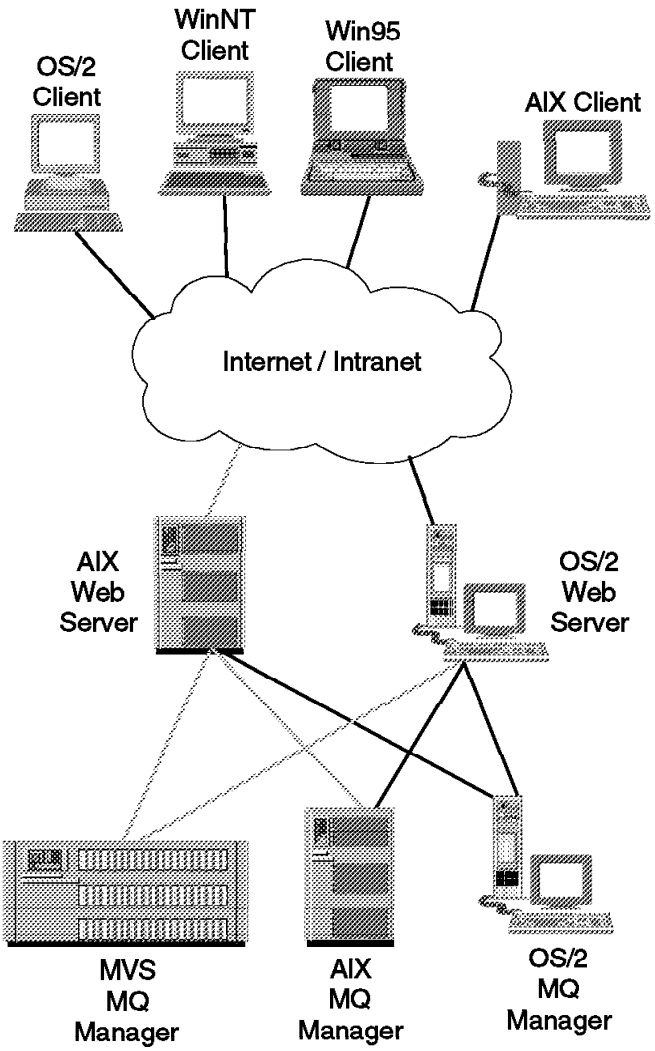


Figure 1. Computer Network Envisioned for This Project

Figure 1 shows the three tier computer network envisioned for this application. All clients run the same Java applet. The two Web servers are also MQSeries servers supporting either client. One can imagine the servers to be two different stores to shop in. For this demonstration, we ignored the third tier. However, the server program can easily be expanded to request services or data from any or all backend systems.

The Java applet has access to three product lists that could be databases in different hosts or servers. For this demonstration, the server program maintains three flat files. Each file contains a list of products with price and quantity available for sale. The inventory is updated each time an order is processed.

The operator requests one product list at a time. It will be displayed in the GUI (applet). From each list he or she can order one or more items by selecting a product and typing a quantity. All selected items are automatically placed in an order list. We limited it to hold a maximum of ten entries. After the order list is completed its content is moved into a message and sent to the server for processing. The operator can then obtain status information about his or her order, indicating if it can be fulfilled or not.

The next section describes the demonstration program and how to run it from wherever you are. The following chapters describe what software you need to develop your own applications, how to obtain, install and use it. This redbook should provide all necessary information and guidance for developing Java applets for the Internet.

Where to find the demo

The demonstration program is currently available on these servers:

- http://w3.itso.ral.ibm.com/mq (AIX)
- http://mercury.itso.ral.ibm.com (AIX)
- http://mqjava.itso.ral.ibm.com (OS/2)

2.2 About the Demonstration Program

You must have a Java-enabled Web browser such as the Netscape Navigator to execute the MQSeries client for Java demonstration. In the Web page shown in Figure 2 on page 6 you see three applets. Each applet displays a clock, one for each of the time zones the authors of this publication live.

If you have a browser that is not Java-enabled such as IBM′s WebExplorer, you will see the Web page shown in Figure 3 on page 7. Instead of the three applets showing the clocks some text is displayed that tells you where to get a Java-enabled browser.

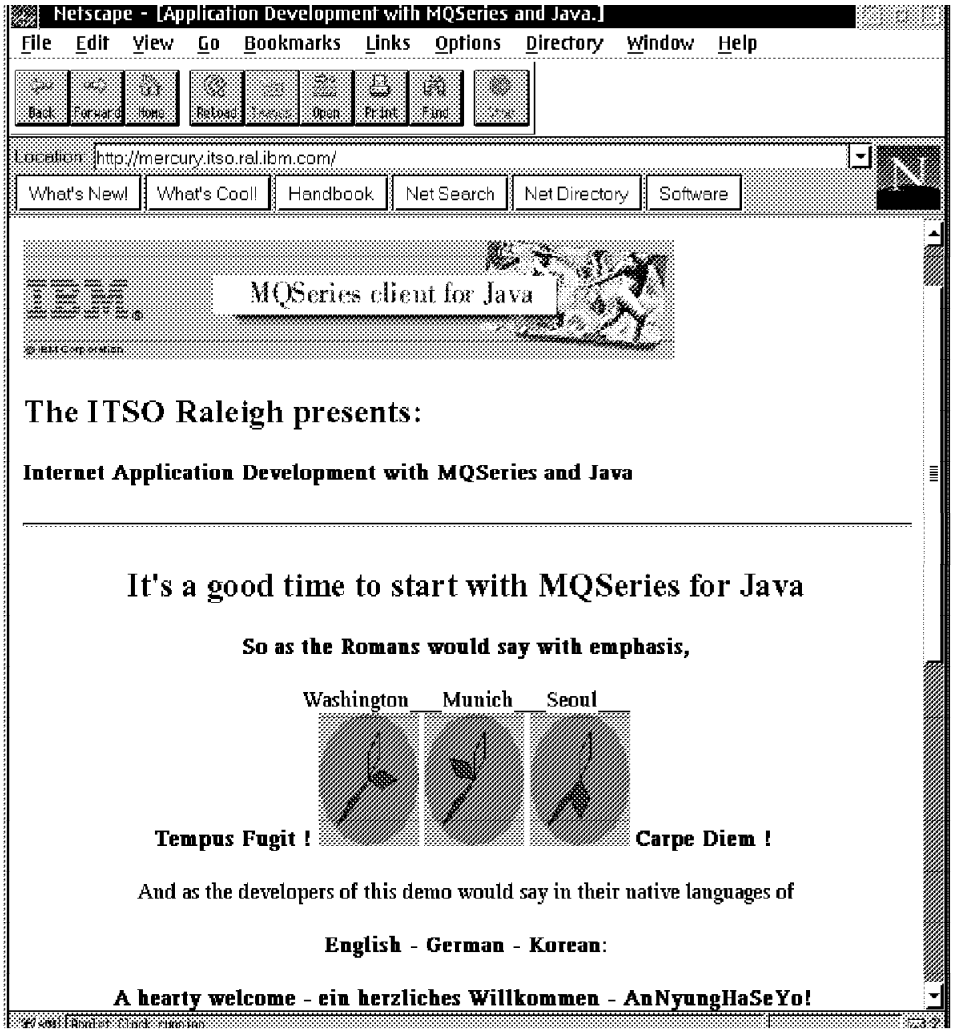


Figure 2. Demo Web Page Shown by Netscape Navigator (Java-Enabled)

The HTML file contains hyperlinks to Web pages featuring:

- The IBM Internet Connection Family
- The MQSeries Client for Java Software Developer's Kit (SDK)
- This and other redbooks
- The demonstration described in this book

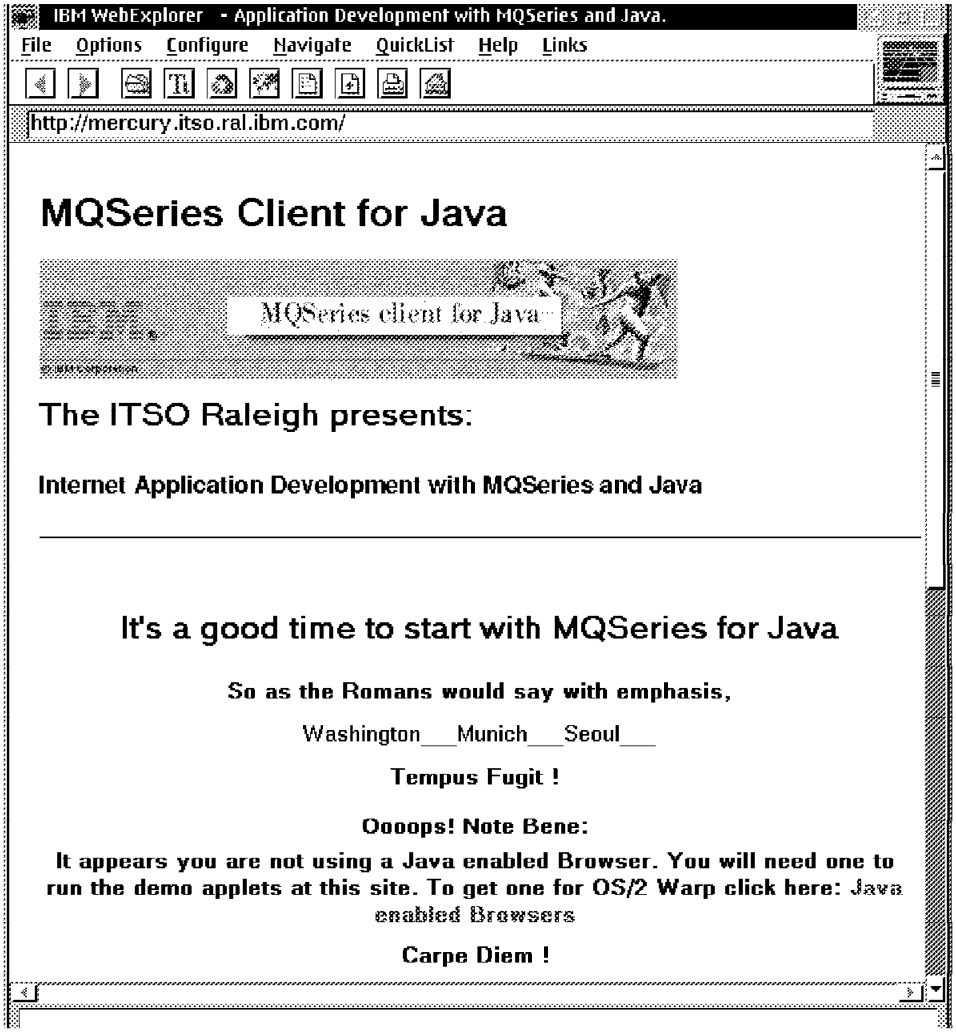


Figure 3. Demo Web Page Shown by IBM WebExplorer (Not Java-Enabled)

The above Web page contains a hyperlink to the Web page where you can download the Netscape Navigator for OS/2 Warp:

`http://www.internet.ibm.com/browsers/netscape/warp`

Figure 4 on page 8 shows the entry form written as a Java applet.

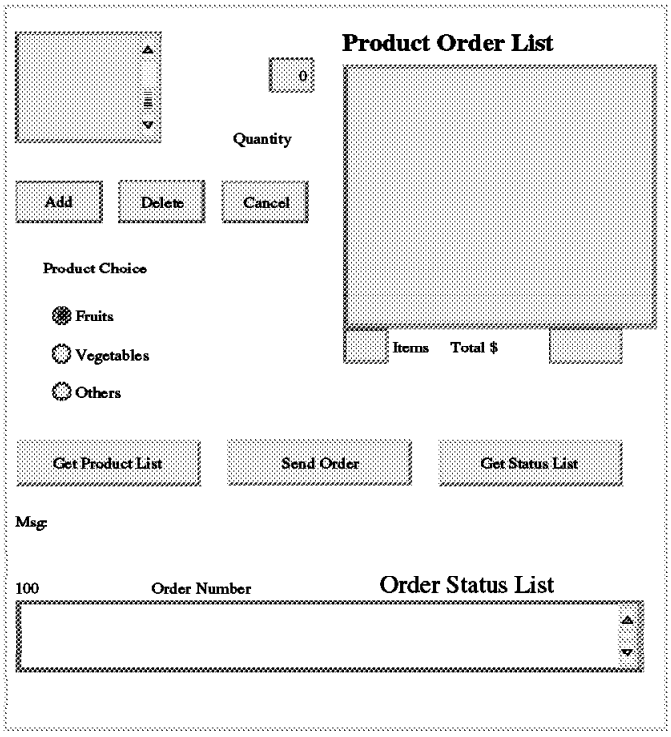


Figure 4. Order Entry Applet from the Demo Application

Some of the readers already know HTML; some may have written Java programs; others have used MQSeries before; and some may not know either. Therefore, this publication contains brief introductions to HTML, Java and MQSeries.

In the following chapters you can read about:

- Software and installation
- The basics of HTML
- The basics of Java
- An introduction to MQSeries
- The MQSeries client for Java SDK
- The sample application

Chapter 3. Installation

Figure 1 on page 4 gives you a good overall view of the hardware considered for this MQSeries/Java project. We installed the following software:

1. OS/2 Server (PS/2 Model 95)
   - OS/2 Warp Server Version 3.0
   - TCP/IP Version 3.0 (included)
   - MQSeries for OS/2 Version 2.0.1
2. OS/2 Workstation
   - OS/2 Warp Version 4.0 (Merlin)
     formatted for the High Performance File System (HPFS)
     including:
     - TCP/IP Version 3.0
     - Java for OS/2
   - Internet Connection Secure Server for OS/2
   - MQSeries for OS/2 Version 2.0.1 (client)
3. WinNT Workstation
   - WinNT Version 4.0 Workstation
     including:
     - TCP/IP
     - Java Compiler
     - Java SDK 1.02
   - Internet Explorer Version 3.0
   - Different tools for building HTML pages and Java development
4. AIX Server and workstation
   - AIX Version 4.1
   - Apache Internet Server Version 1.1.1
   - MQSeries for AIX

In the following sections we describe the installation of some of the Internet software we used.

3.1 IBM Internet Connection Secure Server

This chapter describes how to obtain, install and administer the IBM Internet Connection Secure Server (ICSS). This code has to be installed on the OS/2 and AIX servers in our configuration.

3.1.1 How to Get the ICSS

Using a Web browser of your choice, look at the Internet Connection Family Web page at:

http://www.ics.raleigh.ibm.com

As of the writing of this book, Version 4.2 Web Server betas were available. Selecting this product brings you to the page:

http://www.ics.raleigh.ibm.com/ics/icfbetas.htm

From here select **Internet Connection Secure Servers (Version 4.2)**. This brings you to a page that lists the servers for various products. For our configuration, we download:

- Internet Connection Secure Server for OS/2 (Version 4.2 beta)
- Internet Connection Secure Server for AIX (Version 4.2 beta)

Follow the instructions and you will receive the files:

- os42b1.zip, about 3.2 MB, for OS/2
- aix42b1.tar.Z, about 3.5 MB, for AIX

  If you use the FAT file system give it the file extension trz.

Note: The ICSS Beta-Version 4.2 cannot only handle CGI, but also Java within the HTML pages.

Use ftp to transfer the compressed file to the AIX system. A sample scenario is shown below. The commands are shown in bold.

```
d:\>ftp mercury
IBM TCP/IP for OS/2 - FTP Client ver 15:51:28 on Nov 19 1994
Connected to mercury.itso.ral.ibm.com.
220 mercury FTP server (Version 4.1 Sun Jul 28 12:35:09 CDT 1996) ready.
Name (mercury): root
331 Password required for root.
Password: .....
230 User root logged in.
ftp> cd usr/lpp/internet
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> put aix42b1.trz aix42b1.tar.Z
```

```
200 PORT command successful.
150 Opening data connection for aix42b1.tar.Z.
226 Transfer complete.
local: aix42b1.trz remote: aix42b1.tar.Z
3512578 bytes sent in 10.09 seconds (339 Kbytes/s)
ftp>quit
```

3.1.2 How to Install the ICSS on OS/2

This section describes how you install and administer the ICSS for OS/2. For other operating systems, you have to follow the specific instructions for that operating system. Follow these steps after you downloaded the ICSS:

Step 1. Unzip the file:

To unzip use PKUNZIP or a similar program with the -d option. The -d option will create directories stored in the zip file. Let us assume that the file is in the root directory of the C drive. Execute the following commands:

```
E:\>md webserv
E:\>cd webserv
E:\WEBSERV>pkunzip -d c:\os42b1.zip
```

Notes:

a. The Web server must be installed in an HPFS partition.

b. Read the file README.1ST in the installation directory.

Step 2. Start the installation:

a. To invoke the installation program, type:

```
E:\WEBSERV>install
```

b. In the installation window, click on **Continue**.

c. In the subsequent Install window, click on **OK** to have the installation program update the CONFIG.SYS file.

Step 3. Select the components you want to install:

In the Install - directories window, shown in Figure 5 on page 12, you can:

- Select the components you want to install.
- Overwrite the default directories for the components.

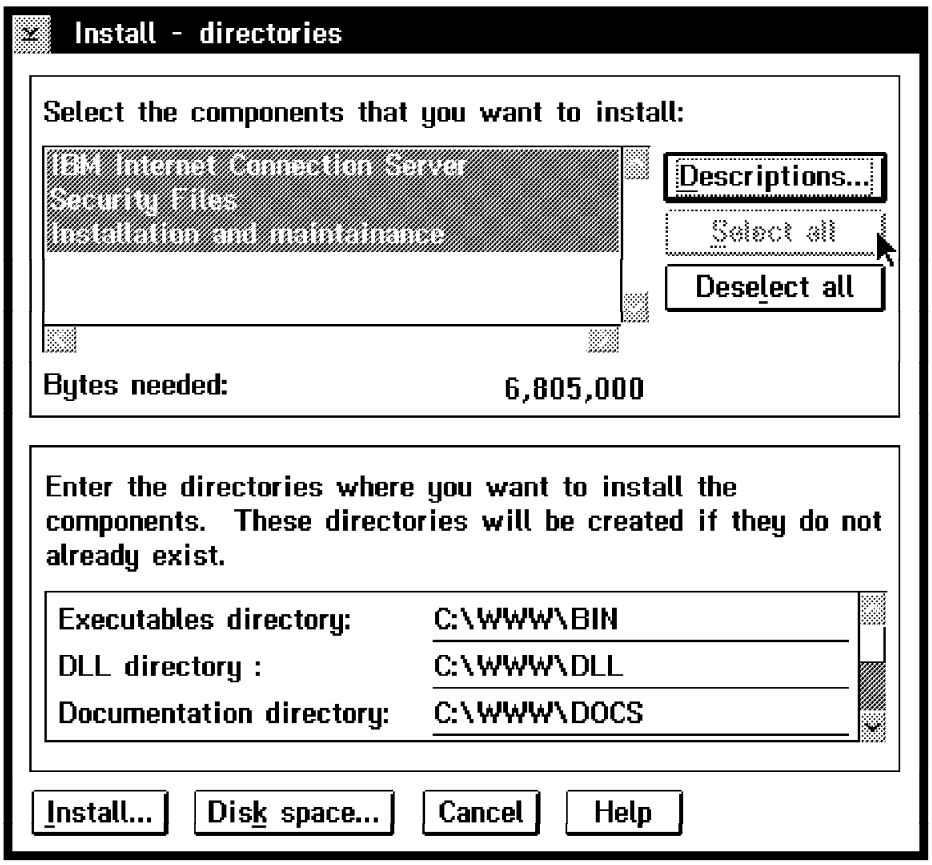


Figure 5. Internet Connection Server Installation

The directories you see in the bottom half of the window define where you want to install the Internet Connection Server components and where you want to store the resources you will be making available through the Internet Connection Server.

You can change these paths by clicking **Disk space** and selecting the drive where you want the directories installed.

Note: The gateway code is installed in the same directory as the client for OS/2.

Attention

Use the scroll bar to see the complete list of directories. The directories must be in a partition formatted for the High Performance File System (HPFS).

A brief description of the contents of the directories follows:

Executables directory
The Internet Connection Server executable program files and other related files are installed in this directory.

DLL directory
The Internet Connection Server DLL files are installed in this directory.

Documentation directory
This directory contains the online documentation for the product.

CGI Bin scripts directory
Into this directory, put your script programs that use the Common Gateway Interface (CGI). Also, the Internet Connection Server htimage program is installed in this directory.

HTML directory
This directory is for your HTML documents. The Internet Connection Server sample HTML pages and the Internet Connection Server Front Page are installed in this directory.

Remote admin directory
This directory contains files used by the Internet Connection Server Configuration and Administration Forms.

Icons and graphics directory
The default directory list icons are installed here. You may also choose to put your own icon and graphics files on this directory.

Logs directory
This directory is for log files written by the Internet Connection Server.

Labels directory
Here you can find the label text, if your are writing diskettes and want to have the correct labels.

Step 4. Click on **Install...** in the Installation window shown in Figure 5 on page 12 to continue. You will be presented the window in Figure 6 on page 14.

Step 5. Configure the Internet Connection Secure Server:

In the Internet Connection Secure Server Install Configuration window, you may change the default values for:

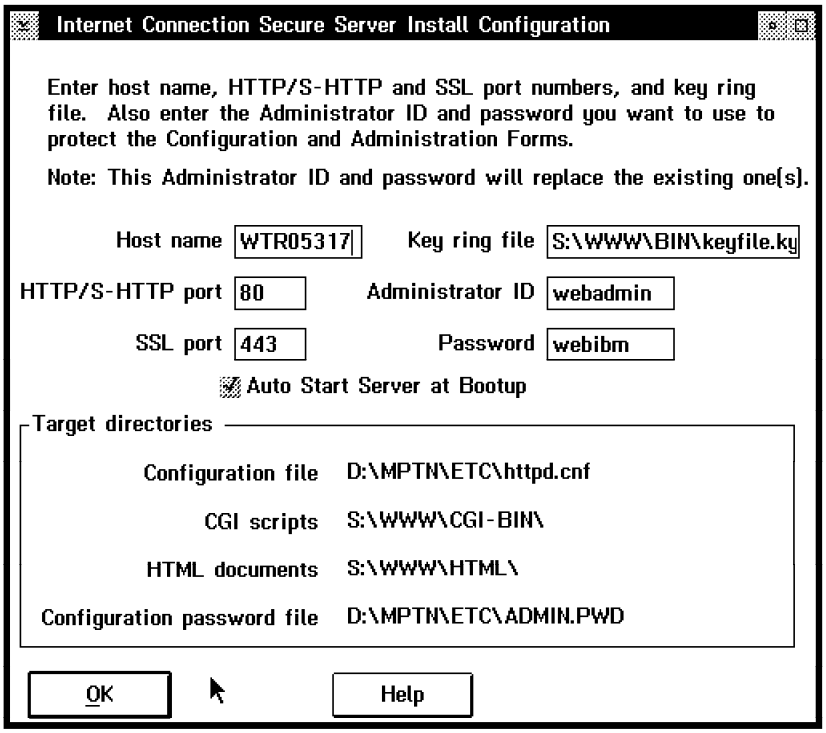


Figure 6. Internet Connection Server Installation - Configuration

Host name

The default value is the host name defined in your CONFIG.SYS file. If you want to use an alias, change this field to a fully qualified host name that is defined in your domain name server.

HTTP/S-HTTP port

The default value of 80 is the well-known port number for Hypertext Transfer Protocol (HTTP) and Secure Hypertext Transfer Protocol (S-HTTP). Other port numbers less than 1024 are reserved for other TCP/IP applications. Port numbers 8080 and 8008 are commonly used for testing servers.

SSL Port

The port you want your server to listen to for requests for documents protected by the Secure Sockets Layer (SSL) protocol. The default is 443.

Key ring file
The name of the file where you want to store public/private key pairs that the server can use for secure communications. The default value is WWW\BIN\keyfile.kyr.

Administrator ID
The ID of your server administrator. Anyone attempting to use the Internet Connection Server Configuration and Administration Forms will be prompted to enter this ID. The default is webadmin.

Password
The password you want to use to protect access to the Configuration and Administration Forms. Anyone attempting to use the Internet Connection Server Configuration and Administration forms will be prompted to enter this password. The default is webibm.

Auto Start Server at Bootup
If you want the Internet Connection Server to start automatically when you start your host machine, mark this check box. The server will then be added to your OS/2 Startup folder.

Step 6. View the target directory assignments:

The target directories cannot be changed. You may want to make a note of the information in this window because it shows where the server will look for certain files when it is running.

Configuration file
This is the file that contains the Internet Connection Server configuration settings. The file is named httpd.cnf and is put in the path specified on the SET ETC statement in your CONFIG.SYS file.

CGI scripts
This is the same directory you specified for your CGI script programs on the Install - directories window.

HTML documents
This is the same directory you specified for your HTML documents on the Install - directories window.

Config password file
This is the server password file that will contain the values you entered above in the Administrator ID and Password fields. The file is named ADMIN.PWD and is put in the path specified on the SET ETC statement in your CONFIG.SYS file.

Step 7. Complete the installation:

a. From the ICSS Install Configuration window, click on **OK**. A window will inform you how the installation progresses.

b. From the Internet Connection Server Installation window, click on **Exit** to complete installation.

If the installation procedure updated your CONFIG.SYS file, you will be prompted to reboot your system before starting the server.

If you checked Auto Start Server at Bootup, the server will be placed in your OS/2 Startup folder and will start automatically each time you start or reboot your system.

Step 8. Connect to your server:

Use your favorite browser to connect to your server's Front Page by typing the URL `http://your.server.name`, where your.server.name is the fully qualified name of your host. In our project we used:

`http://mqjava/itso.ral.ibm.com`

The front page contains links that let you:

- Access the configuration and administration forms.
- Create sample home pages.
- Access this Web site (Internet Connection Family).
- Read online Internet Connection Server documentation.

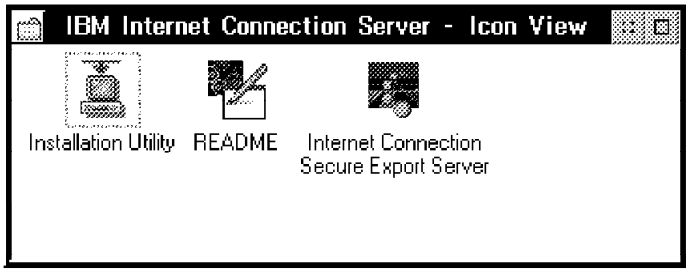


Figure 7. Internet Connection Server - Icon View

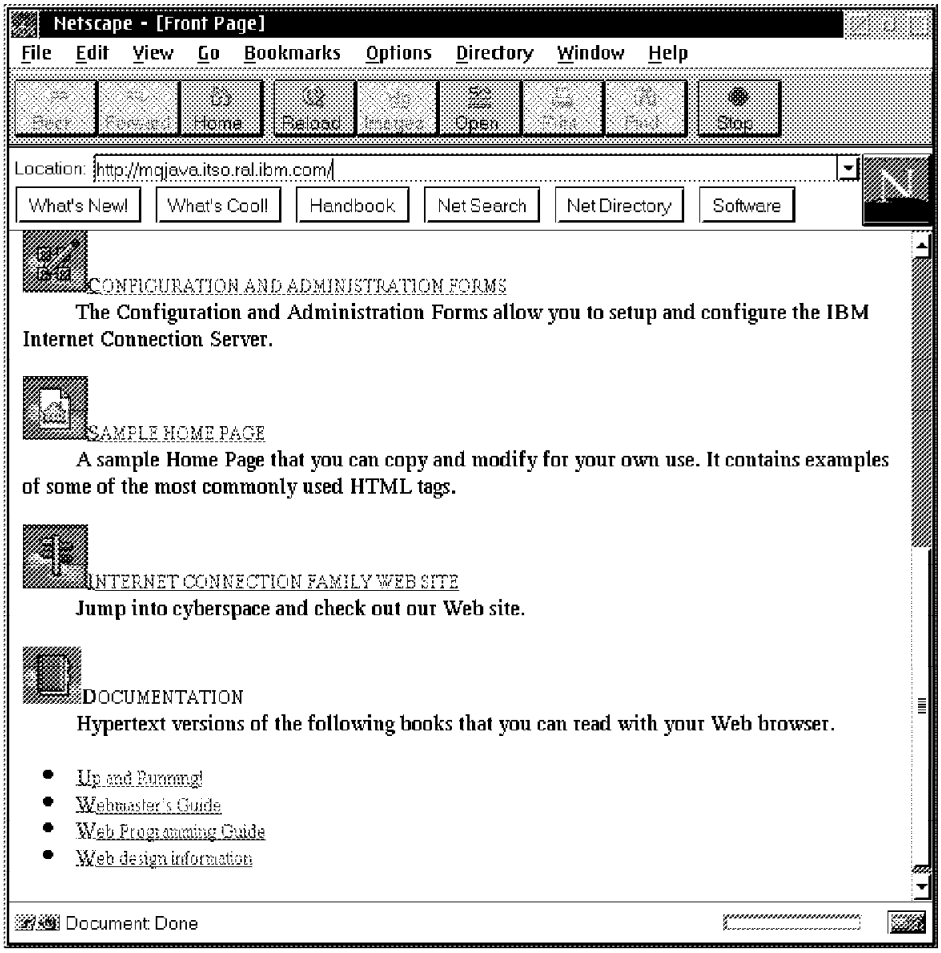


Figure 8. Example of a Server Front Page

3.2 Netscape Navigator for OS/2

If you use Java, you need a Java-enabled browser. For our project, we used the Netscape Navigator for OS/2 Version 2.02, first the Beta versions and then the GA (general availability) version.

You can download this native OS/2 version of the Netscape Navigator client software at no additional charge to OS/2 Warp users. With the Netscape Navigator for OS/2 you receive Netscape′s HTML-enabled mail and news applications as well as support for plug-in extensions and frames. You will also be able to take advantage of JavaScript, incorporated with Netscape

Navigator, which extends HTML capabilities and allows you to open and view live Java applets and applications.

3.2.1 How to Get the Browser

Download the Netscape Navigator for OS/2 Warp from:

`http://www.internet.ibm.com/browsers/netscape/warp`

You receive a file of about 4.5 MB: nsos2202.exe.

Note: You need OS/2 Warp Version 4 for Java programs and applets.

3.2.2 How to Install the Browser

Create a directory and place the files into it using the following commands:

```
c:\>md netscape
c:\>cd netscape
c:\netscape>c:\nsos202
c:\netscape>install
```

Note: Read the read.me file in the netscape directory.

Click on **OK** in the Install window. The CONFIG.SYS will automatically be updated.

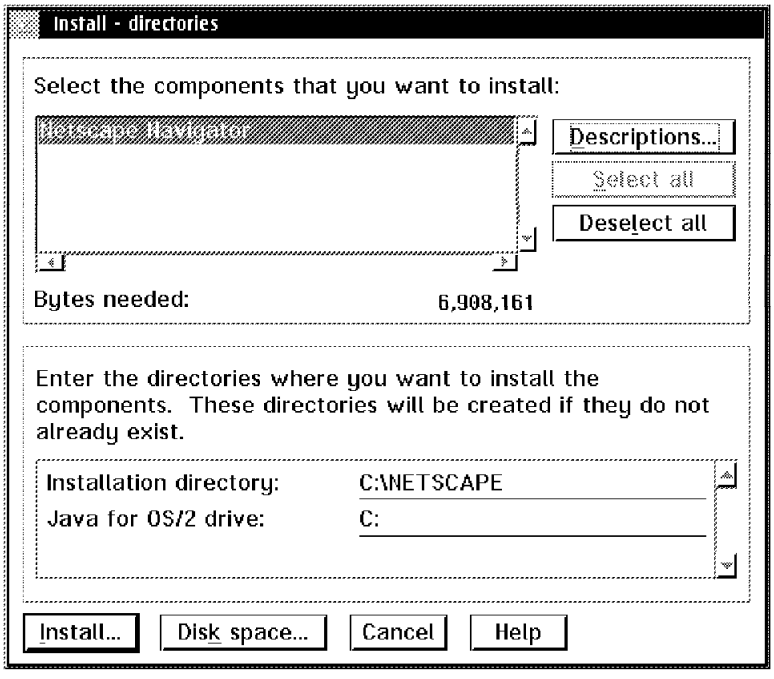


Figure 9. Netscape Navigator Install Window

Click on **Install** if you accept the drive and directory. A window keeps you informed about the progress of the installation.

When the installation is completed click on **Exit**. The installation creates the following folder on your desktop:

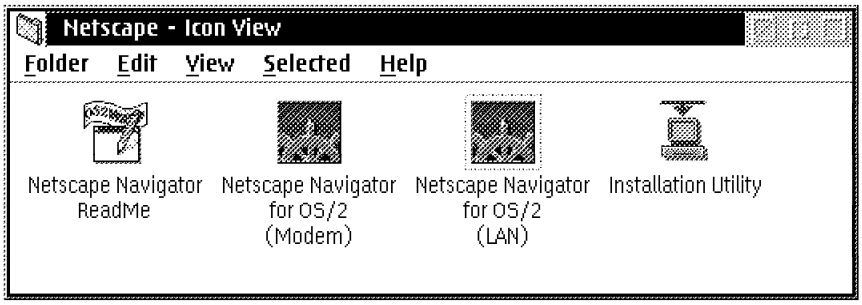


Figure 10. Netscape Navigator Icon View

3.2.3 How to Set Up the Browser for Java

Bring up the Netscape Navigator, select **Network Preferences** from the Options menu and skip to the third notebook page, Proxies. There, select **Manual Proxy Configuration** and then click on **View**.

This brings up a window in which you type in your SOCKS host. Every region has its own socks server. We use socks.raleigh.ibm.com. Click on **OK** to return to the browser's window.

Since we want to use Java we have to ensure that Java is enabled. Select **Security Preferences** from the Options menu. In the subsequent window remove the check mark from Disable Java then close the window.

You have to include the name of the directory that contains your Java applets in the CLASSPATH statement or applets will not run. Modify the CLASSPATH statement in the CONFIG.SYS like this:

```
SET CLASSPATH=C:\NETSCAPE\njclass.zip;C:\JAVAOS2\lib\jempcl10.zip;c:\myjava;.\.;
```

3.3 MQSeries Client for Java

The MQSeries Client for Java is available as SupportPac MA83 from IBM Hursley. The product enables Web browsers and Java applets 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 on the client machine.

This section describes how to obtain and install the MQSeries Client for Java. For this project we used the software from the Beta Test Program.

3.3.1 How to Get the MQSeries Client for Java

You can obtain a copy of the MQSeries for Client SupportPac MA83 from the Web page of IBM Hursley:

```
http://www.hursley.ibm.com/mqseries/txppacs/ma83.html
```

To find out about other SupportPacs link to:

```
http://www.hursley.ibm.com/mqseries/txppacs/txpsumm.html
```

If you work for IBM, you may obtain the product by entering the following command on the command line of your VM system:

```
TOOLS SENDTO WINVMB TOOLS TXPPACS GET MA83 PACKAGE
```

If you agree with the IBM International Program License Agreement, you will receive the file MA83.ZIP (about 500 K in size).

The code is provided in InfoZip compressed format. To unzip these files you need a program that understands long file names and can recreate subdirectories. InfoZip does this. You can download it in the form of a self-extracting executable from the same Web page as the product. The file name for OS/2 is UNZ520X2.EXE. File names for other platforms may vary.

Note: This is a self-extracting EXE. Execute UNZ520X2 (if you downloaded the OS/2 version). This will create the file UNZIP.EXE (and several other files). Use UNZIP.EXE to unzip MA83.ZIP.

```
unzip MA83
```

This creates the following files:

mqc4j.zip	The executables for the client
mqc4jdoc.zip	The documentation in HTML format
ipla	IBM International Program License Agreement
li	License Agreement in different languages
readme.txt	Notes for the installation
Relnotes.txt	Release notes for the SDK version of the product (2.0)

Use the UNZIP.EXE again to unzip the documentation and the executables for the client.

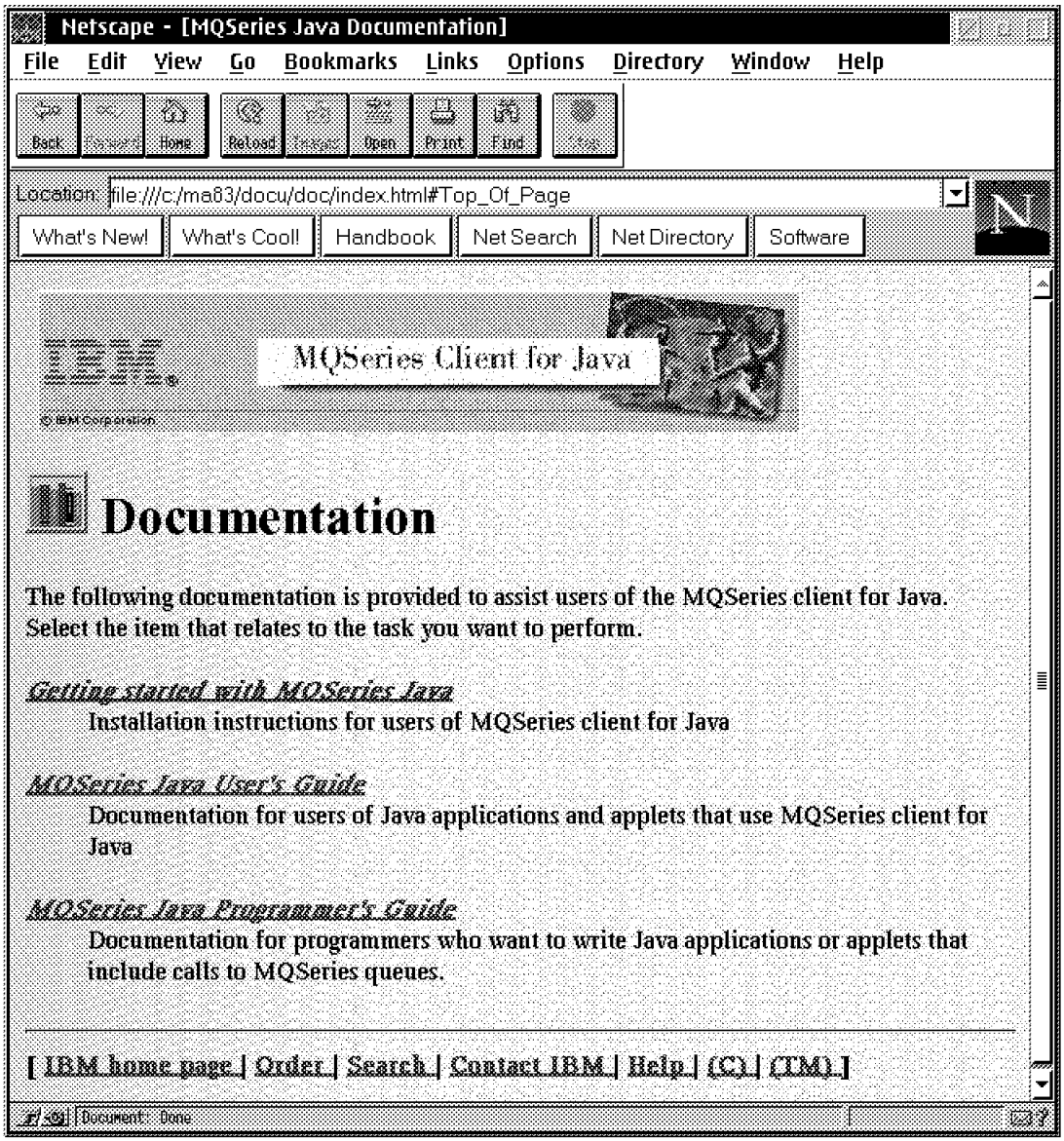


Figure 11. MQSeries Client for Java Documentation (index.html)

3.3.2 Installing the Documentation

Unzip the documentation file (mqc4jdoc.zip) into a directory that is accessible to your Web browser and is on a drive that is capable of supporting long file names. You can use any unzip utility that can handle

long file names and recreate subdirectories. Examples are WinZip and Info-Zip's UnZip. (You cannot use PKUNZIP.)

Note: The *MQSeries Client for Java Programmers Guide* contains links to the Java SDK documentation. For these links to be enabled, the MQSeries Client for Java documentation should be installed in the same directory as the Java SDK documentation.

Open the index.html file in your Web browser and click on the **Getting started with MQSeries Java** link to reach the installation instructions. You find the index.html file in the subdirectory \doc of the directory where you installed the documentation. Figure 11 on page 21 shows the page index.html.

3.3.3 Installing the MQSeries Client for Java

MQSeries Client for Java can be installed either on your local hard disk or on a Web server. Installation on a Web server has the advantage of allowing you to download and run MQSeries client applications on machines that do not have the MQSeries Client for Java installed locally.

The MQSeries Client for Java can be run in three different modes:

- From within any Java-enabled browser
- Using an applet viewer
- As a stand-alone Java program

When running from within a browser, the location of the queue managers that can be accessed may be constrained by the security restrictions of the browser being used.

If you use an applet viewer or write stand-alone programs, you must have the Java Developer's Kit (JDK) installed on the client machine.

The installation depends on the mode you want to use it in.

3.3.3.1 Using a Web Browser

If you intend to run the client from within a Web browser, you must either:

- Unzip the file on your Web server machine into a directory that is accessible from your Web browser.
- Unzip the file into a directory on your local machine.

When the mqc4j.zip file is unzipped, some files are placed into the current directory and some files into a subdirectory called com\ibm\mq.

- The files in the root directory are for the installation verification applet described in the next section.
- The files in the MQ subdirectory constitute the MQSeries Client for Java.

3.3.3.2 Applet Class Directory

When accessing the client from a Web browser, the applet is run from within an HTML file. This file contains an applet tag that specifies the class to be loaded and run. If this applet class is stored in a directory X on the Web server machine, the Java client files must be stored in a directory x\com\ibm\mq on the same machine.

3.3.3.3 Using the Applet Viewer

If you intend to run the client from an applet viewer, you must have the Java run-time environment installed on your machine.

Unzip the file mqc4j.zip to either:

- Your Web server machine into a directory that is accessible from your Web browser.
- Into a directory on your local machine.

If you have installed the client on your local machine, you must ensure that the directory in which you unzipped mqc4j.zip is in your CLASSPATH environment variable.

3.3.3.4 Setting the CLASSPATH Variable

The Java run-time environment uses a variable called CLASSPATH to locate the files in the Java client. If this variable is already set on your machine, you have two choices:

1. Unzip the mqc4j.zip file into a directory that is in your CLASSPATH.
2. Add the directory in which you unzipped the file to the end of your CLASSPATH statement.

Examples:

- On AIX use the command:

  ```
  export CLASSPATH=$CLASSPATH:newdir
  ```

- On DOS, OS/2, or Windows use the command:

  ```
  set CLASSPATH=%CLASSPATH%;newdir
  ```

Where newdir is the directory in which you have installed the Java client files.

If the CLASSPATH variable is not set, make it point to the directory in which you unzipped mqc4j.zip and also to the directory that contains lib\classes.zip from your JDK installation.

3.3.4 Running the Installation Verification Applet

A simple installation verification program is provided as part of the MQSeries client for Java in a file called mqjavac.htm.

The program connects to a given queue manager, executes all the MQ calls, and produces simple diagnostic messages in the event of any failures.

You find a detailed description in section 7.3, “Running the Installation Verification Program” on page 77 and an example in section 6.4.4, “How to Test a Client/Server Connection with MQSeries for Java” on page 66.

3.4 Apache Internet Server for AIX

On the RS/6000, we used the Apache Server Version 1.1.1. This server provides a reliable service on our AIX machine.

3.4.1 How to Get Apache

You find this product and can download it from the Web page:

```
http://www.apache.org
```

3.4.2 Notes Regarding Apache

1. Run the configuration script located at /apache_1.11.1/src. The name of it is configure.

   Before you run the shell script, look in Configuration.tmpl for the line CC=cc and any other lines related to the UNIX you are using, in our case AIX/6000. There may be some differences regarding the C compiler.

   Look closely at the last line of the shell script:

   ```
   cat Makefile.tmpl >> Makefile
   ```

   Therefore, also look at the Makefile.tmpl to ensure that all settings for the AIX/6000 compiler are there.

2. Compile the Apache.

   Before compiling look into Makefile.tmpl, located under /apache_1.1.1/src. Make sure the Configure shell did everything correctly according to the changes made in the Configuration.tmpl file. You are now ready to compile. You may see some warnings that you

can ignore. They are due to extent modules that may not have been downloaded or used.

3. Once compiled, the httpd is ready to run. Before running the daemon, remember to look into the directory /apache_1.1.1/conf and make the required changes to the httpd.conf, srm.conf, and any other related configuration file you may be using. The changes you have to make are related to the path where the files reside.

4. Ensure that the /etc/services file has the proper settings for port 80. This is the Internet services port that is most widely used.

```
www       80/tcp      # World Wide Web HTTP
www       80/udp      # World Wide Web HTTP
```

   Remember to run the inetd daemon by executing:

```
inetimp
refresh -sinetd
```

5. Now you are ready to start the httpd daemon. You can do this either manually or automatically.

   - Manually:

```
/usr/local/etc/apache/src/httpd -f /usr/local/etc/apache/conf/http.conf
```

   - Automatically:

     Create a file with the name rc.http in /etc with the following lines:

```
#! /bin/ksh
/usr/local/etc/apache/src/httpd -f /usr/local/etc/apache/conf/http.conf
```

     Once the shell is created, declare it as executable with:

```
chmod +x rc.http
```

     Now the inittab file in /etc has to be modified. After the rcnfs daemon line, add the following one:

```
rchttp:2:wait:/etc/rc.http > /dev/console/2>1  # Start Apache HTTP daemon
```

     Once this is done, refresh init by issuing an init q command.

Chapter 4. HTML Overview

This chapter introduces you to the HyperText Markup Language (HTML). For those of you who work with script, the General Markup Language (GML), BookMaster or Book Manager will find many similarities. Most tags are identical, only the delimiters changed to left and right arrows. Here are some examples:

Tag	HTML	Script/GML
Paragraph	<P>	:p.
Line break	 	.br;
Begin ordered list	<OL>	:ol.
End ordered list	</OL>	:eol.

On the following pages, we explain the basics of HTML and the HTML files for the demonstration program developed in this project:

- An HTML file that explains the important tags.

  After you review this section you should have a good understanding of HTML and be able to write your own WWW application.

  Note: There are many books available that cover HTML in depth.

- The home page created for this project. From this page you can link to:
  - An applet provided with MQSeries for Java, MQTest.class, that tests the client/server connection.
  - The application developed in this project, an applet that demonstrates how an order entry system for the Internet can be written.

    This application has been developed on a Windows NT system. You can find a description in Chapter 9, “The Sample Application” on page 105.

Note: As a browser we used the Netscape Navigator Version 2.02.

What this chapter is not

This chapter does not cover all of the capabilities of HTML. It provides an overview and discusses some important features of this language and enables you to understand how a WWW page is written.

4.1 How to Build an HTML File

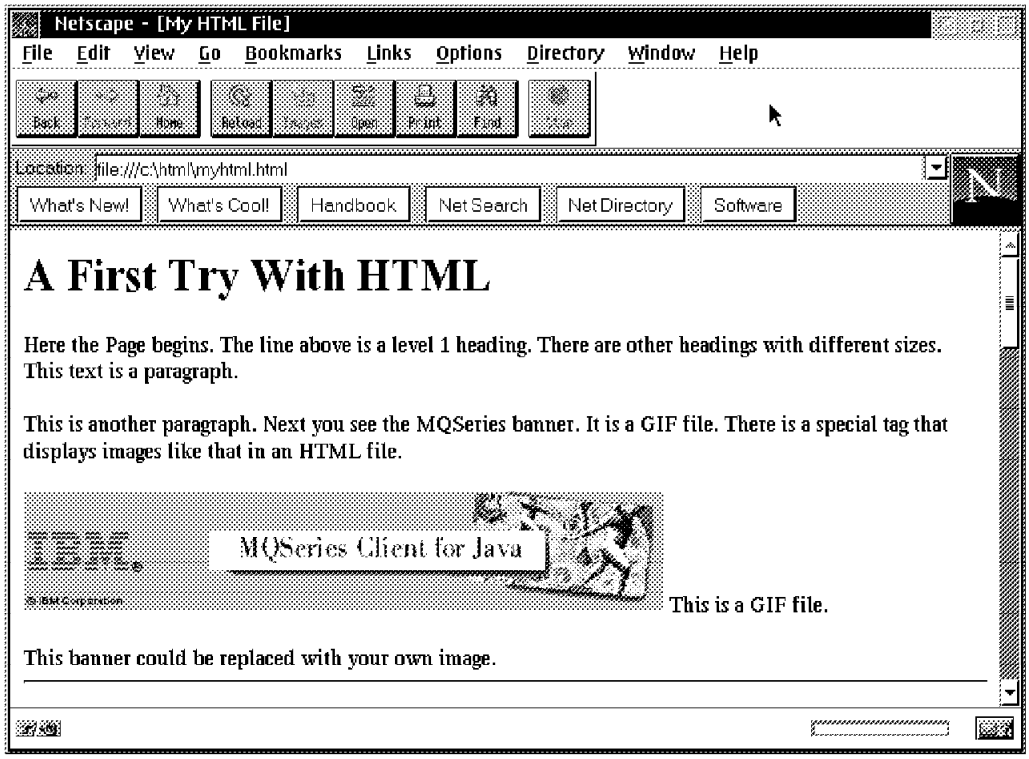


Figure 12. HTML Example - Part 1: An HTML Page

Each HTML file starts with <HTML> and ends with </HTML>.

An HTML file is divided into two parts, header and body. Usually, the header contains the title of the file. In the body you describe what appears in the browser. This is a skeleton of an HTML file:

```
<HTML>
<HEAD>
<TITLE>My HTML File</TITLE>
</HEAD>
<BODY>
⋮
</BODY>
</HTML>
```

1 This indicates the beginning of the HTML file.

```
1 <HTML>
2 <HEAD>
3 <TITLE>My HTML File</TITLE>
  </HEAD>
  <BODY>
4 <H1>A First Try With HTML</H1>
5 <p>
  Here the Page begins.
  The line above is a level 1 heading.
  There are other headers with different sizes.
  This text is a paragraph.
  <p>This is another paragraph.
  Next you see the MQSeries banner.  It is a GIF file. There is a special
  tag that displays images like that in an HTML file.
  <p>
6 <IMG SRC="MQJCBAN.GIF">
7 This is a GIF file.
  <p>
  This banner could be replaced with your own image.
8 <HR>
```

Figure 13. HTML File - Part 1

2 The header is between the <HEAD> and </HEAD> tags.

3 The title appears in the header line of the browser's window as shown in Figure 12 on page 28.

Note: The header can contain other tags to describe the document itself.

4 The first line in the body of the HTML file is a heading. There are headings of different levels. This is a level 1 heading.

5 This tag indicates the begin of a paragraph. All following text belongs to that paragraph. You may end the paragraph with the </P> tag or simply start the next paragraph with another <P> tag.

6 This tag displays the image file MQJCBAN.GIF. The SRC attribute specifies the name of the image file to be displayed.

Note: The image file resides in the same directory as the HTML file.

7 Since the image is not followed by a <p> tag this text appears to the right of the image.

8 The <HR> tag draws a horizontal rule.

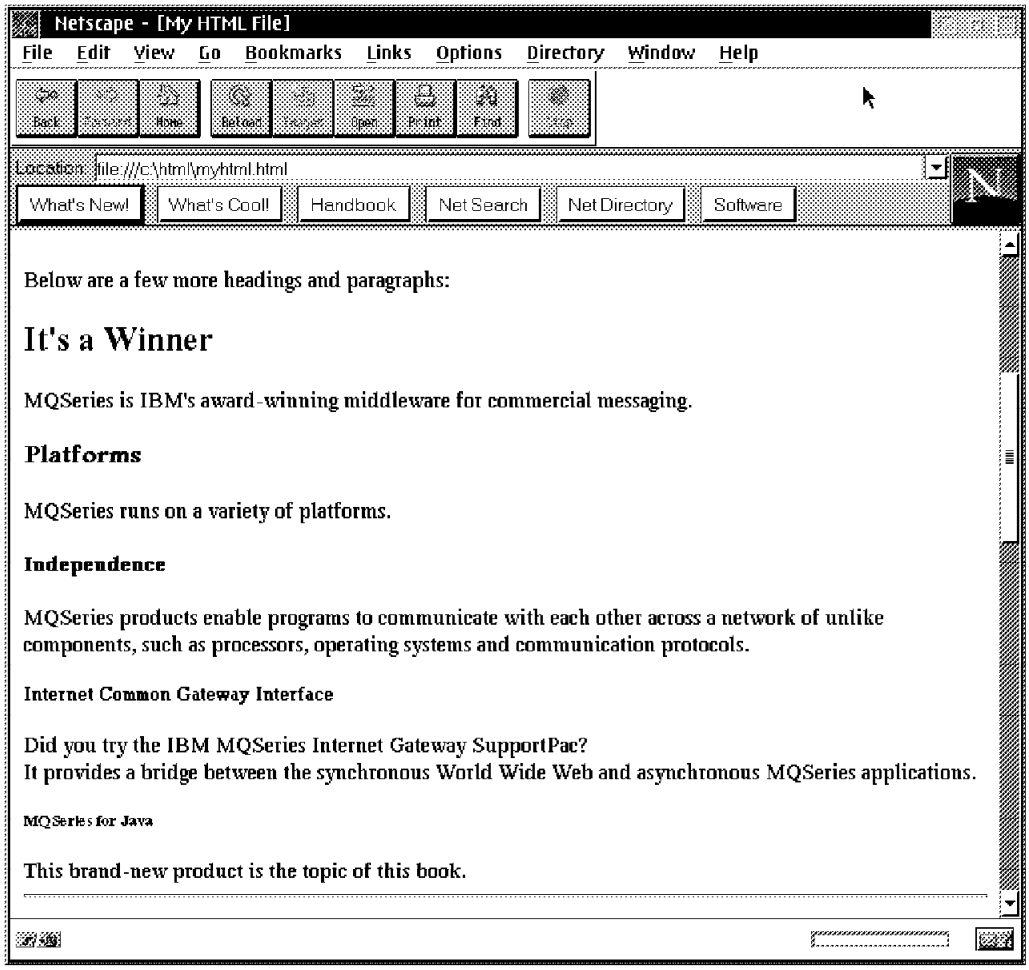


Figure 14. HTML Example - Part 2: Headings

There are six levels of headings in HTML. A level 1 heading is shown in Figure 12 on page 28. The window above shows the other five.

9 This is a level 2 heading.

Note that for the paragraph following the header the tag and the text are on separate lines.

10 This is a level 3 heading.

For this paragraph, the <p> tag and text are on the same line.

```
    <p>Below are a few more headings and paragraphs:
 9  <H2>It's a Winner</H2>
    <P>
    MQSeries is IBM's award-winning middleware for commercial messaging
10  <H3>Platforms</H3>
    <p>MQSeries runs on a variety of platforms.
11  <H4>Independence</H4>
    MQSeries products enable programs to communicate with each other
    across a network of unlike components, such as processors, operating
    systems and communication protocols.
12  <H5>Internet Common Gateway Interface</H5>
    <p>Did you try the IBM MQSeries Internet Gateway SupportPac?
13  <br>It provides a bridge between the synchronous World Wide
    Web and asynchronous MQSeries applications.
14  <H6>MQSeries for Java</H6>
    <p>This brand-new product is the topic of this book.
15  <HR>
16  <! ------------------------------------------------------------>
```

Figure 15. HTML File - Part 2

11 This is a level 4 heading.

This paragraph has no <p> tag. HTML assumes that text following a heading end tag, here </H4>, constitutes a paragraph.

12 This is a level 5 heading.

13 The
 tag causes a line break. Two consecutive tags are equivalent to <p>.

14 This is a level 6 heading.

15 This tag causes the horizontal rule.

16 This is a comment. You may write any text between <! and >.

Note: How the headers actually look depends on the browser you are using. Figure 14 on page 30 shows how they appear in the Netscape Navigator.

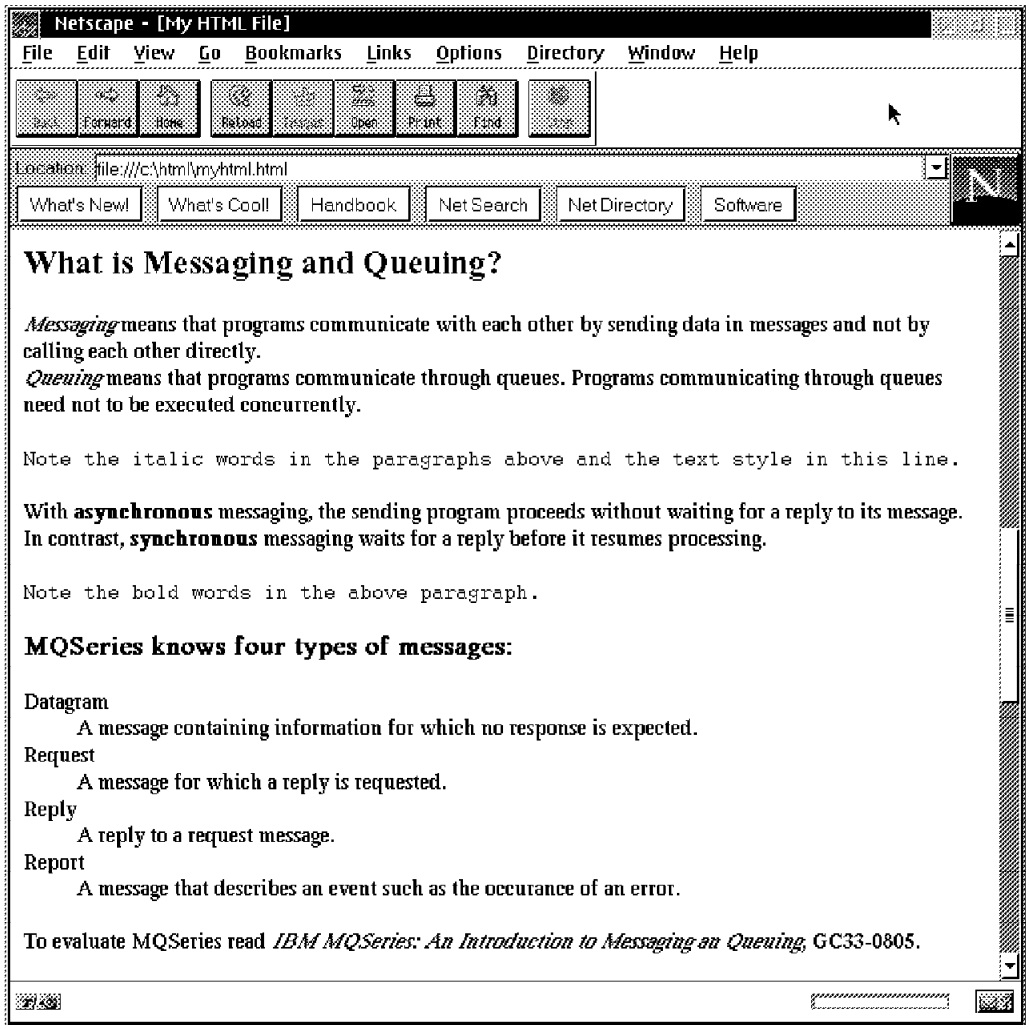


Figure 16. HTML Example - Part 3: Emphasizing

17 Note that the first word in the paragraph is in italics (<i>).

18 Emphasized words (<EMP>) are usually in italics.

19 Usually, the <KBD> tag is used to indicates text to by typed.

20 Stronger emphasized words (<STRONG>) are usually displayed in bold (<B>).

21 The <CODE> tag is used to indicate code samples.

```
    <h2>What is Messaging and Queuing?</h2>
17  <i>Messaging</i> means that programs communicate with each
    other by sending data in messages and not by calling each other directly.
    <br>
18  <EM>Queuing</EM> means that programs communicate through queues.
    Programs communicating through queues need not to be executed concurrently.
    <p>
19  <KBD>Note the italicized words in the paragraphs above
    and the text style in this line.</KBD>
    <p>
20  With <STRONG>asynchronous</STRONG> messaging, the sending
    program proceeds without waiting for a reply to its message.
    In contrast, <B>synchronous</B> messaging waits for a reply
    before it resumes processing.
    <P>
21  <CODE>Note the bold words in the above paragraph.</CODE>
    <p>
    <h3>MQSeries knows four types of messages:</h3>
22  <DL>
    <DT>Datagram
    <DD>A message containing information for which no response is expected.
    <DT>Request
    <DD>A message for which a reply is requested.
    <DT>Reply
    <DD>A reply to a request message.
    <DT>Report
    <DD>A message that describes an event such as the occurrence of an error.
    </DL>
    <p>
    To evaluate MQSeries read
23  <CITE>IBM MQSeries: An Introduction to Messaging an Queuing</CITE>, GC33-0805.
```

Figure 17. HTML File - Part 3

22 A definition list (<DL>) starts here. A DL contains terms (<DT>) and definitions (<DD>) that explain the term:

- <DL> indicates the beginning of the list.
- <DT> is for the term you want to define.
- <DD> is the definition of the term.
- </DL> indicates the end of the list.

23 The <CITE> tag is used to highlight a quote or citation.

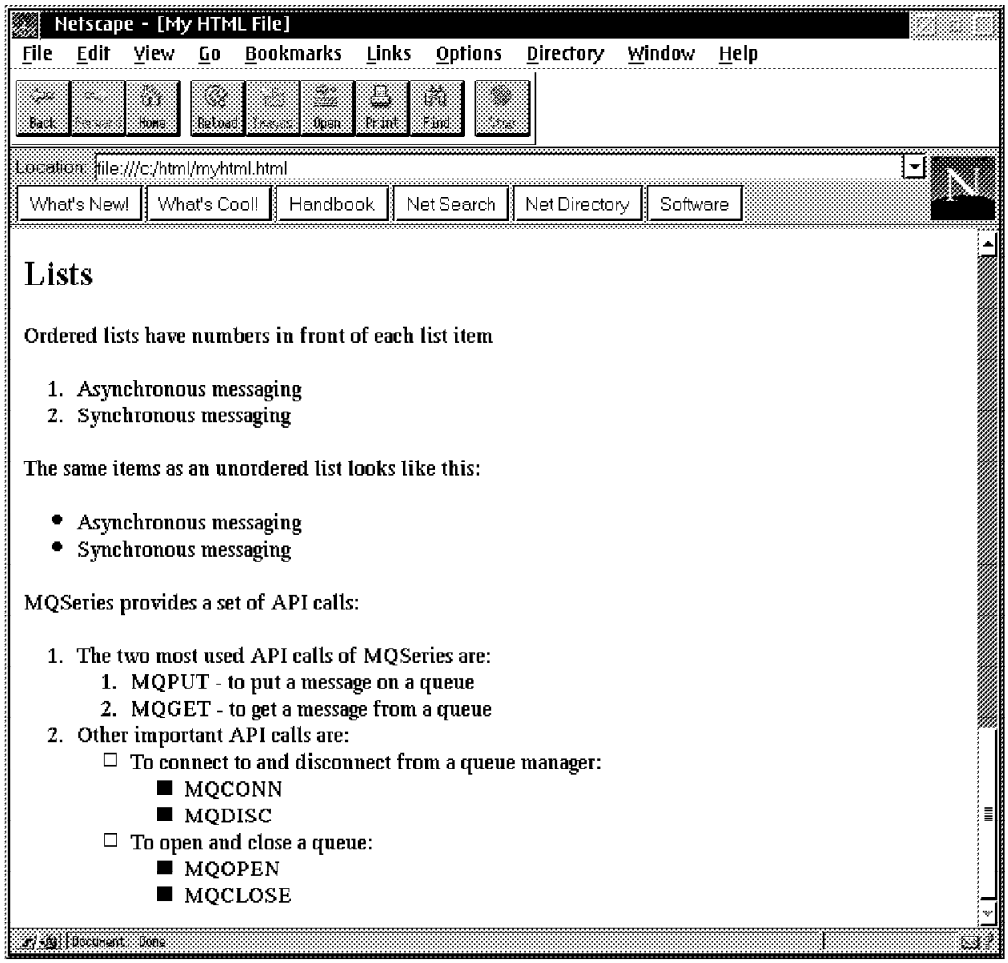


Figure 18. HTML Example - Part 4: Lists

In an HTML file you can build different kind of lists. Besides the definition list seen on the previous page you may display ordered and unordered lists.

24 An ordered list begins with <ol> and ends with .

25 An unordered list begins with <ul> and ends with .

Between those tags are list items, each beginning with a <li> tag. A list item can consist of one or more lines or multiple paragraphs.

Lists can also be nested.

```
    <h2>Lists</h2>
    <p>Ordered lists have numbers in front of each list item
24  <OL>
    <LI>Asynchronous messaging
    <LI>Synchronous messaging
    </OL>
    <p>The same items as an unordered list looks like this:
25  <UL>
    <LI>Asynchronous messaging
    <LI>Synchronous messaging
    </UL>
    <p>MQSeries provides a set of API calls:
26  <ol>
    <li>The two most used API calls of MQSeries are:
27  <ol>
    <li>MQPUT - to put a message on a queue
    <li>MQGET - to get a message from a queue
    </ol>
    <li>Other important API calls are:
28  <ul>
    <li>To connect to and disconnect from a queue manager:
29  <ul>
    <li>MQCONN
    <li>MQDISC
    </ul>
    <li>To open and close a queue:
30  <ul>
    <li>MQOPEN
    <li>MQCLOSE
    </ul>
    </ul>
    </eol>
```

Figure 19. HTML File - Part 4

26 Inside this ordered list is:

Another ordered list (**27**).

An unordered list (**28**).

28 This nested list contains two more lists (**29** and **30**).

Note: You can use <LI> tags outside a list. The browser starts a new line and places a bullet in front of the text.

4.2 How to Build a Link

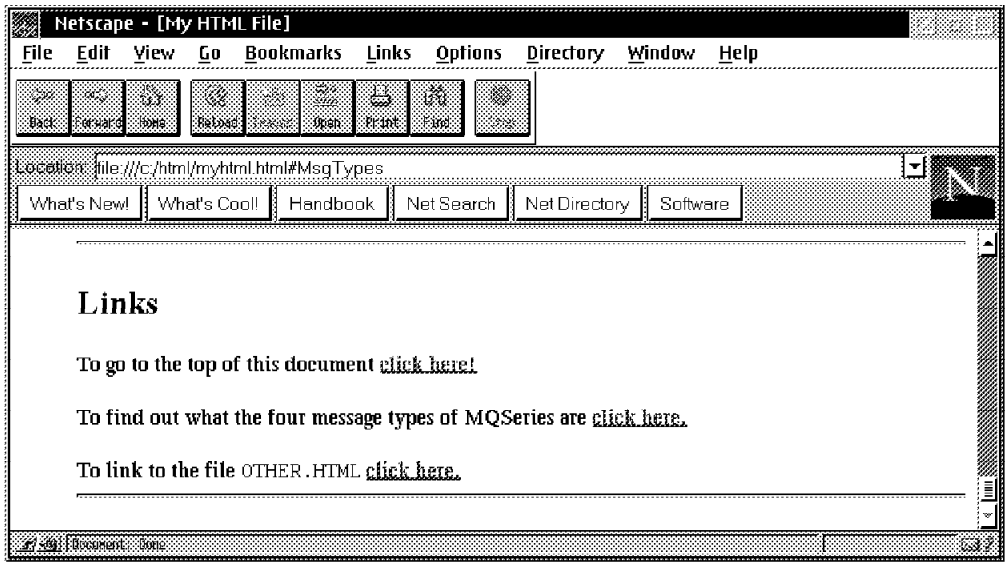


Figure 20. HTML Example - Part 5: Links

You can link from an HTML file to:

- Another HTML file
- A specific place inside another HTML file
- A specific place in the same HTML file

To jump to a specific place in an HTML file you have to define an *anchor*. The anchor is like a label you jump to. It determines what line appears on the top of the window when the browser executes a link to that anchor.

31 We added the following anchor to the header for the definition list (**22** on page 33):

```
<h3> ... <A NAME=MsgTypes>   </A> </h3>
```

This tag defines an anchor with the name MsgTypes.

Note: The name is part of the <A> tag!

You link to this anchor with a tag like this:

```
<A HREF=#MsgTypes> click here.</A>
```

The words click here are usually displayed in a different color than the rest of the text. The browser loads the file and searches for the anchor MsgTypes.

```
⋮
31 <h3>MQSeries knows four types of messages:<A NAME=MsgTypes> </A> </h3>
⋮
    <p>
32  To go to the top of this document <A HREF=myhtml.html> click here!</A>
    <p>
33  To find out what the four message types of MQSeries are
    <A HREF=#MsgTypes> click here.</A>
    <p>
34  To link to the file <KBD>OTHER.HTML</KBD>
    <A HREF="OTHER.HTML">click here.</A>
    <HR>
    </BODY>
    </HTML>
```

Figure 21. HTML File - Part 5

The next three statements show how to define links to a file and to an anchor within a file.

32 If you want to go to the top of the current HTML file you can either use the scroll bar or define a link and click on it.

The link `<A HREF=myhtml.html> click here!</A>` consists of two parts:

1. Inside the < A > tag is the name of the file you want to link to. In this case it is the current HTML file.
2. The text click here! between < A > and < / A > is displayed underlined and blue. When you click on this text the browser will execute the link.

33 The link `<A HREF=#MsgTypes> click here.</A>` does not contain a file name but the anchor name specified in **31**. The name is preceded by an # sign. This tells the browser to display the same file starting with the line that includes the anchor. That's how you jump to a specific place within a file.

34 The link `<A HREF="OTHER.HTML">click here.</A>` points to a different HTML file with the name OTHER.HTML. If you click on the text click here, the browser will display the file shown on page 38.

35 This link displays the file myhtml.html again (page 38).

35 This link contains the a file name and an anchor (page 38). This is how you jump to a specific place in another HTML file.

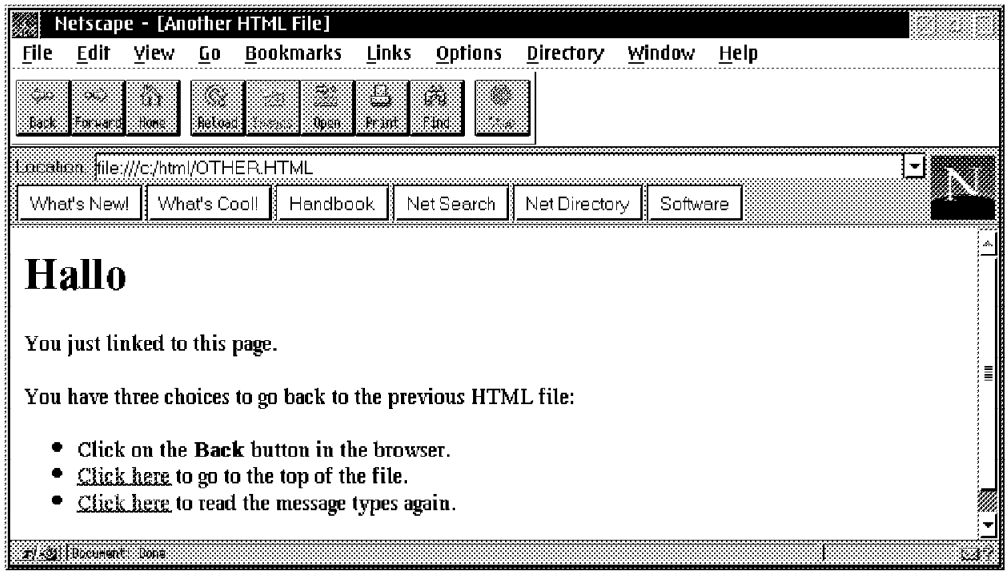


Figure 22. HTML Example - Part 6: More Links

```
    <HTML>
    <HEAD>
    <TITLE>Another HTML File</TITLE>
    </HEAD>
    <BODY>
    <H1>Hallo</H1>
    <p>
    You just linked to this page.
    <p>
    You have three choices to go back to the previous HTML file:
    <ul>
    <li>Click on the <B>Back</B> button in the browser.
35  <li><A HREF="myhtml.html">Click here</A> to go to the top of the file.
36  <li><A HREF="myhtml.html#MsgTypes">Click here</A> to read the message types again.
    </ul>
    </BODY>
    </HTML>
```

Figure 23. HTML Example - Part 6

4.3 How to Load an Applet

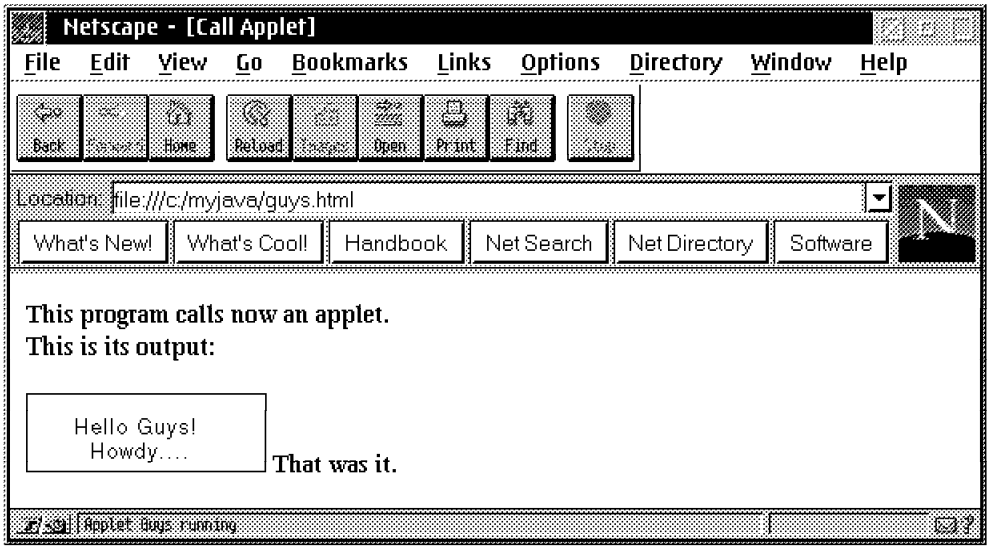


Figure 24. HTML Displays an Applet

```
<HTML>
<HEAD>
<TITLE> Call Applet </TITLE>
</HEAD>
<BODY>
This program calls now an applet.
<br>
This is its output:
<p>
<APPLET CODE="Guys.class" WIDTH=150 HEIGHT=100">
Attention:
<br>
This browser does not know applets.
<p>
</APPLET>
That was it.
</BODY>
</HTML>
```

Figure 25. HTML Example - Display an Applet

Note: Only Java-enabled browsers can load applets.

Figure 24 shows a box with Hello Guys and Howdy in it. The box is displayed by an applet with the name Guys.class. You include Java applets in a Web page with the <APPLET> tag. The tag requires that you specify some attributes:

- The attribute CODE specifies the name of the compiled Java code, here Guys.class.
- WIDTH and HEIGHT of the applet in pixels.

The HTML in Figure 25 on page 39 contains some text between the <APPLET> and </APPLET> tags. This text is displayed, instead of the applet, if the HTML file is loaded by a browser that is not Java-enabled.

Figure 2 on page 6 shows three clocks. The applet was written by Nizze and downloaded from the Web site of the Computer Science Department of the Linköping University, Sweden. To define and set the clock you have to hand over to the applet parameters. This is done with the <PARAM> tag. Here is an example:

```
<applet  code="Clock.class" width=200 height=200>
<param name=num_lines value=3>
<param name=hour_len value=60>
<param name=minute_len value=75>
<param name=second_len value=95>
<param name=hour_col value=ff0000 >
<param name=minute_col value=00ff00>
<param name=second_col value=0000ff>
<param name=border_col value=00ffff>
<param name=background_col value=cccc22>
<param name=timezone value=1>
<p>
You do not see the clock because this browser is not Java-enabled!
</applet>
```

An introduction to Java can be found in Chapter 5, “Java Overview” on page 41.

More about HTML

For detailed information about HTML refer to publications widely available in bookstores.

Chapter 5. Java Overview

This chapter is a brief introduction into Java and is intended for those readers who have no experience with this enveloping programming language.

5.1 Some Basics about Java

The following shows what Java is and why you should learn Java or use Java with your browser:

- Java is a programming language, developed by Sun Microsystems.
- Java is an object-oriented programming language.
- Java was modeled after Smalltalk and C++.
- Java is easier than C++.
- Java is platform-independent.
- Java is operating system-independent.
- Java is an interpreter language.
- Java is a secure language.
- Java is distributed.
- Java is robust.
- Java is the ideal programming language for Internet applications.

5.1.1 Java Is Platform-Independent

You can write Java programs with an editor of your choice. The source code is plain ASCII code. This source code you can transfer to any system that can read ASCII code. Then you can compile this code using the Java compiler for that system. The compiler generates Java byte code. The byte code again can be transferred to any Java-enabled operating system to run it, or if it is an applet, to run it within a Java-enabled browser.

Figure 26 on page 42 shows how Java and C programs are compiled and distributed. It elucidates that the byte code generated by any Java compiler will run on any machine that supports Java.

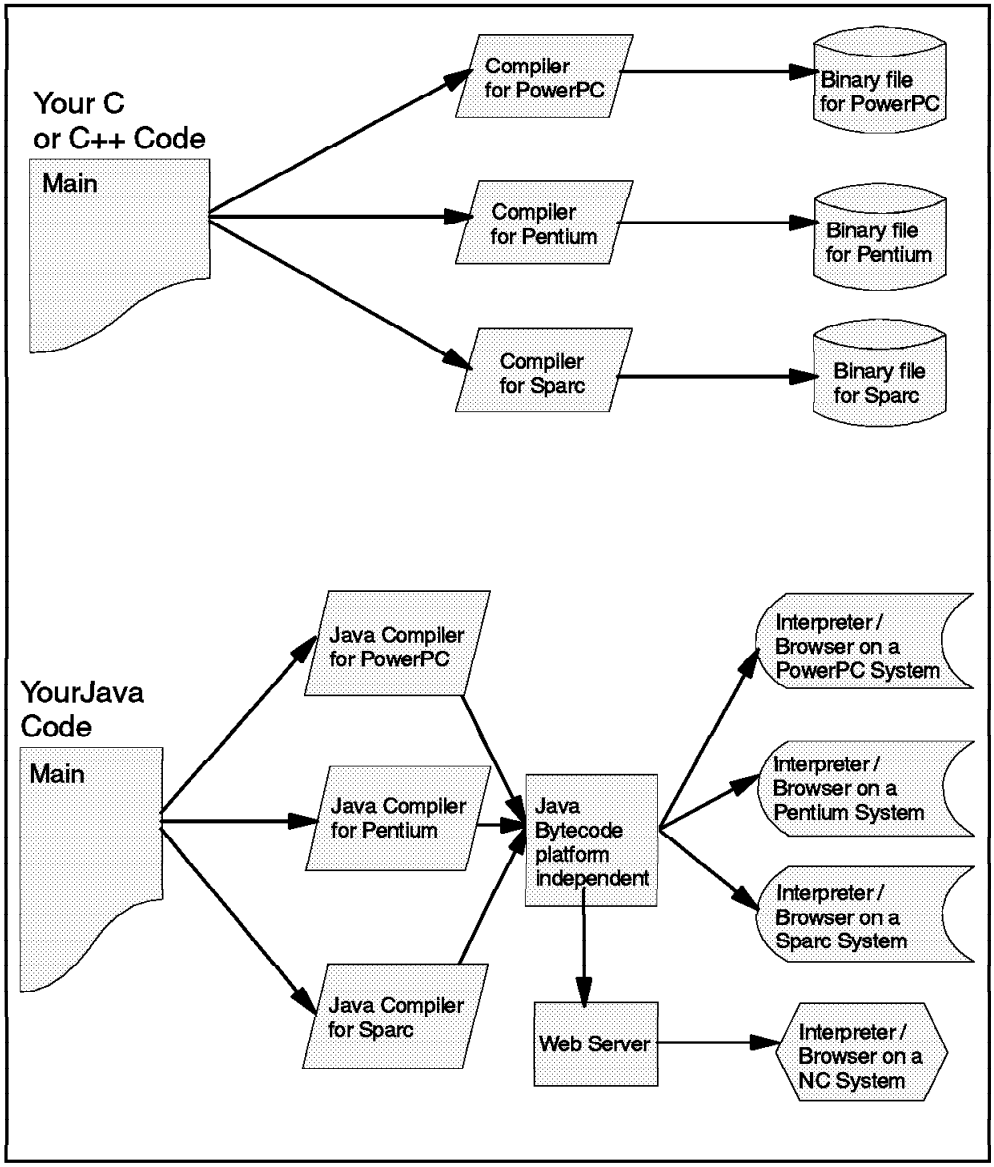


Figure 26. Difference between C And Java Compiler

5.1.2 Java Is Distributed

Java is inherently distributed. The Java class libraries contain a lot of routines for coping with TCP/IP protocols such as FTP and HTTP. Java programs can access URLs as easily as a file system.

In our case, writing an applet using the MQSeries client for Java. The user can download the Java byte code of our program from the Internet and run it on his or her own system. That means that anyone with access to your Web server can load and run your applet with no prior installation needed on his or her machine. When an update to the program is required, you simply update the applet on your Web server and the user automatically receives the latest version the next time he or she accesses the applet. This reduces cost for program service and updates significantly.

5.1.3 Java Is Secure

Java is intended to run in networked/distributed environments, and a lot of emphasis has been placed on security. Java programs cannot overrun their run-time stack, cannot corrupt memory outside of their process space, and when downloaded via the Internet, cannot even read or write local files. So Java is the ideal language for the upcoming network computer.

5.1.4 Java Is Robust

Java puts a lot of emphasis on early checking for possible problems, dynamic (run-time) checking, and the elimination of situations that are error prone. Java uses a concept of references that eliminates the possibility of overwriting memory and corrupting data.

The following features are what make it robust:

- Java eliminates pointer manipulation, so that the memory usage is encapsulated in classes specifically built for that purpose.
- Java maintains run-time integrity by ensuring that distribution and dynamic linking have not introduced errors into the code (in addition to type checking at compile time). The interpreter ensures that the byte code has not tempered with and that transmission errors are not modifying the code.
- Java eliminates the common problems of out-of-bounds array access attempts in C and C++. Java always catches accesses to invalid array elements. Some are caught at compile time, others at runtime when computing index values.
- Java supports multithreading by providing synchronization modifiers in the language. At the object level, threaded applications can inherit classes specifically created for that purpose. The priority of specific threads can be set by applications to suit specific needs, allowing unique modes of preemptive multitasking.

5.1.5 Java Is Object-Oriented

Like Smalltalk and C++, Java is an object-oriented programming language. However, it is not hybrid like C++. Java uses the concepts of classes and objects, instances, interfaces, methods, single inheritance, encapsulation and polymorphism. There are many books about object-oriented technology out in the world, so we are not going to explain what object-oriented technology means.

With Java you already get a lot of classes for the main functions you need. So it is easy to start writing your first application or applet. The following packages are included in Java:

- **Language Package (java.lang.class)**

  This package provides the elementary classes for strings, arrays and elementary data types.

- **Utility Package (java.util.class)**

  This package includes classes for the support of handling vectors, stacks, hash tables, encoding and decoding.

- **I/O Package (java.io.class)**

  With this package you get classes for standard input and output, as well as file I/O.

- **Applet Package (java.applet.class)**

  This package provides support to interact with the browser.

- **Abstract Window Toolkit (AWT) Package (java.awt.class)**

  This package was used mainly by us to build the GUI (graphical user interface). It provides support to control the visual aspects of your application or applet. Objects such as buttons, scroll bars, text fields, lists and fonts are available in this class.

- **Network Package (java.net.class)**

  For communication with other applications, this package provides the basic support to communicate with peer programs over the network, as well as standard protocols as TCP, FTP and URL access.

  Note: In our applet we use the MQSeries Client for Java classes instead of this network package.

Remember these features, because in the next section we develop a small application to become familiar with the real world of Java programming. We start with an application that we know from other languages, the Hello World application. We create and analyze the statements in the next section.

5.2 Applications and Applets

You can create two types of programs with Java: an application and an applet. The main difference between them is the way the program is run.

A Java application is a regular program like a C or C++ program with a main statement. However, unlike C or C++, a Java application requires an interpreter. OS/2 Warp Version 4.0 includes a Java interpreter called JAVA.EXE (or JAVAPM.EXE for programs that use the AWT classes).

A Java applet is a more restricted program. Because an applet is intended to be delivered over the Internet, it is small and does not have access to all the functions available to regular programs. This restriction gives a Java applet the security level required to avoid intentional data corruption and malicious programming, such as viruses.

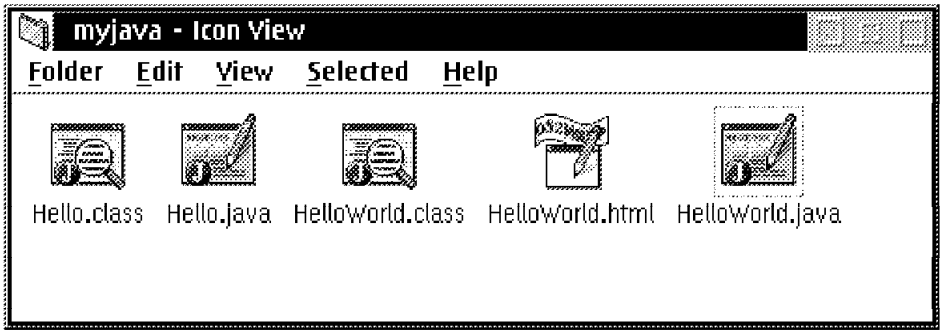
A Java applet is usually run by a Java-enabled browser, such as Netscape Navigator Version 2.02 or Microsoft Internet Explorer Version 3.0.

OS/2 Warp Version 4.0 includes an applet viewer (APPLET.EXE) that allows you to run a Java applet without a Java-enabled browser. This is very helpful when you are testing an applet. You will see this later, when we write an applet.

Once loaded, the program runs inside a Java virtual machine. The virtual machine is a controlled environment where the Java byte code is interpreted and translated into machine language.

5.3 A First Try with Java

In this section we create a small Java application and a small applet. For both application and applet, we at first create a directory structure, that holds our examples. After completing them your directory (folder) should look like this.



Let us create the application step by step.

First create a directory to store your HTML files and your Java applet and application files.

```
[C:\myjava]md myjava
[C:\myjava]cd myjava
```

Ensure that the name of the directory is in the CLASSPATH environment variable in the CONFIG.SYS:

```
SET CLASSPATH=C:\NETSCAPE\njclass.zip;C:\JAVAOS2\lib\jempcl10.zip;c:\myjava;.\.;
```

Now using your favorite editor, type the source code of your first Java application.

5.3.1 The Hello World Application

The following code demonstrates a simple Java application. The file extension of the source code is always .java, such as Hello.java.

```
class Hello {
      public static void main (String args[]) {
             System.out.println("Hello World!");
             System.out.println("Welcome to MQSeries client for Java.")
      }
}
```

Figure 27. A Simple Java Application

Now let us analyze the source code of Hello.java:

- The first statement, `class Hello`, declares a Java class called Hello. The outer curly braces, { and }, define the scope of the code that belongs to the Hello class.
- The second line defines a method. In this example, several Java keywords are used to define it:
  - `public` indicates that this method can be invoked from any other class.
  - `static` specifies that this method applies to the class globally, instead of at the instance level.
  - `void` indicates that this method does not return any value.
  - `main` makes this program an application. It tells the Java interpreter to call this method when the program is loaded.
  - The next two lines print the text written between the double quotes. Each line invokes the println method of the PrintStream class. The

PrintStream class is instantiated as out of the class named System. The System class is instantiated as System.

Note: All Java statements end with a semicolon.

Compile the application with the Java compiler:

```
[C:\myjava]javac Hello.java
```

The compiler creates the file Hello.class. To run the application invoke the Java interpreter:

```
[C:\myjava]java Hello
```

Note: The interpreter does not need the extension class to find the application. As result you will see the following lines in your window:

```
[C:\myjava]javac Hello.java

[C:\myjava]java Hello
Hello World!
Welcome to MQSeries client for Java.

[C:\myjava]
```

5.3.2 The Hello World Applet

An applet is always called from within an HTML file. Therefore, we have to create two files:

1. The Java program HelloWorld.java
2. The HTML file HelloWorld.html

5.3.2.1 Writing a Java Applet

This is the applet HelloWorld.java:

```
import java.awt.Graphics;
public class HelloWorld extends java.applet.Applet {
    public void init () {
           resize (300,200);
    }
    public void paint (Graphics g) {
           g.drawRect(0,0,size().width - 1, size().height - 1);
           g.drawString("Hello World!", 25, 25);
           g.drawString("Welcome to MQSeries client for Java.",25,50);
    }
}
```

Figure 28. A Simple Java Applet

Now let us analyze the source code of HelloWorld.java.

- `import java.awt.Graphics;`

  The first line imports the java.awt package that contains the graphics classes. You need them when you want to draw something on the screen. Packages are the means by which several different Java classes can be stored together. java.awt refers to the Abstract Window Toolkit (awt) package.

- `public class HelloWorld extends java.applet.Applet`

  The second line declares the public class HelloWorld as an applet. The word main is absent from this statement. The parameter extends indicates that the class inherits from the java.applet.Applet class. Java allows single inheritance.

- `public void init()`

  This is the initialization method. This is the first called method of an instance of a class. Here you could initialize variables. Be aware that if this class is inherited by another class, this method can be overridden in that class and, therefore, is never called.

- `resize(300, 200);`

  With this line you define the size of the applet in pixels.

  Note: This value has no effect inside a browser, only in the applet viewer.

- `public void paint(Graphics g)`

  This second method is called by the AWT when the applet needs to be drawn or redrawn. In our example, we write (draw) two character strings and a box around them. To do that we use two drawing methods of the graphics class.

- `g.drawRect(0,0,size().width - 1, size().height - 1);`

  The first statement in the paint method draws a rectangle just inside the applet. drawRect has four parameters, the coordinates for the top left corner (0,0), and the width and height of the rectangle, all values in pixels. Width and height depend on the size of the applet.

- `g.drawString("Hello World!", 25, 25);`
  `g.drawString("Welcome to MQSeries client for Java.",25,50);`

  The two drawString methods cause the text between the double quotes to be displayed. The first parameter following the text is the left side of the text, the second is the baseline for the string.

  Note: g is an instance of the graphics class.

5.3.2.2 Compiling an Applet

Compile the applet with the Java compiler:

```
[C:\myjava]javac HelloWorld.java
```

The compiler creates the file HelloWorld.class.

If compilation fails, make sure you typed in and named the program exactly as shown above. One of the most common mistakes in the beginning is to forget a semicolon on the end of a statement or mismatch within the pair of { } .

5.3.2.3 Writing an HTML File that Calls an Applet

Applets are always called from an HTML file. The following file, HelloWorld.html, calls the HelloWorld applet.

```
<HTML>
<HEAD><TITLE>A Simple Program</TITLE>
</HEAD>
<BODY>
Here is the output of my program:
<APPLET CODE="HelloWorld.class" WIDTH=300 HEIGHT=200>
</APPLET>
</BODY>
</HTML>
```

Figure 29. HTML File that Calls an Applet

The file HelloWorld.html in Figure 29 calls the applet "HelloWorld.class" with this statement:

```
<applet code="HelloWorld.class" width=150 height=50>
```

The keywords mean:

applet Is the tag that calls a Java applet.

code Defines the full name of the Java applet.

width Defines the width of the applet window.

height Defines the height of the applet window.

The other HTML keywords define the rest of the page. They are described in more detail in Chapter 4, "HTML Overview" on page 27.

5.3.2.4 Executing an Applet

You can run the applet two ways:

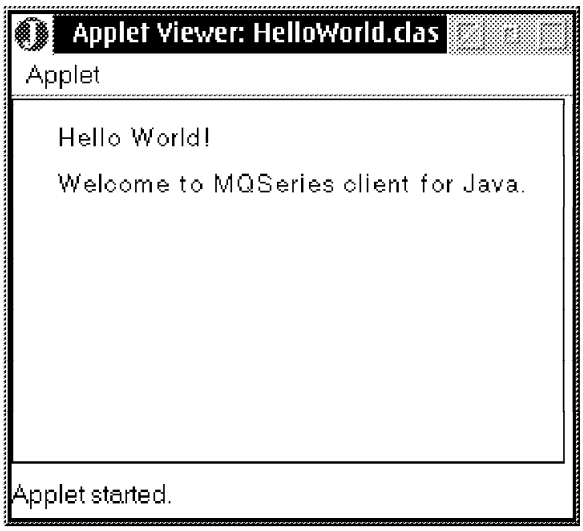
- From an applet viewer.
- From a Java-enabled browser.

In either case you load the HTML file first and let it display the applet. If you use the applet viewer, you will see only the applet and not the other contents of the HTML file. To view the applet with the applet viewer enter on the command line:

```
[C:]applet \myjava\HelloWorld.html
-- or --
[C:\myjava]applet HelloWorld.java
```

Note: In some operating systems the name of the appletviewer EXE is applet, in others appletviewer.

Once you have successfully completed all the above steps, you should see the applet in your window:



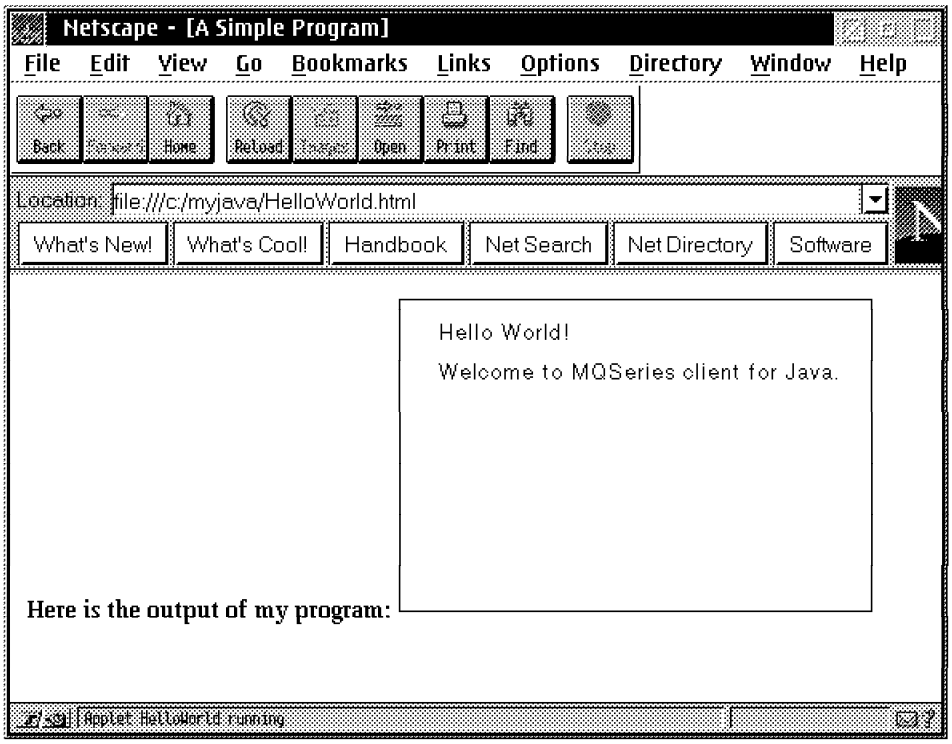
Important

Do *not* invoke the applet viewer from the HTML directory if you might want to reload the applet. Because of the way the class loader works, an applet can′t be reloaded (for example, after you make changes to its code) when you invoke the applet viewer from the directory that contains the applet′s compiled code.

To view the applet with a browser type the file name and its full path:

```
file:///c:/myjava/HelloWorld.html
```

The Netscape Navigator displays the applet as shown on page 51. Note that the applet appears to the right of the text. To draw the applet below the text insert a <p> tag before the <APPLET> tag in the HTML file in Figure 29 on page 49.



5.3.2.5 You Did It

This should only be the start to your next and more complex and powerful applet.

In this redbook we don't describe the Java language in more detail, because there are so many good books available.

If you are familiar with programming in C, you should have a look at the book *Java in a Nutshell* by David Flanagan (ISBN 1-56592-183-6).

But if you are a beginner, you can try the book *Teach Yourself Java in 21 Days* by Laura Lemay and Charles L.Perkins (ISBN 1-57521-030-4).

Chapter 6. MQSeries Overview

MQSeries is IBM's award-winning middleware for commercial messaging and queuing. It runs on a variety of platforms. The MQSeries products enable programs to communicate with each other across a network of unlike components, such as processors, subsystems, operating systems and communication protocols. MQSeries programs use a consistent application program interface (API) across all platforms.

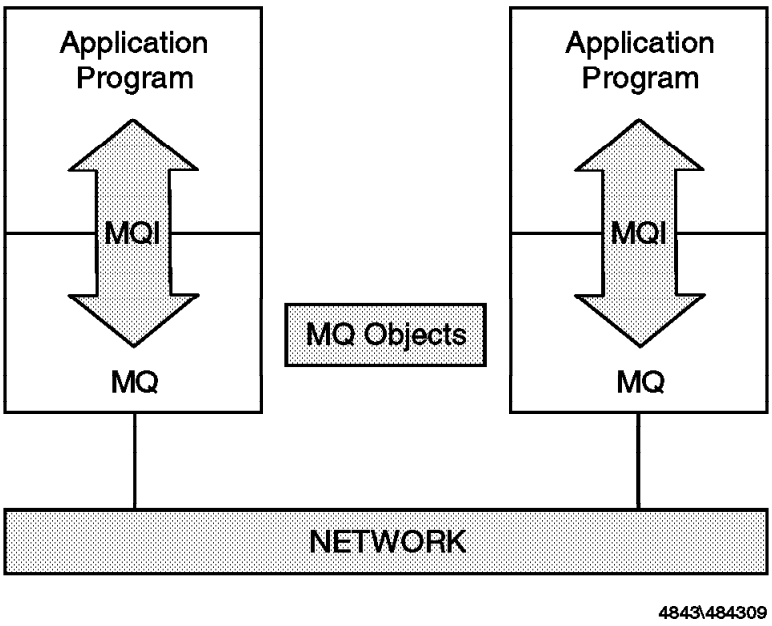


Figure 30. MQSeries at Runtime

Figure 30 shows the parts of an MQSeries application at runtime. Programs use MQSeries API calls, that is the message queue interface (MQI), to communicate with a queue manager (MQM), the run-time program of MQSeries. For the queue manager to do its work, it refers to objects, such as queues and channels. The queue manager itself is an object as well.

The following sections provide a brief overview of MQSeries.

6.1 What Is Messaging and Queuing?

Messaging means that programs communicate with each other by sending data in messages and not by calling each other directly.

Queuing means that programs communicate through queues. Programs communicating through queues need not be executed concurrently.

With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. In contrast, synchronous messaging waits for the reply before it resumes processing.

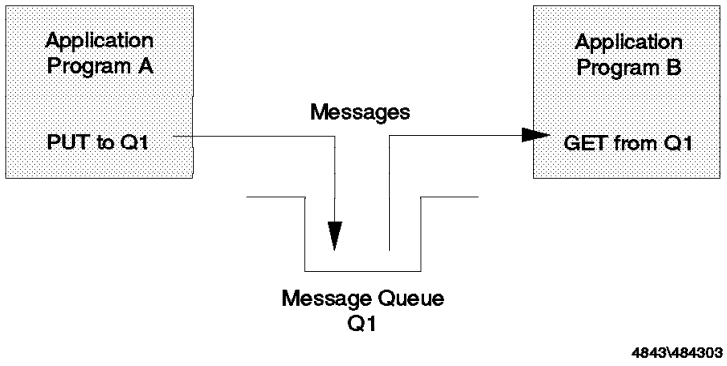


Figure 31. Message Queuing: Principle

MQSeries is used in a client/server or distributed environment. Programs belonging to an application can run in one workstation or in different machines on different platforms.

Attention

Message queuing is a method of program-to-program communication. Programs within an application communicate by writing and retrieving application-specific data (messages) to/from queues, without having a private, dedicated, logical connection to link them.

6.1.1 Messages

A message consists of two parts: data that is sent from one program to another and a message descriptor. The message descriptor identifies the message (message ID) and contains control information, also called attributes, such as message type, expery time, correlation ID, priority, and the name of the queue for the reply.

MQSeries knows four types of messages:

Datagram A message containing information for which no response is expected.

Request A message for which a reply is requested.

Reply A reply to a request message.

Report A message that describes an event such as the occurrence of an error.

There are persistent and non-persistent messages. *Persistent messages* are written to logs on a hard drive and survive system failures. *Non-persistent* messages cannot be recovered after a system restart.

6.1.2 Queue Manager

The heart of MQSeries is its run-time program, the *queue manager* (MQM). Its job is to manage queues of messages. Application programs invoke functions of the queue manager by issuing API calls. For example, the MQPUT API puts a message on a queue to be read by another program using the MQGET API. This scenario is shown in Figure 31 on page 54.

A program may send messages to another program that runs in the same machine as the queue manager, or to a program that runs in a remote system, such as a server or a host. The remote system has its own queue manager with its own queues.

Application programmers do not need to know where the program runs they send messages to. They put their message on a queue and let the queue manager worry about the destination machine and how to get it there.

For the queue manager to do its work, it refers to objects that are defined by an *administrator*, usually when the queue manager is created or when a new application is added. The objects are described in the next section.

The functions of a queue manager can be defined as follows:

- It manages queues of messages for application programs.
- It provides an application programming interface, the Message Queue Interface (MQI).

  Note: The Networking Blueprint identifies three communication styles:

  - Common Programming Interface - Communications (CPI-C)
  - Remote Procedure Call (RPC)
  - Message Queue Interface (MQI)

- It uses networking facilities to transfer messages to another queue manager when necessary.
- It provides additional functions that allow administrators to create and delete queues, alter the properties of existing queues, and control the operation of the queue manager. These functions are invoked through the utility RUNMQSC, which stands for run MQSeries commands.

6.1.3 Queue Manager Objects

The queue manager itself is an object. Usually, an administrator creates it with the command crtmqm, either from the command line or from an icon. You can create several queue managers in one system. One of them should be the default queue manager. The following command creates the default queue manager JAVAMQM:

```
crtmqm /q JAVAMQM
```

The /q makes it the default MQM. The name is case-sensitive. To start the default queue manager issue the command:

```
strmqm
```

Before the queue manager can do any messaging and queueing the administrator has to define objects, such as queues. There are some default definitions for objects every queue manager needs. They are defined in a file provided with MQSeries. To define these default objects use the utility RUNMQSC, also provided with the product. The command to create these objects is:

```
runmqsc < c:\mqm\mqsc\amqscoma.tst > out.lst
```

The queue manager must be running to create the objects defined in the file amqscoma.tst. Check the last lines of the output file, here out.lst, for any errors.

The queue manager can own objects of the following types:

- Queues
- Process definitions
- Channels

The objects are common across different MQSeries platforms. There are other objects that apply to MVS systems only, such as the buffer pool, PSID, and the storage class.

6.1.3.1 Queues

Message queues are used to store messages sent by a programs. There are local queues that are owned by the local queue manager, and remote

queues that belong to a different queue manager. Queues are described in more detail in 6.3, “Message Queues” on page 59.

6.1.3.2 Channels

A channel is a logical communication link. In MQSeries, there are two different kinds of channels:

- Message channel

  A message channel connects two queue managers via message channel agents (MQA). Such a channel is unidirectional. It comprises two message channel agents (MCA), a sender and a receiver, and a communication protocol. An MCA is a program that transfers messages from a transmission queue to a communication link or vice versa. For bidirectional messaging you have to define two channels, a sender channel and a receiver channel.

- MQI channel

  A Message Queue Interface (MQI) channel connects an MQI client to a queue manager in a server machine. MQI clients don′t have a queue manager of their own. An MQI channel is bidirectional.

Figure 36 on page 62 shows the use of both channel types. For more detailed information refer to the *Distributed Queuing Guide*.

6.1.3.3 Process Definitions

A process definition object defines an application to a queue manager. For example, it contains the name of the program to trigger when a message arrives.

6.2 Manipulating MQM Objects

MQSeries provides the utility RUNMQSC to create and delete queue manager objects and to manipulate them. The queue manager must be running when you use the utility. RUNMQSC works in two ways:

- You can type the commands.
- You can create a file containing a list of commands and use this list as input.

The commands in Figure 32 on page 58 start the default queue manager and create a local queue for it.

```
[C:\]strmqm
MQSeries queue manager running.

[C:\]runmqsc
33H2205,5622-908 (C) Copyright IBM Corp. 1994,1995. ALL RIGHTS RESERVED.
Starting MQSeries Commands.

define qlocal('QUEUE1') replace descr ('test queue')
     1 : define qlocal('QUEUE1') replace descr ('test queue')
AMQ8006: MQSeries queue created.
Ctrl + C             <--- ends RUNMQSC
1 MQSC commands read.
0 commands have a syntax error.
0 commands cannot be processed.

[C:\]
```

Figure 32. RUNMQSC - Interactive

Another way to create MQSeries objects is by using an input file instead of typing the commands.

```
[C:\]strmqm
MQSeries queue manager running.

[C:\]runmqsc < mycoma.tst > a.a

[C:\]
```

Figure 33. RUNMQSC - Using Command File

The input file contains the following lines. The + indicates that the command continues on the next line.

```
*********************************************************************/
* File:  MYCOMA.TST                                                  */
*********************************************************************/

    DEFINE QLOCAL('QUEUE1') REPLACE +
           DESCR('Test Queue')
```

Figure 34. RUNMQSC - Input File

The output can appear in the window or can be redirected to a file by specifying a < followed by a file name. The output for the above file would look like this:

```
       : 622-908 (C) Copyright IBM Corp. 1994,1995. ALL RIGHTS RESERVED.
Starting MQSeries Commands.

       : ************************************************************
       : * File:  MYCOMA.TST
       : ************************************************************
       :
     1 :     DEFINE QLOCAL('QUEUE1')  REPLACE +
       :            DESCR('Test Queue')
AMQ8006: MQSeries queue created.
       :
1 MQSC commands read.
0 commands have a syntax error.
0 commands cannot be processed.
```

Figure 35. RUNMQSC - Output File

6.3 Message Queues

Queues are defined as objects belonging to a queue manager. MQSeries knows a number of different queue types, each with a specific purpose.

6.3.1 Local Queue

A queue is local if it is owned by the queue manager to which the application program is connected. They are used to store messages for programs that use the same queue manager. For example, each of program A and program B has a queue for incoming messages and another queue for outgoing messages. Since the queue manager serves both programs, all four queues are local.

Note: Both programs do not have to run in the same workstation. Client workstations usually use a queue manager in a server machine.

6.3.2 Remote Queue

A queue is remote if it is owned by a different queue manager. It is the local definition of a remote queue.

Applications do not need to know the location of the remote queue. Programs write messages to queues. The local queue manager is responsible for forwarding the messages to the remote queue manager.

Note: A program cannot read messages from a remote queue.

6.3.3 Transmission Queue

A remote queue is associated with a transmission queue. Transmission queues are used as an intermediate step when sending messages to remote queues.

Typically, there is only one transmission queue for each remote queue manager. All messages written to queues owned by a remote queue manager are actually written to the transmission queue for this remote queue manager. The messages will then be read from the transmission queue and sent to the remote queue manager.

Transmission queues are transparent to the application. They are used internally by the queue manager.

Note: When a program opens a remote queue, the attributes of the queue are obtained from the transmission queue. Therefore, the results of a program writing messages to a queue will be affected by the transmission queue characteristics.

6.3.4 Dynamic Queue

Such a queue is defined *on the fly* when the application needs it. They may be retained by the queue manager or automatically deleted when the application program ends.

Dynamic queues are local queues. They are often used in conversational applications, to store intermediate results. Dynamic queues can be:

- Temporary queues that do not survive queue manager restarts
- Permanent queues that do survive queue manager restarts

6.3.5 Model Queue

A model queue is not a real queue. It is a collection of attributes that can be used when a *dynamic queue* is created.

6.3.6 Alias Queue

Alias queues are not real queues but definitions. They are used to assign different names to the same physical queue. This allows multiple programs to work with the same queue, accessing it under different names and with different definitions.

6.3.7 Initiation Queue

An initiation queue is a local queue to which the queue manager writes a *trigger message* when certain conditions are met on another local queue, for example, when a message is put into an empty message queue. Trigger

messages are read by the trigger monitor, an MQSeries application. The trigger monitor then starts the application that will process the message.

Note: Applications do not need to be aware of initiation queues, but the triggering mechanism implemented through them is a powerful tool to design and write asynchronous applications.

6.3.8 Reply-To Queue

A request message must contain the name of the queue into that the responding program must to put the reply message.

6.3.9 Dead-Letter Queue

A queue manager must be able to handle situations when it cannot deliver a message. Here are some examples:

- The destination queue is full.
- The destination queue does not exist.
- Message puts have been inhibited on the destination queue.
- The sender is not authorized to use the destination queue.
- The message is too large.
- The message contains a duplicate message sequence number.

These messages are written by the queue manager to a dead-letter queue. A dead-letter queue is defined when the queue manager is created. It will be used as a repository for all messages that cannot be delivered.

6.4 Clients and Servers

Before you install MQSeries you have to decide if the workstation shall become an MQ client or an MQ server. There are two kinds of clients:

- Slim client
- Fat client

Fat clients have a local queue manager; slim clients don′t.

When a slim client cannot connect to its server it cannot work, because queue manager and queues for a slim client reside in the server. Usually, an MQSeries client is a slim client. Several of these clients share MQSeries objects, and the queue manager is one of them, in the server they are attached to.

In some cases it may be advantageous to have queues in the end user′s workstation, especially in a mobile environment. That allows you to run

your application when a connection between client and server does (temporarily) not exist.

The difference between an end user's workstation that is a client and one that has a queue manager is the way messages are sent.

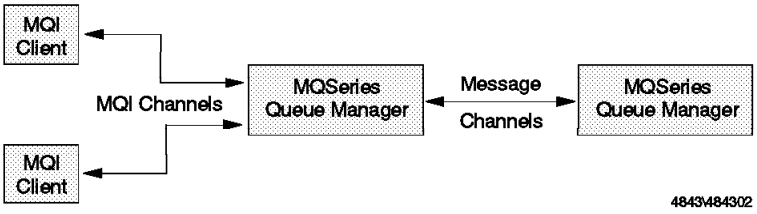


Figure 36. MQSeries Channels

Figure 36 shows the use of MQI and message channels.

- MQI channels connect clients to a queue manager in a server machine.
- A message channel connects a queue manager to another queue manager in another system.

Note: MQI channels are faster than message channels.

The following sections describe what you have to do to define and test the connection between an MQ client and an MQ server. A more detailed description is in the publication *MQSeries Clients*.

Note

Since the MQSeries client for Java runs in client machines, we need to define MQI channels between clients and server. In this publication we do not provide examples of setting up message channels.

6.4.1 How to Define a Client/Server Connection

The following describes MQM objects and other definitions needed for the Java application developed throughout this publication.

To define the connection, you have to know the transmission protocol and the addresses of the systems. We use TCP/IP; the addresses are:

- 9.24.104.206 for the server.
- 9.24.104.116 for the client.

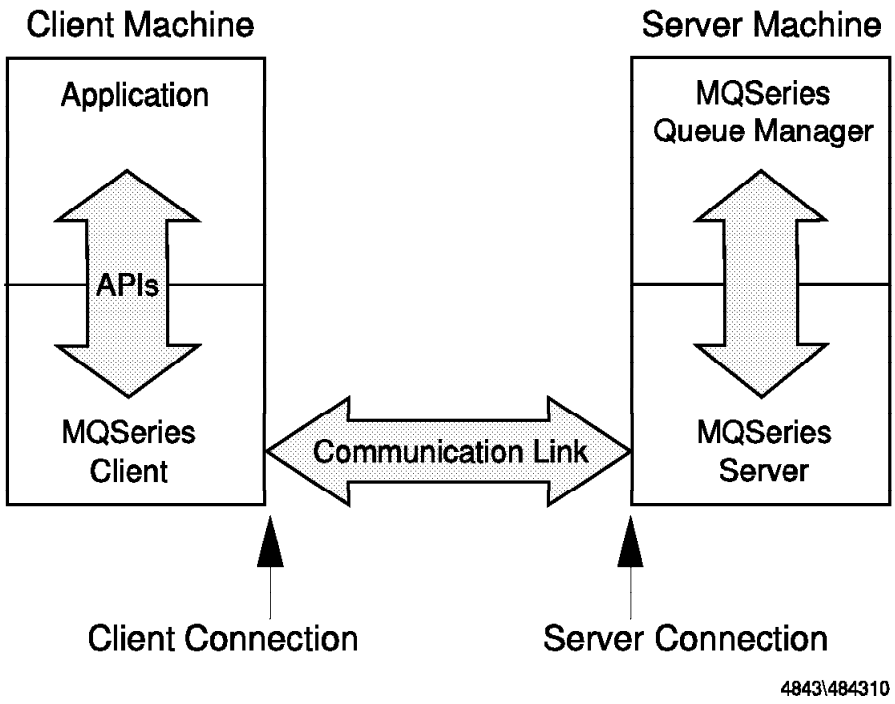


Figure 37. Client/Server Connection

On the server: Define the queues that the application needs and a channel of the type server connection. The queue manager definitions are in the file javacoma.tst, shown in Figure 38.

```
*********************************************************************/
* File:  JAVACOMA.TST                                               */
*********************************************************************/

    DEFINE QLOCAL('JAVAQ1')  REPLACE +
           DESCR('Queue for Java Application')

    DEFINE CHANNEL('JAVACH1')  CHLTYPE(SVRCONN) REPLACE +
           TRPTYPE(TCP) MCAUSER(' ')
```

Figure 38. Definitions for Server Connection

We define a queue for the application to put messages in and an MQI channel of the type server connection. Create the objects by issuing the command:

```
runmqsc < javacoma.tst > a.a
```

On the client: Define an environment variable for the MQSeries client that defines the connection on the client side. Set the variable with the following command or place the command in the CONFIG.SYS.

```
set MQSERVER=JAVACH1/TCP/9.24.104.206(1414)
```

Notes:

1. MQSERVER is the name of the environment variable.
2. JAVACH1 is the name of the channel to be used for communication between client and server. The channel must be defined in the server.
3. TCP denotes that TCP/IP is to be used to connect to the machine with the address following the parameter.
4. (1414) is the default port number for MQSeries. You may omit this parameter if the listener on the server side uses this default, too.

6.4.2 How to Start a Client/Server Connection

Before you can start the application in the client you have to start a program that listens to the communication link between client and server. MQSeries provides a program that does just that. You start it with the following command:

```
start runmqlsr /t tcp /m JAVAMQM /p 1414
```

Notes:

1. start creates a new window for the listener.
2. runmqlsr is the name of the listener.
3. /t tcp defines that there is a TCP/IP connection between client and server.
4. /m JAVAMQM specifies the name of the queue manager the client connects to. If omitted, the default queue manager is used.
5. /p 1414 defines the TCP/IP port number. 1414 is the default assigned to MQSeries applications hence, you may omit this parameter.

The server is now ready to process MQI calls from the application running in the client.

6.4.3 How to Test a Client/Server Connection

With the steps described above communication between client and server is established. You can test the connection using programs provided with MQSeries. They are in the directory:

```
c:\mqm\tools\c\samples\bin
```

- AMQSPUTC puts messages on a queue.
- AMQSGETC gets messages from a queue.

On the client machine, type the commands shown in Figure 39.

- After AMQSPUTC is started type a few messages and then press Enter twice to end it.
- AMQSGETC times out after a few seconds.

```
[C:\mqm\tools\c\samples\bin]amqsputc JAVAQ1
Sample AMQSPUT0 start
target queue is JAVAQ1
111111
222222
333333
                    <--- 2 x Enter ends AMQSPUTC
Sample AMQSPUT0 end

[C:\mqm\tools\c\samples\bin]amqsgetc JAVAQ1
Sample AMQSGET0 start
message <111111>
message <222222>
message <333333>
no more messages    <--- Wait for time out
Sample AMQSGET0 end

[C:\mqm\tools\c\samples\bin]
```

Figure 39. Testing Client/Server Connection

On the server machine, you see the following listener window after completion of the two programs:

```
RUNMQLSR.EXE

11/19/96  14:54:10 Channel program started
11/19/96  14:54:28 Channel program ended normally
11/19/96  14:55:20 Channel program started
11/19/96  14:55:49 Channel program ended normally
```

Figure 40. Listener Window (RUNMQLSR)

6.4.4 How to Test a Client/Server Connection with MQSeries for Java

MQSeries for Java provides a test program you may use to check if the client/server connection works correctly. In a Java-enabled browser, we used the Netscape Navigator:

- To connect to the AIX machine mercury:

  `http://mercury.itos.ral.ibm.com/mqjavac.html`

- To connect to the OS/2 server mqjava:

  `http://mqjava.itos.ral.ibm.com/mqjavac`

The HTML file displays the applet shown below:

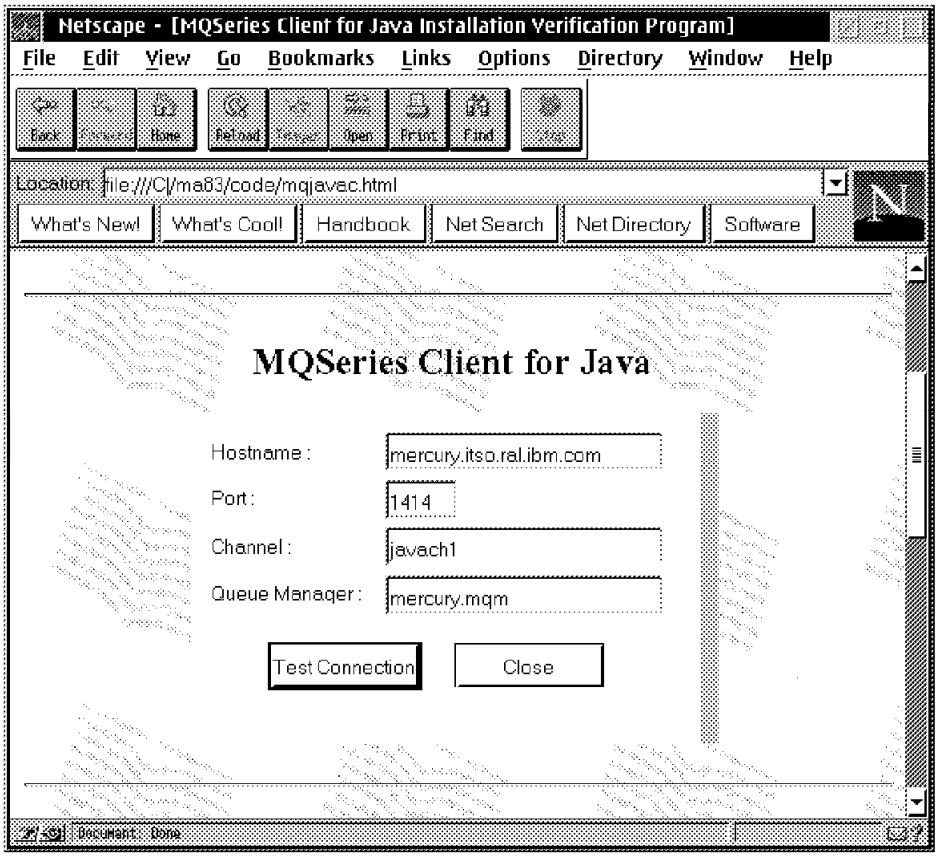


Figure 41. Verify Installation Program (IVP)

To verify your MQSeries Client for Java installation, proceed as follows:

1. In the Hostname entry field, enter the TCP/IP hostname of the machine on which your MQSeries Queue Manager is running. If you downloaded

this page from a Web server, you may find that the Hostname field has already been filled in for you.

Note: When you download an applet from a Web server, you can only communicate with an MQSeries Queue Manager running on the same machine as the Web server.

2. In the Port entry field, enter the port on which your MQSeries Queue Manager listens for connection requests. (The MQSeries default is 1414.)
3. In the queue manager entry field, enter the name of the queue manager to which you wish to connect. If you leave this field blank, the client will connect to the default queue manager of the target machine.

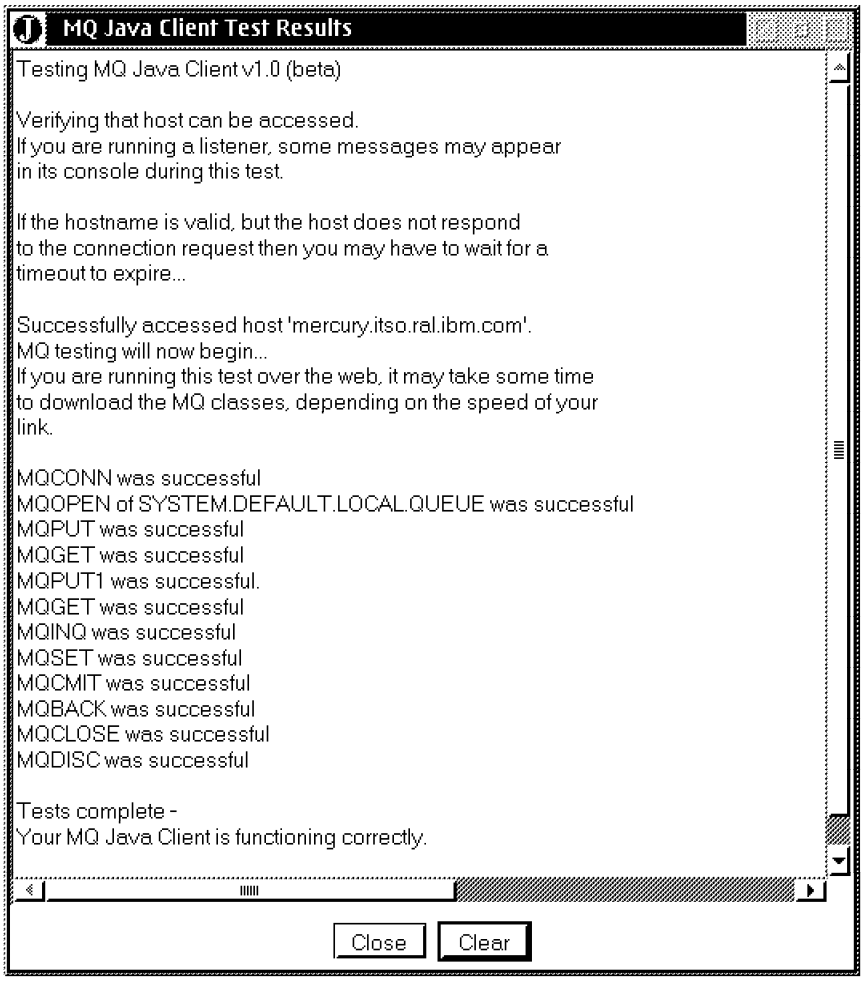


Figure 42. Verify Installation Results

4. In the channel entry field, enter the name of the MQSeries channel that the client should use when connecting. Remember that channel names are case-sensitive.
5. The program uses the SYSTEM.DEFAULT.LOCAL.QUEUE that is created when you create the default objects for the queue manager using the file amqscoma.tst.
6. Click on the **Test Connection** button. A dialog box appears displaying the the results of the test. If the test fails for any reason, please take the corrective action suggested in the results window and try again.
7. If the window shows the results as shown in Figure 42 on page 67, you have successfully communicated with your queue manager using the MQSeries Client for Java.

   Refer now to the *MQSeries Client for Java User's Guide* for details on writing your own applications to access MQSeries Queue Managers via the World Wide Web.

6.4.5 How to Trigger Applications

This section describes how to trigger an application program that runs in the server machine. Since there are MQI channels of the type server connection between clients and server, all clients use the queue manager in the server machine. When a client puts a message on a queue it has to be read and processed by some program. This program can be started when the server starts or the queue manager starts it when it is needed, that is using the MQSeries triggering mechanism.

Scenario:

1. The client starts a program that puts a message on a queue.

   For this function five MQSeries API calls are executed:

   - MQCONN connects the queue manager in the server (JAVAMQM).
   - MQOPEN opens the message queue (JAVAQ1).
   - MQPUT puts the message on the queue.
   - MQCLOSE closes the queue (JAVAQ1).
   - MQDISC disconnects from the queue manager (JAVAMQM).

   The MQSeries client code that runs in the client machine processes the API calls and routes them to the machine defined in the environment variable, such as:

   ```
   set MQSERVER=JAVACH1/TCP/9.24.1104.206
   ```

2. In the server machine, the following queue manager objects are needed:

- A channel, JAVACH1, of the type server connection.
- A local queue, JAVAQ1, into which the clients put their messages.
- An initiation queue into which the queue manager puts a trigger message, for example then when a message is put on JAVAQ1. Here we use the default queue defined in AMQSCOMA.TST.
- A process definition, process.appl1, that contains the name of the program to be started when the trigger event occurs.
- A queue, JAVAQ2, in which the program puts the reply message.
- Another queue, JAVAQ3, that will contain order completion status messages.

```
DEFINE CHANNEL('JAVACH1') CHLTYPE(SVRCONN) REPLACE +
       TRPTYPE(TCP) MCAUSER(' ')

DEFINE QLOCAL('JAVAQ1') REPLACE +
       DESCR('Queue for Application') +
   TRIGTYPE(EVERY) +
       TRIGGER INITQ (system.default.initiation.queue) +
       PROCESS (process.appl1)

DEFINE PROCESS(process.appl1) REPLACE +
       DESCR('Process for business logic') +
       APPLTYPE (OS2) +
       APPLICID('c:\mqtest\bl1.exe')

DEFINE QLOCAL('JAVAQ2') REPLACE +
       DESCR('Reply queue')

DEFINE QLOCAL('JAVAQ3') REPLACE +
       DESCR('Order status queue')
```

3. In the server machine, two programs have to be started:
   - The listener runmqlsr /t tcp.
   - The trigger monitor runmqtrm.
   - On AIX, process amqcrsta is started via inetd for the listener process.
4. The listener listens for messages on the channel and puts them on the queue JAVAQ1.
5. By default, the MQM puts a trigger message on the trigger queue each time a message is put on JAVAQ1.
6. When a message is placed on the trigger queue, the trigger monitor starts the program defined in the process.

6.5 Message Queuing Interface (MQI)

A program talks directly to its local queue manager. It resides in the same processor or domain (for clients) as the program itself. The program uses the Message Queuing Interface (MQI). The MQI is a set of API calls that request services from the queue manager.

There are eleven APIs. The most important ones are:

MQPUT Put a message on a queue.

MQGET Get a message from a queue.

The other calls are used less frequently:

MQCONN Establish connection with a queue manager.

MQOPEN Open or obtain access to a queue.

MQCLOSE Close a queue.

MQDISC Disconnect from the queue manager

MQPUT1 Open a queue, put a message on it and close the queue.

MQINQ Request information about one of the queue manager's objects.

MQSET Change the attributes of a queue.

MQCMIT A syncpoint has been reached. Messages put as part of a unit of work are made available to other applications. Messages retrieved as part of a unit of work are deleted.

MQBACK The queue manager has to back out all message puts and gets that have occurred since the last syncpoint. Messages put as part of a unit of work are deleted. Messages retrieved as part of a unit of work are reinstated on the queue.

Example:

The code fragment in Figure 43 on page 71 shows the APIs to put a message on a queue and get the reply from another queue.

```
MQHCONN  HCon;                  // Connection handle
MQHOBJ   HObj1;                 // Object handle for queue 1
MQHOBJ   HObj2;                 // Object handle for queue 2
MQLONG   CompCode, Reason;      // Return codes
MQLONG   options;
⋮
MQOD     od1 = {MQOD_DEFAULT};  // Object descriptor for queue 1
MQOD     od2 = {MQOD_DEFAULT};  // Object descriptor for queue 2
MQMD     md  = {MQMD_DEFAULT};  // Message descriptor
MQPMO    pmo = {MQPMO_DEFAULT}; // Put message options
MQGMO    gmo = {MQPMO_DEFAULT}; // Get message options
⋮
// 1 Connect application to a queue manager.
strcpy  (QMName,"MYQMGR");
MQCONN  (QMName, &HCon, &CompCode, &Reason);

// 2 Open a queue for output
strcpy  (od1.ObjectName,"QUEUE1");
MQOPEN  (HCon,&od1, MQOO_OUTPUT, &Hobj1, &CompCode, &Reason);

// 3 Put a message on the queue
MQPUT   (HCon, Hobj1, &md, &pmo, 100, &buffer, &CompCode, &Reason);

// 4 Close the output queue
MQCLOSE (HCon, &Hobj1, MQCO_NONE, &CompCode, &Reason);

// 5 Open input queue
options = MQOO_INPUT_AS_Q_DEF;
strcpy  (od2.ObjectName, "QUEUE2");
MQOPEN  (HCon, &od2, options, &Hobj2, &CompCode, &Reason);

// 6 Get message
gmo.Options = MQGMO_NO_WAIT;
buflen  = sizeof(buffer - 1);
memcpy (md.MsgId, MQMI_NONE, sizeof(md.MsgId);
memset (md.CorrelId, 0x00, sizeof(MQBYTE24));
MQGET (HCon, Hobj2, &md, &gmo, buflen, buffer, 100, &CompCode, &Reason);

// 7 Close the input queue
options = 0;
MQCLOSE (HCon, &Hobj2,options, &CompCode, &Reason);

// 8 Disconnect from queue manager
MQDISC  (HCon, &CompCode, &Reason);
```

Figure 43. Fragments of an MQSeries Program

Note: The fields CompCode and Reason will contain completion codes for the APIs. You find them in the *Application Programming Reference*.

1 This statement connects the application to the queue manager with the name MYQMGR. If the parameter QMName does not contain a name, then the default queue manager is used. HCon receives the handle to the queue manager. This handle must be used in all subsequent APIs.

2 To open a queue the queue name must be moved into the object descriptor that will be used for that queue. This statement opens QUEUE1 for output only (open option MQOO_OUTPUT). Returned are the handle to the queue and values in the object descriptor. The handle Hobj1 must be specified in the MQPUT.

3 MQPUT places the message assembled in a buffer on a queue. Parameters for MQPUT are:

- The handle of the queue manager (from MQCONN).
- The handle of the queue (from MQOPEN).
- The message descriptor.
- A structure containing options for the put (refer to the *Application Programming Reference*).
- The message length.
- The buffer containing the data.

4 This statement closes the output queue. Since the queue is predefined no close processing takes place (MQOC_NONE).

5 This statement opens QUEUE2 for input only using the queue-defined defaults. You could also open a queue for browsing, meaning that the message will not be removed.

6 For the get the nowait option is used. The MQGET needs the length of the buffer as input parameter. Since there is no message ID and correlation ID specified, the first message from the queue is read.

7 This statement closes the input queue.

8 The application disconnects from the queue manager.

Communication between programs is *time-independent*. The sender can continue processing without waiting for a reply. The receiving program does not even have to run. MQSeries holds the messages until it is ready to process them.

MQSeries applications are *message-driven*. The arrival of a message triggers an event. Just like clicking on a push button in a GUI invokes some procedure, a message starts a program that processes the message data.

Chapter 7. MQSeries Client for Java

This chapter tells you about IBM MQSeries Client for Java. MQSeries Client for Java provides a set of Java class libraries that permit Java applets on a Web browser, or stand-alone Java applets, to access MQSeries applications. It describes the capabilities of and the advantages MQSeries Client for Java offers to you and your organization.

Documentation for the MQSeries Client for Java is available in in this book and in HTML format. The HTML files accompany the Support Pack version of the product. The identification number (PID) is 5639-C34.

This product has been tested on the following platforms:

- MVS
- AIX
- OS/2
- Windows NT

7.1 Who Should Read This

This chapter is for professionals in industry, finance, government, education, or service organizations who want to explore the benefits that MQSeries Client for Java can bring to their business.

For managers and planners it provides a product overview to help in evaluating the advantages of MQSeries Client for Java and information on how to obtain it. For administrators and support personnel the document provides detailed information on the installation and everyday use of MQSeries Client for Java including problem diagnosis.

Information about programming applications to use MQSeries Client for Java can be found in Chapter 8, “MQSeries Client for Java Programmer’s Guide” on page 83.

7.2 Overview of MQSeries Client for Java

MQSeries Client for Java provides support to enable Java applets and programs to use MQSeries applications. Some scenarios for its use are in Chapter 1, “MQSeries Client for Java Positioning” on page 1.

MQSeries Client for Java is an MQSeries client written in the Java language for communicating via TCP/IP. It enables Web browsers and Java applets to issue calls and queries to MQSeries over the Internet. This gives clients

access to applications running on mainframes and a variety of servers. No other MQSeries code is needed in the client machine.

Transaction Processing

With MQSeries Client for Java the user of an Internet terminal can become a true participant in transactions, rather than just a giver and receiver of information.

The MQSeries Client for Java enables application developers to exploit the power of the Java programming language when writing applets and applications. Applets and applications can run on any platform that supports the Java run-time environment.

Benefits

- MQSeries Client for Java reduces development time for multiplatform MQSeries applications.
- Future enhancements to applets are automatically picked up by end users as the applet code is downloaded.

The client can be installed either on your local hard disk or on a Web server. Installation on a Web server has the advantage of allowing you to download and run MQSeries Client applications on machines that do not have the MQSeries Client for Java installed locally.

Wherever you choose to install the client, it can be run in three different modes:

- From within any Java-enabled Web browser

  When running in this mode, the locations of the MQ Queue Manager that can be accessed may be constrained by the security restrictions of the browser being used.

  This mode permits Java applets on the browser to access MQSeries queues on a server machine. The MQSeries client for Java code must be installed on the MQSeries server. No MQSeries code has to be installed on the client machines as the client code is automatically downloaded when the applet is executed.

  The MQSeries queue manager to which the client connects must be accessible from within the Web browser, which usually means that it must run on the same machine as the Web server.

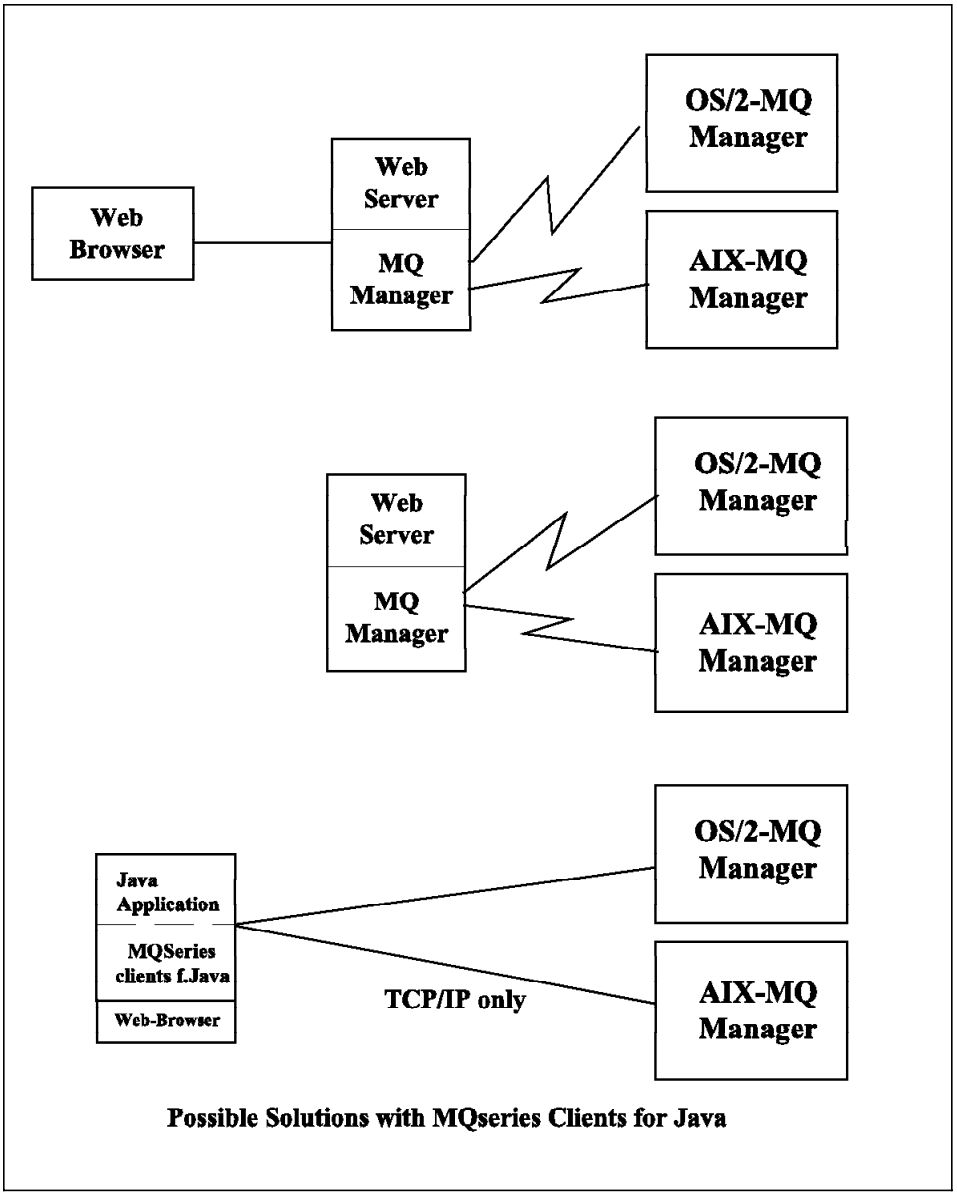


Figure 44. Possible Solutions

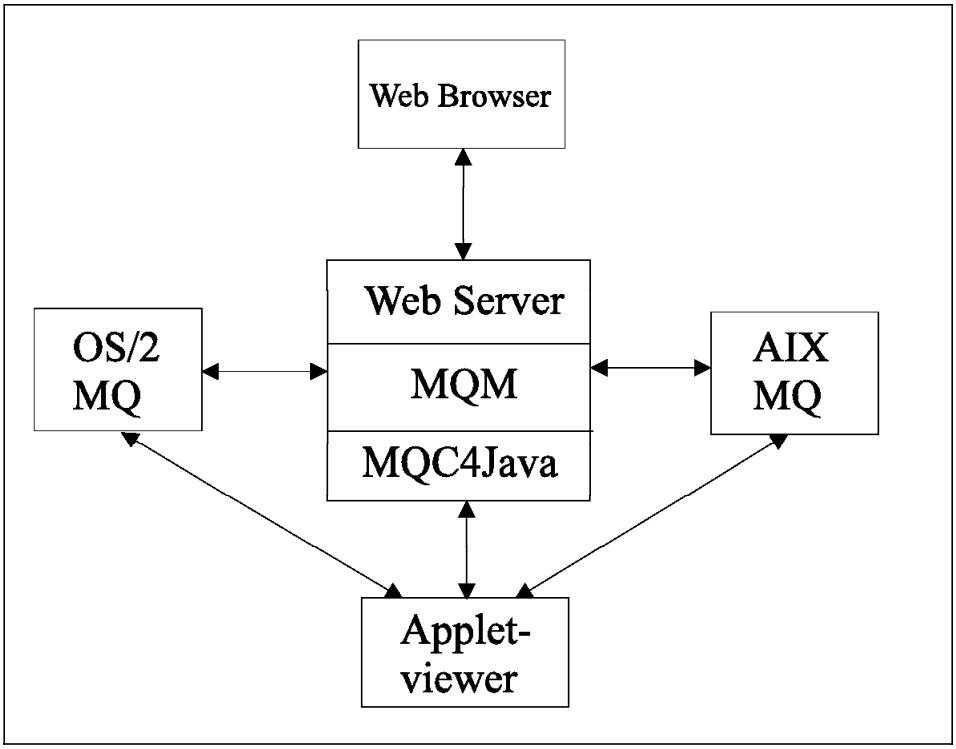


Figure 45. Another View

- Using an applet viewer

  To use this method you must have the Java Developer's Kit (JDK) installed on the client machine.

  For using an applet viewer to view a URL, the MQSeries Client for Java is installed on the server. The queue manager restrictions are the same as for the Java-enabled browser above.

  For using the applet viewer to view a local file, the MQSeries Client for Java code must be installed on the client machine, together with a suitable Java or applet viewer program. In this case, the queue manager to which the client connects can be installed on any machine that is accessible via TCP/IP.

- As a stand-alone Java program

  To use this method you must have the Java Developer's Kit (JDK) installed on the client machine.

7.3 Running the Installation Verification Program

A simple installation verification applet is provided as part of the MQSeries Client for Java in a file called mqjavac.HTML.

The applet connects to a given queue manager, exercises all the MQ calls, and produces simple diagnostic messages in the event of any failures.

The applet can be run from any Java-enabled browser, or with an applet viewer. The method of running the applet depends on where your MQSeries Client for Java is installed, and the method you use to run it. Procedures are described for running with Netscape Navigator, Microsoft Internet Explorer, and an applet viewer in each of the installation environments.

The MQ Queue Manager locations that you can access are also dependent on where your MQSeries Client for Java is installed, and the method you use to run it. The following table summarizes the Queue Manager locations that can be accessed from each environment:

Table 1. MQ Queue Manager Location

Browser	**Installation Location**	
	Local Disk	**Web Server**
Netscape Navigator	any	Web server
Microsoft Internet Explorer	localhost	Web server
appletviewer	any	any

Your queue manager will need to be configured to accept incoming connection requests from the MQSeries clients. You need to:

- Define a server connection channel using the following procedure:
  1. Start your queue manager using the strmqm command.
  2. Type runmqsc to start the runmqsc program.
  3. Define a channel by typing the following command:

     ```
     DEF CHL ('JAVA.CHANNEL') CHLTYPE(SVRCONN) +
      TRPTYPE(TCP) MCAUSER(' ') +
      DESCRIPTION('Sample channel for MQSeries Client for Java')
     ```

- For OS/2 and NT operating systems:

  Start a listener program with the following command:

  ```
  runmqlsr -t tcp [-m QMNAME] -p 1414
  ```

Note: If you are using the default queue manager, the -m flag is not required.

- For UNIX operating systems:

  Configure the inetd daemon, so that the inetd starts the MQ channels. See *MQSeries Clients*, GC33-1632 for instructions on how to do this.

Choose the appropriate procedure from the options below. If the applet does not complete successfully, follow the advice given in the diagnostic messages and try to run the applet again.

7.3.1 Running from a Local Disk Installation

The following procedures assume that the MQSeries Client for Java is installed in a directory called MQJavaClient on the C drive.

7.3.1.1 Running from Netscape Navigator

1. Select **File** from the menu bar.
2. Select **Open File** from the menu.
3. Use the file dialog to select the file mqjavac.HTML in your installation directory.

You can also load this file by specifying a URL in the Location field at the top of the browser window, such as:

```
file:///C:/MQJavaClient/mqjavac.HTML
```

Note: The pathname begins with an extra /.

Using either of these methods, you should be able to connect to any queue manager running on any host to which you have TCP/IP access.

7.3.1.2 Running from Microsoft Internet Explorer

1. Select **File** from the menu bar.
2. Select **Open** from the menu.
3. Use the file dialog to select the file mqjavac.HTML in your installation directory.

You can also load this file by typing in its pathname in the Address field at the top of the browser window. For example:

```
C:\MQJavaClient\mqjavac.HTML
```

Because of security restrictions imposed by the Microsoft Internet Explorer browser, you will only be able to connect to a queue manager running on

your local machine, and you must specify the hostname as localhost in the Hostname entry field of the mqjavac applet.

7.3.1.3 Running from Appletviewer

To use this method you must have the Java Developer's Kit (JDK) installed on your machine.

1. Set your CLASSPATH environment variable to point to your installation directory and the standard Java class files.

   For example, if you have the JDK installed in C:\Java and the MQSeries Client for Java installed in C:\MQJavaClient, then for DOS-based shells, set your CLASSPATH as follows:

   ```
   set CLASSPATH=C:\MQJavaClient;C:\Java\lib\classes.zip
   ```

2. Change to your installation directory.
3. Type `appletviewer mqjavac.HTML`.

   Note: On some platforms, such as OS/2, the command is applet, and not appletviewer.

Using this technique you should be able to connect to any queue manager running on any host to which you have TCP/IP access.

Note that on some platforms you may need to select Properties from the Applet menu at the top left of your screen, and then set Network Access to Unrestricted.

7.3.1.4 Running the Text-Only Version

There is also a text-only version of the installation verification program. To use this you must have the JDK installed on your machine.

1. Set the CLASSPATH environment variable as described above under 7.3.1.3, "Running from Appletviewer."
2. Change to your installation directory.
3. Type `java MQIVP`.

Using this technique you should be able to connect to any queue manager running on any host to which you have TCP/IP access.

7.3.2 Running from a Web Server Installation

The following procedures assume that the MQSeries Client for Java is installed in a directory called MQJavaClient on the web.server.hostname Web server.

7.3.2.1 Running from Netscape Navigator

1. Select **File** from the menu bar.
2. Select **Open Location** from the menu.
3. Enter the following URL in the dialog (or your installations equivalent):

   `http://web.server.hostname/MQJavaClient/mqjavac.HTML`

You can also load this URL by entering it in the Location field at the top of the browser window.

Because of security restrictions imposed by the Netscape browser, you will only be able to connect to a queue manager running on the same host as the Web server.

7.3.2.2 Running from Microsoft Internet Explorer

1. Select **File** from the menu bar.
2. Select **Open** from the menu.
3. Enter the following URL in the dialog or your installations equivalent:

   `http://web.server.hostname/MQJavaClient/mqjavac.HTML`

You can also load the URL by entering it in the Address field at the top of the browser window.

Because of security restrictions imposed by the Microsoft Internet Explorer browser, you will only be able to connect to a queue manager running on the same host as the Web server.

7.3.2.3 Running from Appletviewer

To use this method you must have the Java Developer's Kit (JDK) installed on the client machine. You do not need to set up an explicit CLASSPATH environment variable. Enter the command:

```
appletviewer http://web.server.host/MQJavaClient/mqjavac.HTML
```

On some platforms the command is applet, and not appletviewer.

Using this technique you should be able to connect to any queue manager running on any host to which you have TCP/IP access.

Note: On some platforms, you may need to select Properties from the Applet menu at the top left of your screen, and then set Network Access to Unrestricted.

7.4 Using the Verification Applet to Test Your Customer's Access

You may want to use the installation verification applet to allow your customers to verify their access to your queue manager. A set of optional parameters are included in the file mqjavac.HTML which allows you to modify the applet to suit your requirements. Each parameter is defined in a line of HTML, which looks like the following:

```
<!PARAM name="xxx" value="yyy">
```

To specify a parameter value simply remove the initial exclamation mark, and edit the value as desired. The following parameters can be specified:

hostname Pre-fills the hostname edit box with the supplied value.

port Pre-fills the port number edit box with the supplied value.

channel Pre-fills the channel edit box with the supplied value.

queueManager Pre-fills the queue manager edit box with the supplied value.

userID Uses the specified user ID when connecting to the queue manager.

password Uses the specified password when connecting to the queue manager.

trace Causes the client to write a trace log. Use this option only at the direction of IBM service.

7.5 Running Your Own Applets

To run your own Java applets, load your file into a Web browser, or use the appletviewer command as described above, substituting your application name in place of mqjavac.HTML.

Chapter 8. MQSeries Client for Java Programmer′s Guide

The programmer′s guide provides the information required by programmers who want to write Java applications or applets that include calls to MQSeries queues. MQSeries Client for Java provides a set of Java class libraries that permit Java applets on a Web browser, or stand-alone Java applets, to access MQSeries applications.

The guide includes details of the classes together with advice on how to write MQSeries Java applications.

8.1 Who Should Read This

This chapter is designed for use by programmers who want to write Java applications that use the MQSeries Client for Java.

General information about MQSeries Client for Java is not included in this chapter. For an overview of the product and the advantages it offers to your organization, together with details of how to run and maintain MQSeries Client for Java applications, see Chapter 7, “MQSeries Client for Java” on page 73 and Chapter 1, “MQSeries Client for Java Positioning” on page 1.

8.2 MQSeries Client for Java Support

MQSeries client for Java provides support to enable Java applets and programs to use MQSeries applications.

The MQSeries Client for Java also enables application developers to exploit the power of the Java programming language to create applets and applications that can run on any platform that supports the Java run-time environment. These factors combine to dramatically reduce the development time for multiplatform MQSeries applications and future enhancements to applets are automatically picked up by end users as the applet code is downloaded.

8.2.1 Java Developer′s Kit (JDK)

Before you can compile any applets that you write, you need to have access to the Java Developers Kit (JDK) for your development platform. The JDK contains all the standard Java classes, variables, constructors, and interfaces on which the MQSeries Java classes depend, and the tools required to compile and run the applets on each supported platform.

If you do not have the JDK that you need, obtain it from one of the following Web sites:

- http://www.hursley.ibm.com/javainfo/hurindex.HTMLl for OS/2, AIX, Windows 3.1, OS/400, OS/390
- http://java.sun.com/download.HTML for Solaris, Windows 95, Windows/NT

8.2.2 Java Client Class Library

The MQSeries client for Java is a set of Java classes that enable Java applets and applications to interact with MQSeries queues without the need for any other MQSeries code on the client machine.

The Java client contains the following classes:

- MQChannelDefinition
- MQChannelExit
- MQEnvironment
- MQException
- MQGetMessageOptions
- MQManagedObject
- MQMessage
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager

The Java client contains the following Java interfaces:

- MQReceiveExit
- MQSecurityExit
- MQSendExit
- MQC

The classes and interfaces are shipped as a Java *package* called com.ibm.mq.

See the class hierarchy on page 101.

8.2.3 Writing Programs for the MQSeries Client for Java

The MQSeries Client for Java is similar to the MQSeries C client but has the following differences:

- It only supports TCP/IP.
- It does not support connection tables.
- It does not read any environment variables at startup.
- Information that would be stored in a connection definition and in environment variables, is stored in an instance of a class called MQEnvironment.
- It does not write a trace log.
- Error and exception conditions are written to a log specified in the MQEnvironment class. The default error destination is the Java console.

For general information on MQSeries clients see *MQSeries Clients*, GC33-1632.

To access MQSeries queues using the Java client you need to write a Java applet containing calls to put messages onto, and get messages from MQSeries queues, as shown in the following code fragment.

8.2.4 Sample Code Fragment

The code fragment in Figure 46 on page 86 through Figure 48 on page 88 demonstrates a very simple applet that connects to a queue manager, puts a message onto SYSTEM.DEFAULT.LOCAL.QUEUE and gets it again.

This sample runs as an applet using the appletviewer and HTML file using the command:

```
appletviewer MQSample.HTML
```

Output is to the command line, *not* the applet viewer window.

Notes:

1. If you receive MQ error 2 reason 2059 and you are sure your MQ and TCP/IP setup is correct. You should click on the Applet selection in the Applet viewer window, select properties, and change Network access to unrestricted.
2. It is assumed that MQ Server is listening on the default TCP/IP port of 1414.

```
import com.ibm.mq.*;        // Include the MQ package

public class MQSample extends java.applet.Applet
{
                            // Define the name of your host to connect to
  private String hostname = "your_hostname";

                            // Define name of channel for client to use
  private String channel  = "server_channel";

                            // Define name of queue manager object to connect to
  private String qManager = "your_Q_manager";

                            // define a queue manager object
  private MQQueueManager qMgr;

                // ****************************
                // *      INITIALIZATION      *
                // ****************************
  // When the class is called, this initialisation is done first.

  public void init()
  {
                                        // Set up MQ environment
     MQEnvironment.hostname = hostname;   // Could have put the hostname and
     MQEnvironment.channel  = channel;    // channel string directly here!

  }                                     // end of init


  public void start()
  {
    try {
                // ****************************
                // *         CONNECT          *
                // ****************************
                            // Create a connection to the queue manager
      qMgr = new MQQueueManager(qManager);

                // ****************************
                // *           OPEN           *
                // ****************************
                            // Set up the options on the queue we wish to open...

                            // Note: All MQ Options are prefixed witha

      int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                        MQC.MQOO_OUTPUT ;
```

Figure 46. MQSeries Client for Java Sample Applet - Part 1

```
                    // Note: MQOO_INQUIRE & MQOO_SET are alwa
                             by default.

                    // Now specify the queue that we wish to open,
                    // and the open options...
MQQueue system_default_local_queue =
        qMgr.accessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
                         openOptions,
                         null,           // default queue manager
                         null,           // no dynamic queue name
                         null);          // no alternate user ID

            // ****************************
            // *        MESSAGE           *
            // ****************************
                    // Define a simple MQ message,
                    // and write some text in UTF format..
MQMessage hello_world = new MQMessage();
hello_world.writeUTF("Hello World!");

            // ****************************
            // *          PUT             *
            // ****************************
                    // Specify the message options...
                    // (accept the defaults, same as MQPMO_DEFAULT constant)
MQPutMessageOptions pmo = new MQPutMessageOptions();

                    // Put the message on the queue
system_default_local_queue.put(hello_world,pmo);

            // ****************************
            // *          GET             *
            // ****************************
                    // Get the message back again...
                    // First define a MQ message buffer to receive the
                    // message into
MQMessage retrievedMessage = new MQMessage();
retrievedMessage.messageId = hello_world.messageId;

                    // Set the get message options..
                    // (accept the defaults, same as MQGMO_DEFAULT)
MQGetMessageOptions gmo = new MQGetMessageOptions();

                    // get the message off the queue...
system_default_local_queue.get(retrievedMessage, gmo);
```

Figure 47. MQSeries Client for Java Sample Applet - Part 2

```
                // ****************************
                // *        DISPLAY           *
                // ****************************
                          // Prove we have the message by displaying the
                          // UTF message text.
      String msgText = retrievedMessage.readUTF();
      System.out.println("The message is: " + msgText);

                // ****************************
                // *         CLOSE            *
                // ****************************
                          // Close the queue
      system_default_local_queue.close();

                // ****************************
                // *       DISCONNECT         *
                // ****************************
                          // Disconnect from the queue manager
      qMgr.disconnect();
    }

                // ****************************
                // *     ERROR HANDLING       *
                // ****************************

    // If an error has occured in the above, try to identify what went wrong.

                          // Was it an MQ error?
    catch (MQException ex)
    {
      System.out.println("An MQ error occurred : Completion code " +
                         ex.completionCode +
                         " Reason code " + ex.reasonCode);
    }

                          // Was it a Java buffer space error?
    catch (java.io.IOException ex)
    {
     System.out.println("An error occurred while writing to the message buffer: " ex);
    }

  }                       // end of start

}                         // end of sample
```

Figure 48. MQSeries Client for Java Sample Applet - Part 3

8.3 Why Should I Use the Java Interface?

This section is written for programmers who are familiar with the procedural MQSeries application programming interface as described in the *MQSeries Application Programming Guide*, and shows how to transfer this knowledge to become productive with the MQSeries Java programming interface.

The Java White Paper lists several of the design goals and benefits of Java. The MQSeries Java programming interface makes these benefits available to you as a developer of MQSeries applications:

- The Java programming language is *simple*. There is no need for header files, pointers, structures, unions, and operator overloading. Programs written in Java are simpler to develop and easier to debug than their C and C++ equivalents.
- Java is *object-oriented*. The object-oriented features of Java are comparable to those of C++, but there is no multiple inheritance . Instead, Java uses the concept of an interface.
- Java is inherently *distributed*. The Java class libraries contain a library of routines for coping with TCP/IP protocols such as HTTP and FTP. Java programs can access URLs as easily as accessing a file system.
- Java is *robust*. Java puts a lot of emphasis on early checking for possible problems, dynamic (run-time) checking, and the elimination of situations that are error prone. Java uses a concept of references that eliminates the possibility of overwriting memory and corrupting data.
- Java is *secure*. Java is intended to be run in networked/distributed environments, and a lot of emphasis has been placed on security. Java programs cannot overrun their run-time stack, cannot corrupt memory outside of their process space, and when downloaded via the Internet cannot even read or write local files.
- Java programs are *portable*. There are no implementation-dependent aspects of the Java specification. The Java compiler generates an architecture neutral object file format. The compiled code is executable on many processors, so long as the Java run-time system is present.

When you write your application using the MQSeries Client for Java, users can download the Java byte codes for your program (called applets) from the Internet and run them on their own machines. This means that anyone with access to your Web server can load and run your application with no prior installation needed on their machine. When an update to the program is required, you simply update the copy on the Web server and users automatically receive the latest version the next time they access the applet. This can significantly reduce the costs involved in installing and updating

traditional client applications where a large number of desktops are involved.

If you place your applet on a Web server that is accessible outside of the corporate firewall, then anyone on the Internet can download and use your application. This means that you can get messages into your MQ system from anywhere on the Internet. This opens the door to building a whole new set of Internet accessible service, support and electronic commerce style applications.

8.4 The MQSeries Java Programming Interface

The procedural MQSeries application programming interface is built around 11 verbs:

- MQBACK
- MQCLOSE
- MQCMIT
- MQCONN
- MQDISC
- MQGET
- MQINQ
- MQOPEN
- MQPUT
- MQPUT1
- MQSET

These verbs all take, as a parameter, a handle to the MQSeries object on which they are to operate. Because Java is object-oriented, the Java programming interface turns this around. Your program consists of a set of MQSeries objects, which you act upon by calling methods on those objects, as in the following example.

Using the procedural interface, you disconnect from a queue manager using the call:

```
MQDISC(Hconn, CompCode, Reason)
```

where Hconn is a handle to the queue manager.

In the Java interface, the queue manager is represented by an object of class MQQueueManager. You disconnect from it by calling the disconnect() method on that class.

```
// declare an object of type queue manager
MQQueueManager queueManager;
⋮
// do some stuff...
⋮
// disconnect from the queue manager
queueManager.disconnect();
```

In Java, a *package* is a mechanism for grouping sets of related classes together. To include the MQSeries package in your program add the following line at the top of your source file:

```
import com.ibm.mq.*;
```

As there are no header files, you will find the MQSeries constants in a class called MQC. You refer to these constants in your program using the notation MQC.constant_name. For example:

```
MQC.MQOO_OUTPUT
```

8.4.1 Handling Errors

Methods in the Java interface do not return a completion code and reason code. Instead, they throw an exception whenever the completion code and reason code resulting from an MQ call are not both zero. This simplifies the program logic so that you do not have to explicitly check the return codes after each call to MQ. You can decide at which point in your program you want to deal with the possibility of failure by surrounding your code with try and catch blocks, as in the following example:

```
try {
    myQueue.put(messageA,putMessageOptionsA);
    myQueue.put(messageB,putMessageOptionsB);
}
catch (MQException ex) {
    // This block of code is only executed if one of the two put methods
    // gave rise to a non-zero completion code or reason code.
    System.out.println("An error occurred during the put operation:" +
                       "CC = " + ex.completionCode +
                       "RC = " + ex.reasonCode);
}
```

8.4.2 Operations on Queue Managers

Before connecting to a queue manager, you must take care to set up the MQEnviroment.

The C-based MQ clients rely on environment variables to control the behavior of the MQCONN call. Since Java applets have no access to

environment variables, the Java programming interface includes a class MQEnvironment which allows you to specify the following details that are to be used during the connection attempt.

- Channel name
- Hostname
- Port number
- User ID
- Password

To specify the channel name and hostname use the following code:

```
MQEnvironment.hostname = "host.domain.com";
MQEnvironment.channel  = "java.client.channel";
```

This is equivalent to an MQSERVER environment variable setting of "java.client.channel/TCP/host.domain.com".

Note: The Java client always communicates using TCP/IP, so the protocol specifier is not needed.

By default, the Java client will attempt to connect to port 1414. To specify a different port, use the code:

```
MQEnvironment.port = nnnn;
```

The user ID and password default to blanks. To specify a non-blank user ID or password use the code:

```
MQEnvironment.userID   = "uid";  // equivalent to env var MQ_USER_ID
MQEnvironment.password = "pwd";  // equivalent to env var MQ_PASSWORD
```

Once the MQEnvironment is correctly set up, you can connect to a queue manager simply by creating a new instance of the MQQueueManager class:

```
MQQueueManager queueManager = new MQQueueManager("qMgrName");
```

To disconnect from a queue manager, simply call the disconnect() method on the queue manager:

```
queueManager.disconnect();
```

Calling the disconnect method causes all open queues and processes that you have accessed via that queue manager to be closed. It is good programming practice however to close these resources yourself when you have finished using them. You do this with the close() method.

The commit() and backout() methods on a queue manager replace the MQCMIT and MQBACK calls of the procedural interface.

8.4.3 Accessing Queues and Processes

Queues and process are accessed via the MQQueueManager class. The MQOD (object descriptor structure) has been collapsed into the parameters of these methods. For example, to open a queue on a queue manager queueManager, use the following code:

```
MQQueue queue = queueManager.accessQueue("qName",
                                         MQC.MQOO_OUTPUT,
                                         "qMgrName",
                                         "dynamicQName",
                                         "altUserId");
```

The options parameter is the same as the options parameter in the MQOPEN call.

Note: MQOO_INQUIRE and MQOO_SET are always added to the options specified so that the inquire and set operations are available on any queue.

The accessQueue method returns a new object of class MQQueue.

When you have finished using the queue, you should close it using the close() method, as in the following example:

```
queue.close();
```

New in this version of the MQSeries Client for Java is the ability to create a queue using a new MQQueue constructor. The parameters are exactly the same as for the accessQueue method, with the addition of a queue manager parameter. For example:

```
MQQueue queue = new MQQueue(queueManager,
                            "qName",
                            MQC.MQOO_OUTPUT,
                            "qMgrName",
                            "dynamicQName",
                            "altUserId");
```

Constructing a queue object in this way enables you to write your own subclasses of MQQueue.

To access a process use the accessProcess method in place of accessQueue. This method does not have a dynamic queue name parameter since this does not apply to processes.

The accessProcess method returns a new object of class MQProcess.

When you have finished using the process object you should close it using the close() method, as in the following example:

```
process.close();
```

New in this version of the MQSeries Client for Java is the ability to create a process using a new MQProcess constructor. The parameters are exactly the same as for the accessProcess method, with the addition of a queue manager parameter. Constructing a process object in this way enables you to write your own subclasses of MQProcess.

8.4.4 Handling Messages

You put messages onto queues using the put() method of the MQQueue class and you get messages from queues using the get() method of the MQQueue class. Unlike in the procedural interface, where MQPUT and MQGET put and get arrays of bytes, the Java programming language puts and gets instances of the MQMessage class. The MQMessage class encapsulates the data buffer that contains the actual message data, together with all the MQMD parameters that describe that message.

To build a new message, create a new instance of the MQMessage class and use the writeXXX methods to put data into the message buffer.

When the new message instance is created, all of the MQMD parameters are automatically set to their default values, as defined in the *MQSeries Application Programming Reference*. The put method of MQQueue also takes an instance of the MQPutMessageOptions class as a parameter. This class represents the MQPMO structure.

The following example shows the creation of a message and putting it onto a queue:

```
// Build a new message containing my age followed by name.

MQMessage myMessage = new MQMessage();
myMessage.writeInt(25);

String name = "Adrian Colyer";
myMessage.writeInt(name.length());
myMessage.writeBytes(name);

// Use the default put message options...
MQPutMessageOptions pmo = new MQPutMessageOptions();

// put the message!
queue.put(myMessage,pmo);
```

The get() method of MQQueue returns a new instance of MQMessage, which represents the message just taken from the queue. It also takes an

instance of the MQGetMessageOptions class as a parameter. This class represents the MQPMO structure.

There is no need to specify a maximum message size as the get() method automatically adjusts the size of its internal buffer to fit the incoming message. Use the readXXX methods of the MQMessage class to access the data in the returned message.

The following example shows the getting of a message from a queue:

```
// Get a message from the queue
MQMessage theMessage    = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();  // has default values
queue.get(theMessage,gmo);

// Extract the message data
int age = theMessage.readInt();
int strLen = theMessage.readInt();
byte[] strData = new byte[strLen];
theMessage.readFully(strData,0,strLen);
String name = new String(strData,0);
```

The number format used by the read and write methods can be altered by setting the encoding data member.

The character set to use for reading and writing strings can be altered by setting the characterSet data member.

See the documentation of the MQMessage class class for more details.

A note on reading and writing strings: Using the writeUTF method of MQMessage automatically encodes the length of the string as well as the unicode bytes it contains. When your message is to be read by another Java program (using readUTF()), this is the simplest way to send string information.

8.4.5 Inquire and Set

For many of the common attributes, the classes MQManagedObject, MQQueue, MQProcess and MQQeueManager contain getXXX() and setXXX() methods which allow you to easily get and set their attribute values.

For less common attributes, the MQQueueManager, MQQueue and MQProcess classes all inherit from a class called MQManagedObject. This class defines the inquire() and set() interfaces.

When you create a new queue manager object using the new operator it is automatically opened for inquiry.

When you access a process object using the accessProcess() method, it is automatically opened for inquiry.

When you access a queue object using the accessQueue() method, it is automatically opened for both inquire and set operations. The inquire and set methods take three parameters:

1. selectors array
2. intAttrs array
3. charAttrs array

There is no need for the SelectorCount, IntAttrCount and CharAttr Length parameters found in MQINQ, since the length of an array in Java is always known.

The following example shows how to make an enquiry on a queue:

```
// inquire on a queue
final static int MQIA_DEF_PRIORITY = 6;
final static int MQCA_Q_DESC = 2013;
final static int MQ_Q_DESC_LENGTH = 64;

int[] selectors  = new int[2];
int[] intAttrs   = new int[1];
byte[] charAttrs = new byte[MQ_Q_DESC_LENGTH];

selectors[0] = MQIA_DEF_PRIORITY;
selectors[1] = MQCA_Q_DESC;

queue.inquire(selectors,intAttrs,charAttrs);

System.out.println("Default Priority = " + intAttrs[0]);
System.out.println("Description : " + new String(charAttrs,0));
```

8.4.6 Multithreaded Programs

Multithreaded programs are hard to avoid in Java. Consider a simple program that connects to a queue manager and opens a queue at startup. The program displays a single button on the screen and when the button is selected, it fetches a message from the queue.

Because the Java run-time environment is inherently multithreaded, your application initialization will take place in one thread, and the code that is executed in response to the button selection, executes in a separate thread (the user interface thread).

With the C-based MQSeries client this would cause a problem, since handles cannot be shared across multiple threads. The MQSeries Client for

Java relaxes this constraint, allowing a queue manager object (and its associated queue and process objects) to be shared across multiple threads.

The implementation of the Java client ensures that, for a given connection (queue manager object instance), all access to the target MQSeries queue manager is synchronized. This means that a thread wishing to issue a call to a queue manager is blocked until all other calls in progress for that connection have completed.

If you require simultaneous access to the same queue manager from within your program, create a new queue manager object for each thread requiring concurrent access. This is equivalent to issuing a separate MQCONN call for each thread.

MQ calls inside an applet's stop() and destroy() methods: If you need to make MQ calls inside the stop() or destroy() methods of an applet (for example, to disconnect from a queue manager), then you need to declare your stop and destroy methods as synchronized. For example:

```
public synchronized void stop() {
⋮
}

public synchronized void destroy() {
⋮
}
```

If you do not use the synchronized keyword, the Java virtual machine may choose to suspend your thread when you issue an MQ call (since you will be waiting on I/O). For the special cases of the stop() and destroy() methods (where the applet is being quiesced / terminated) your thread may never be resumed again. The synchronized keyword prevents this from happening by ensuring that the Java virtual machine waits for your MQ call to complete before it can complete its shutdown.

8.4.7 Writing User Exits

The MQSeries Client for Java allows you to provide your own send, receive, and security exits.

To implement an exit, you must define a new Java class that implements the appropriate interface. There are three exit interfaces defined in the MQ package:

1. MQSendExit
2. MQReceiveExit

3. MQSecurityExit

The following sample defines a class that implements all three:

```
class MyMQExits implements MQSendExit, MQReceiveExit, MQSecurityExit {

   // This method comes from the send exit
   public byte[] sendExit(MQChannelExit channelExitParms,
                          MQChannelDefinition channelDefParms,
                          byte agentBuffer[])
   {
   // fill in the body of the send exit here
   }

   // This method comes from the receive exit
   public byte[] receiveExit(MQChannelExit channelExitParms,
                             MQChannelDefinition channelDefParms,
                             byte agentBuffer[])
   {
   // fill in the body of the receive exit here
   }

   // This method comes from the security exit
   public byte[] securityExit(MQChannelExit channelExitParms,
                              MQChannelDefinition channelDefParms,
                              byte agentBuffer[])
   {
   // fill in the body of the security exit here
   }
}
```

Each exit is passed an MQChannelExit and an MQChannelDefinition object instance. These objects represent the MQCXP and MQCD structures defined in the procedural interface.

The agentBuffer parameter contains the data that is about to be sent (in the case of the send exit), or has just been received (in the case of the receive and security exits). There is no need for a length parameter, since the expression agentBuffer.length tells you the length of the array.

- For the send and security exits, your exit code should return the byte array that you wish to be sent to the server.
- For a receive exit, your code should return return the modified data that you wish to be interpreted by the MQSeries Client for Java.

The simplest possible exit body is:

```
{
  return agentBuffer;
}
```

If your program is to run as a downloaded Java applet, you should be aware that under the security restrictions placed upon it you will not be able to read or write any local files. If your exit needs a configuration file, you can place the file on the Web and use the java.net.URL class to download it and examine its contents.

8.5 Compiling MQSeries Java Programs

If you installed the MQSeries Client for Java in a directory called MQJavaClient, the installation process will have created a subdirectory com\ibm\mq of MQJavaClient in which the package files are stored. When you compile your program, your CLASSPATH environment variable must contain a reference to the MQJavaClient directory (and not the com\ibm\mq subdirectory).

To compile a class MyClass.java, you use the command:

```
javac MyClass.java
```

If you are writing an applet (subclass of java.applet.Applet), then you also need to create an HTML file referencing your class before you can run it. A sample HTML file might look as follows:

```
<html>
<body>
<applet code="MyClass.class" width=200 height=400>
</applet>
</body>
</html>
```

You can now run your program either by loading this HTML file into a Java-enabled Web browser, or by using the appletviewer that comes with the Java Development Kit (JDK). To use the applet viewer, enter the command:

```
appletviewer myclass.HTML
```

If you are writing an application (a class that contains a main() method), then you run your program using the Java interpreter instead. In this case, use the command:

```
java MyClass
```

Note: The .class extension is omitted from the class name.

8.6 Tracing MQSeries Java Programs

The MQSeries client for Java includes a trace facility that can be used to produce diagnostic messages if you suspect there might be a problem with the client code. (You will normally only need to use this facility at the request of IBM service.)

Tracing is controlled by the enableTracing and disableTracing methods of the MQEnvironment class. For example:

```
MQEnvironment.enableTracing(2);   // trace at level 2
these commands will be traced
MQEnvironment.disableTracing();   // turn tracing off again
```

The trace is written to the Java console (System.err).

If your program is an application, or you are running it from your local disk using the appletviewer command, you also have the option of redirecting the trace output to a file of your choice. The following code fragment shows an example of how to make the redirection to a file called myapp.trc:

```
import java.io.*;

try {
  FileOutputStream  traceFile = new FileOutputStream("myapp.trc");
  MQEnvironment.enableTracing(2,traceFile);
}
catch (IOException ex) {
  // couldn't open the file, trace to System.err instead
  MQEnvironment.enableTracing(2);
}
```

There are five different levels of tracing:

1. Provides entry, exit and exception tracing.
2. Provides as above plus parameter information.
3. Provides as above plus transmitted and received MQ headers and data blocks.
4. Provides as above plus transmitted and received user message data.
5. Provides as above plus tracing of methods in the Java Virtual Machine.

Tracing at level 5 will only trace methods in the Java Virtual Machine if you run your application using java_g in place of java, or your applet using appletviewer_g instead of appletviewer.

8.7 Class Hierarchy

- class java.lang.Object
  - class java.lang.Character
  - class java.lang.Class
  - interface java.lang.Coneable
  - interface java.io.DataInput
  - interface java.io.DataOutput
  - class java.util.Date
  - class java.util.Dictionary
    - class java.util.Hashtable (implements java.lang.Cloneable)
      - class java.util.Properties
  - interface java.util.Enumeration
  - class java.io.File
  - class java.io.FileDescriptor
  - interface java.io.FilenameFilter
  - class java.io.InputStream
    - class java.io.FileInputStream
    - class java.io.FilterInputStream
      - class java.io.BufferedInputStream
      - class java.io.DataInputStream (implements java.io.java.io.DataInput)
      - class java.io.PushbackInputStream
  - interface com.ibm.mq.**MQC**
  - class com.ibm.mq.**MQChannelDefinition**
  - class com.ibm.mq.**MQChannelExit**
  - class com.ibm.mq.**MQEnvironment**
  - class com.ibm.mq.**MQGetMessageOptions**
  - class com.ibm.mq.**MQMessage** (implements java.io.DataInput and java.io.DataOutput)
  - class com.ibm.mq.**MQManagedObject**
    - class com.ibm.mq.**MQProcess**

    - class com.ibm.mq.**MQQueue**
    - class com.ibm.mq.**MQQueueManager**
- class com.ibm.mq.**MQPutMessageOptions**
- interface com.ibm.mq.**MQReceiveExit**
- interface com.ibm.mq.**MQSecurityExit**
- interface com.ibm.mq.**MQSendExit**
- class java.lang.Math
- class java.lang.Number
    - class java.lang.Double
    - class java.lang.Float
    - class java.lang.Integer
    - class java.lang.Long
- class java.io.OutputStream
    - class java.io.FileOutputStream
    - class java.io.FilterOutputStream
        - class java.io.BufferedOutputStream
        - class java.io.DataOutputStream (implements java.io.″java.io.DataOutput
        - class java.io.PrintStream
- class java.util.Random
- interface java.lang.Runnable
- class java.lang.Runtime
- class java.lang.SecurityManager
- class java.lang.String
- class java.lang.StringBuffer
- class java.util.StringTokenizer (implements java.util.Enumeration)
- class java.lang.System
- class java.lang.Thread (implements java.lang.Runnable)
- class java.lang.ThreadGroup
- class java.lang.Throwable
    - class java.lang.Error

- class java.lang.LinkageError
  - class java.lang.IncompatibleClassChangeError
    - class java.lang.NoSuchMethodError
  - class java.lang. UnsatisfiedLinkError
- class java.lang. ThreadDeath
- class java.lang. VirtualMachineError
  - class java.lang. InternalError

- class java.lang.Exception
  - class java.lang.ClassNotFoundException
  - class java.lang.CloneNotSupportedException
  - class java.io.IOException
    - class java.io.EOFException
    - class java.io.FileNotFoundException
    - class java.io.InterruptedIOException
    - class java.io.UTFDataFormatException
  - class java.lang.IllegalAccessException
  - class java.lang.InstantiationException
  - class java.lang.InterruptedException
  - class com.ibm.mq.**MQException**
  - class java.lang.RuntimeException
    - class java.lang.IllegalArgumentException
      - class java.lang.IllegalThreadStateException
      - class java.lang.NumberFormatException
    - class java.lang.IndexOutOfBoundsException
      - class java.lang.ArrayIndexOutOfBoundsException
      - class java.lang.StringIndexOutOfBoundsException
    - class java.util.NoSuchElementException
    - class java.lang.NullPointerException
    - class java.lang.SecurityException

– class java.util.Vector (includes java.lang.Clonable)

Chapter 9. The Sample Application

This chapter describes the sample Internet application that has been developed in this MQSeries/Java project. We selected this application to demonstrate the functions you can implement using MQSeries, Java, OOP, and the Internet. You may use this demonstration as a base for your applications.

There are numerous opportunities for combining legacy systems with MQSeries and Java. Many legacy applications on many platforms want to be accessed by many users. Users of MQSeries Client for Java programs could be any global citizens who have a Java-enabled Web browser. No MQSeries software is necessary in their workstation. They downloads what they needs from the server, just like HTML files.

9.1 Overview

This prototype sales order entry demonstration utilizes the MQSeries Client for Java software. A Java-enabled Web browser is all you need to run it. No other software is needed in you client workstation. In fact, you don't need a PC. An NC or network computer is all that is needed. The frontend GUI is all Java code that is talking to a backend C program. The C program is automatically triggered to present the user's information via MQSeries messages. Simulate a grocery store on the Web by ordering fruit, vegetables and other items.

The applications consists of:

- An order entry program (applet) written in Java that runs in the end user's workstation.
- An HTML file that calls the applet.
- A server program that processes orders, written in C.
- A file for the setup of the MQSeries environment.

9.1.1 Functions of the Application

We selected a sales order entry application as our example. A server program responds to the Internet user's inquiries and processes his or her orders. Just imagine what it could means for a business when world-wide customers access the product catalog via the Internet and place orders that can be processed by its legacy application interactively.

The user can perform the following functions:

1. Inquire what products are available.

   There are three product groups in our application: fruit, vegetables and others. Each product group can be displayed (downloaded from the server) by selecting the appropriate radio button in the applet.

2. Select one or more products to order.

   From the displayed products, users select one item at a time and type the quantity they wants to order. The order is moved into a product order list:

   - A message is displayed when the quantity exceeds the stock.
   - The user can clear the entire order list or delete specific items.

3. Send the order for processing.

   When the order list is complete the order will be sent to the host for fulfillment processing and order status posting. The host checks the user's credit rating and may reject the order.

4. Inquire about the status of the order.

   The host posts the order status by sending a message to the user. There may be several status messages per order.

Note: All processes are asynchronous. To simplify the program replies are retrieved by a user action such as clicking on a button instead of a background thread checking for messages in the input queues.

9.1.2 Software Used to Develop the Applet

To write and test the sample application we used the following software:

- Web browser:
  - Netscape 2.02 on OS/2 and AIX
  - MS-Explorer on Win95 and WinNT
- Web server:
  - IBM Internet Connection Secure Server on OS/2
  - Apache on AIX
  - MQSeries on OS/2 and AIX
  - MQSeries Client for Java on OS/2 and AIX
  - Java compiler: Sun's JDK and Symantec's VisualCafe

Note: If you run the application from an applet viewer, you must have the Java Developer's Kit (JDK) installed on the client machine.

9.1.3 A View of the Internet

Figure 49 shows the MQSeries Client for Java workstations and the server. In our project we had two servers, an OS/2 and an AIX machine. The application running in the server is a C program that maintains the product lists and processes the orders.

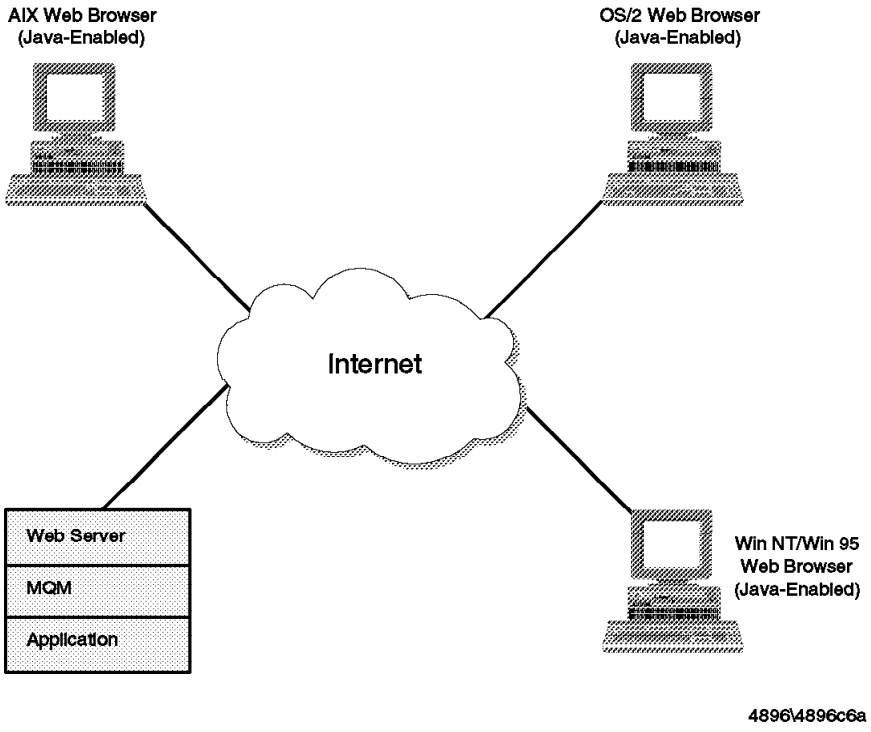


Figure 49. A View of the Internet

9.1.4 The Value of This Application

Even though it is not a real life application, our sales order entry demonstration shows some very valuable points:

- The Internet order entry system is available to the whole world. Users can do business directly with any customer who has a Java-enabled Web browser.
- This application uses the inexpensive existing Internet network.
- Java applets do not require MQSeries programs in the end user's workstation. This feature enables the network computing.

- The order processing program in the server starts automatically. MQSeries′ triggering mechanism takes care of that. Inventory maintenance and credit check is done by the triggered program. That means, users can use their legacy systems with Java applets interactively. It really generates synergy effect of their computer power. Information that has been hidden in the users warehouses can be delivered to anywhere in the world if only the owner wants to do it.
- Java is an object-oriented programming language. So Java applets can be developed easily.
- Java is very good for creating fancy GUIs.

9.2 Program Logic and Message Flow

Figure 50 on page 109 shows the program logic of the application and the queues it uses:

1. The user starts the application from a Web page of a Java-enabled browser.

   The applet including the MQSeries Client for Java code is downloaded into the user′s workstation.

   Figure 51 on page 111 shows an applet after ten items have been entered in the order list.

2. At the beginning the user selects one of three product groups by clicking on one of the three radio buttons in the GUI. The server maintains inventory list for fruit, vegetables and others.
3. When the user clicks on the **Get Product List** button the applet creates a message that contains the ID of the product list.
4. A TCP/IP connection between Web browser and Web server is established. The MQSeries client software then makes a MQI channel connection between MQSeries client and the queue manager in the server.
5. A message requesting one of the three product lists is put into Queue 1.
6. Queue 1 is triggered. Every time a message is put into this queue a trigger message is placed in a trigger queue that is monitored by MQSeries′ trigger monitor. This program then starts the backend C program.

   Note: The trigger monitor must be started in the server. If the server is an OS/2 machine, then the listener has to be started too.

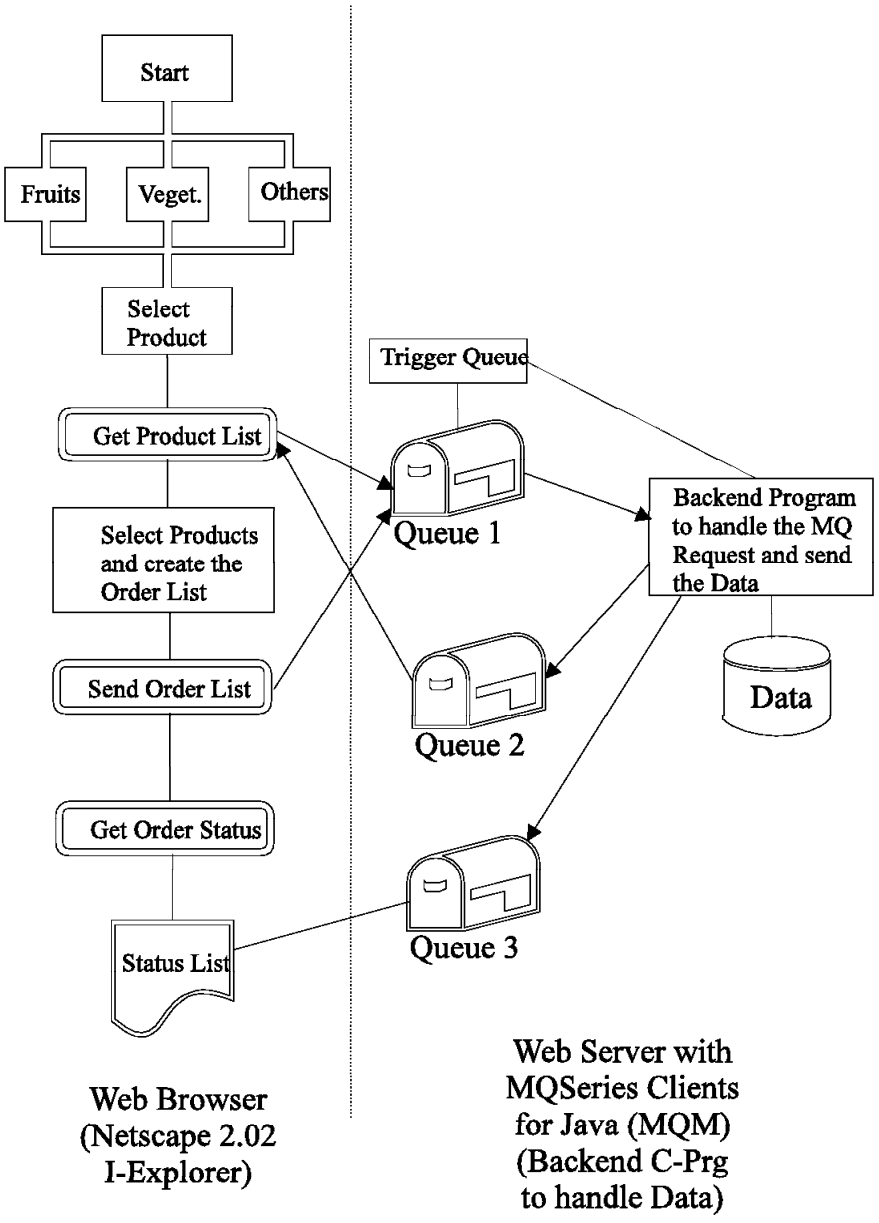


Figure 50. Program Logic and Message Flow

Note: Since the users are MQSeries clients, all queues reside in the server machine. Clients can only work when connection to the server has been established.

7. The backend program processes two message types; both are read from Queue 1:

   - Messages that request a product list.

     One product list is read from disk and put into Queue 2.

   - Messages that contain orders.

     The appropriate product file on disk is updated and the order status is put into Queue 3.

   The server program reads the requested file and moves it into a message that is put into Queue 2. The message contains for each product name, the available quantity and price.

   Note: For this demonstration each list is limited to ten items.

8. After the applet sends the message it issues a get with the wait option. This makes the request/reply synchronous. As soon as the server application puts the message into Queue 2 the waiting applet will remove it from the queue. Its contents is then displayed in the applet.

9. The user selects items from the product list displayed and enters a quantity. Clicking on the **Add** button adds the item to the order list. If the requested quantity exceeds the stock, a message is displayed.

   The user can remove an item from the order list by selecting it and clicking on the **Delete** button. The **Cancel** button clears the entire list.

   The list may contain products from all three lists.

   Note: For this demonstration only ten items can be placed on the order list.

   Each order is given an order number that is maintained by the system, in our case it is the applet.

10. After the order list is completed the user clicks on the **Send Order** button. The program builds the message that can contain one to ten items. Again, a connection to the server is established and the message is put into Queue 1 for processing by the backend C program. This message will trigger the application too. This time the client does not execute a get following the put. Sending the order and receiving the status is asynchronous.

11. The server program updates the product file on the hard disk. It sends a status message indicating that the order will be fulfilled or that one or more items have to be back ordered. These messages are put on Queue 3.

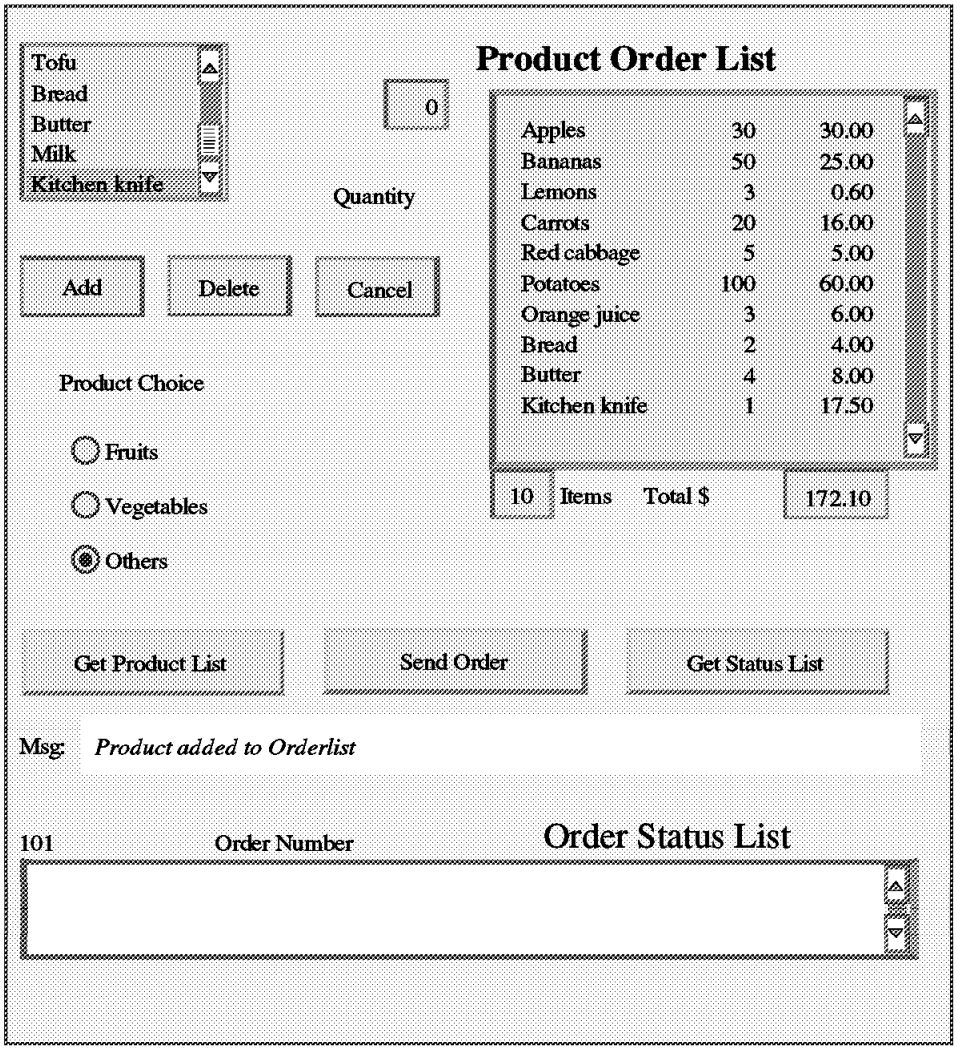


Figure 51. Applet View

12. As soon as the application puts the message on the queue the applet can retrieve it.
13. Clicking on the **Get Status List** push button makes the applet read the messages in Queue 3 and display them in a scrollable status list.

Note: All queues are local queues.

9.3 Design Issues

The following sections help you to understand the design of the application.

9.3.1 Message Flow Diagram

The message flow diagram below shows the message flow between applet and server program and the three local queues.

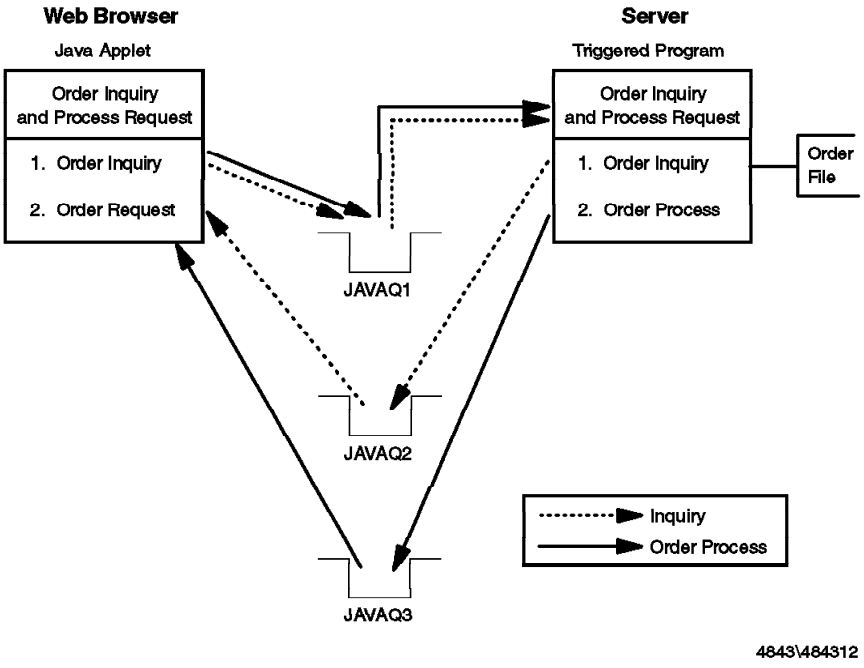


Figure 52. Message Flow Diagram

9.3.2 MQSeries Objects

Table 2. Queues for Demo Application

Queue Name	Triggered	Trigger Depth	Trigger Type
JAVAQ1	yes	1	every
JAVAQ2	no		
JAVAQ3	no		

```
*******************************************************/
* File:  JAVACOMA.TST
*******************************************************/

DEFINE CHANNEL('JavaCH1')  CHLTYPE(SVRCONN)  REPLACE +
                           TRPTYPE(TCP)  MCAUSER(' ')

DEFINE QLOCAL('JavaQ1')  REPLACE +
                         DESCR('Output queue for client') +
                         TRIGTYPE(EVERY) +
                         INITQ (system.default.initiation.queue) +
                         PROCESS (PROCESS.APPL1)

DEFINE PROCESS(PROCESS.APPL1) REPLACE +
                              DESCR('Business logic in server') +
                              APPLTYPE(OS2) +
                              APPLICID('d:\javatest\bl1.exe')

DEFINE QLOCAL('JavaQ2')  REPLACE +
                         DESCR('Input queue for client')

DEFINE QLOCAL('JavaQ3')  REPLACE +
                        DESCR('Input queue for client')
```

Figure 53. MQSeries Objects

9.3.3 Programs

The following table shows the files that comprise the application.

Table 3. Files for Demo Application

Program Name	Description
OrderlistApplet.html	HTML file that loads the applet.
OrderListView.java	The applet (GUI program) running in the client.
FBPutQ1.java	Puts messages on queue 1 for the server program.
FBGetQ2.java	Gets messages (product lists) from queue 2.
GTCGET.java	Gets status messages from queue 3.
bl1.c	The server program the applet communicates with.
bl1.mak	The make file to compile bl1.c.
bl1.def	The definition file for bl1.c.
javacoma.tst	Queue manager objects for the application.

You find listings of these files in:

- Appendix A, “Client Program” on page 117
- Appendix B, “Server Program” on page 135

9.3.4 Message Structure

The message structure is the same for:

- Messages that request a product list.
- Messages that contain a product list.
- Messages that contain orders.

Table 4. Message Structure

Field Number	Type	Length	Description
1	String	4	(a) Product group ID (1-3) (b)Order Number (starting with 100)
2	String	2	Number of items (max. 10)
3	String	15	Product name
4	String	5	Quantity
5	String	5	Price

Notes:

1. For messages that request a product list only field 1 is required. It must contain either 1, 2 or 3.
2. In this demo, all product lists have ten entries.
3. Order messages have in field 1 an order number (100 or higher). This number is automatically assigned by the applet.
4. Fields 3 through 5 can occur up to ten times.

9.3.5 Inventory Files

All three inventory files are flat files and contain ten entries. When the server program detects that they do not exist it will create them automatically.

Note: Fields 3 through 5 occur ten times.

Table 5. Inventory File Structure

Field Number	Type	Length	Description
1	String	4	File ID (product group 1-3)
2	String	4	Number of items (always 10)
3	String	15	Product name
4	String	5	Quantity (initially 999)
5	String	5	Price

9.4 Set Up the Demonstration

- You need a Java-enabled browser such as:
  - Sun's HotJava
  - Netscape Navigator
  - Microsoft Explorer
- The Web server must be up and running:
  - Start it from the Internet Connection Server icon.
  - Start it from the command line by typing `httpd`.
  - Start it automatically from the OS/2 startup folder.
  - Restart it in the Internet Connection Server window.
  - Restarted it from the Configuration and Administration forms.

  Note: Refer to the *IBM Internet Connection Server* manual for more details.
- The queue manager must be up and running:

  ```
  STRMQM [queue_manager_name]
  ```

  Note: Ensure that all MQ objects are created. Refer to the file javacoma.tst in Figure 53 on page 113.
- Ensure that the MQI channel connection between the clients and the queue manager in the server is running.
  - For OS/2 clients, set the environment variable MQSERVER as follows:

    ```
    SET MQSERVER=javamqm/tcp/mqjava.itso.ral.ibm.com
    ```

    Note: mqjava.itso.ral.ibm.com is the TCP/IP hostname of the OS/2 server.

- For UNIX clients, enter the following command:

  ```
  EXPORT MQSERVER=javamqm/tcp/mercury.itso.ral.ibm.com
  ```

  Note: mercury.itso.ral.ibm.com is the TCP/IP hostname of the AIX server.

- On an OS/2 server, make sure that the listener is running:

  ```
  start runmqlsr
  ```

- Ensure that the trigger monitor is started. On an OS/2 server, enter:

  ```
  start runmqtrm
  ```

  Note: There are slight differences in handling the trigger monitor for the different platforms. Refer to chapter 14 of the *MQSeries Application Programming Guide*.

Appendix A. Client Program

A.1 OrderlistApplet.html

```
<applet code=OrderListView.class width=500 height=400>
</applet>
```

A.2 OrderListView.java

```
/*    A basic extension of the java.applet.Applet class     */

 import java.awt.*;           // Include the AWT package
 import java.applet.*;        // Include the applet package
 import java.lang.*;          // Include the language package
 import java.io.*;            // Include the i/o package
 import MQ.*;                 // Include the MQSeries package

 public class OrderListView extends Applet {

         // Definitions for MQSeries

         public String hostname = "mqjava";
         public String channel = "javach1";
         public String qManager = "javamqm";
         public static String msgText;
         public static MQQueueManager qMgr = null;
         public static byte MsgId[];

         // Definitions

         public String  getMsg;         // define the message buffers

         public static String pmsg, omsg, tempStr;

         String fstring = ("000101fruits         0000000000");
         String vstring = ("000201vegetables     0000000000");
         String ostring = ("000301others         0000000000");

         StringBuffer orderMessage;             // for constr.order msg
         int lgStr = 0;                         // length of a string
         public char oChar[] = new char[5];

         int tempQuant, tempPrice;              // temp.Values for calc.
         int c;                                 // counter, # of entries
```

```
     double oPrice = 0.00, total = 0.00;
     double plimit = 500.00;                // price limit
     int qlimit = 0;                        // quantity limit
     int si = 1, j = 0;
     int i1, i2, i3, i4;
     int msgId, nrOfProd;
     int orderNr = 100;

     public String stock[];
     public String price[];

// Method for radio button Fruits
// Get fruits from database

void FruitsRButton_Action (Event event) {
      pmsg = fstring;
}  // end of method

// Method for radio button Vegetables
// Get vegetables from database

void VegetablesRButton_Action(Event event) {
     pmsg = vstring;
}  // end of method

// Method for radio button Others
// Get other items from database

void OthersRButton_Action(Event event) {
     pmsg = ostring;
}  // end of method

// Method for push button Get Procuct List

void productButton_Clicked(Event event) {
     msgArea.setText(" ");
     productChoice.clear();

     FBputQ1 fbput = new FBputQ1();
     fbput.fb_mqput(pmsg);

     FBgetQ2 fbget = new FBgetQ2();
     fbget.fb_mqget();
     getMsg = fbget.msgText;

     msgId = Integer.parseInt(getMsg.substring(0,4));
     nrOfProd = Integer.parseInt(getMsg.substring(4,6));
```

```
     i1 = 6;
     i2 = 21;
     i3 = 26;
     i4 = 31;

     stock = new String[nrOfProd];    // how many in stock
     price = new String[nrOfProd];    // price of one w.dot

     for (int i = 0; i < nrOfProd; i++)
     {
         productChoice.addItem(getMsg.substring(i1,i2));
         stock[i] = getMsg.substring(i2,i3);
         price[i] = getMsg.substring(i3,i4);
         i1 = i1 + 25;
         i2 = i2 + 25;
         i3 = i3 + 25;
         i4 = i4 + 25;
     }  //end of for loop
}  // end of method

// Method for Add push button

void AddButton_Clicked(Event event) {
     msgArea.setText(" ");
     si = productChoice.getSelectedIndex();
     if ( si < 0) {
        msgArea.setText
        ("You didn't select a product");
        return;
     }

     tempQuant = Integer.parseInt(Quantity.getText());
     if (tempQuant < 1) {
        msgArea.setText
        ("The quantity is not set");
        return;
     }

     qlimit = Integer.parseInt(stock·si‘);
     if (tempQuant > qlimit) {
        msgArea.setText
        ("Quantity is higher than Stock: " + qlimit);
        return;
     }

     tempPrice = Integer.parseInt(price·si‘);
     oPrice = (tempQuant * (tempPrice * 0.01));
     total += oPrice;
```

```
     if (total > plimit) {
        msgArea.setText
        ("You are over your creditlimit");
        total -= oPrice;
        return;
     }
     c = productList.countItems()+1;
     if (c > 10 ) {
        msgArea.setText
        ("For this Demo only 10 Products allowed");
        total -= oPrice;
        return;
     }

     quantityList.addItem(Quantity.getText());
     productList.addItem(productChoice.getSelectedItem());
     Quantity.setText ("0");

                                  //   Quantity.
     priceList.addItem(String.valueOf(oPrice));
     orderPriceList.addItem(price·si‘);
     TotalCost.setText(String.valueOf(total));
     Counter.setText(String.valueOf(c));
     msgArea.setText
     ("Product added to Orderlist");
}  // end of method

// Method for Delete push button

void DeleteButton_Clicked(Event event) {
     msgArea.setText(" ");
     si = (productList.getSelectedIndex());
     if (si < 0) {                        // -1 = nothing selected
        msgArea.setText
        ("You didn't select a product in the orderlist to delete");
        return;
     }

     tempQuant = Integer.parseInt(quantityList.getItem(si));
     tempPrice = Integer.parseInt(orderPriceList.getItem(si));
     oPrice = (tempQuant * (tempPrice * 0.01));
     productList.delItem(si);
     quantityList.delItem(si);
     priceList.delItem(si);
     orderPriceList.delItem(si);
     c = productList.countItems();
     Counter.setText(String.valueOf(c));
     total -= oPrice;
```

```
     TotalCost.setText(String.valueOf(total));
     msgArea.setText
     ("Product deleted from Orderlist");
}  // end of method

// Method for Cancel push button

void CancelButton_Clicked(Event event) {
     msgArea.setText(" ");
     if (c < 1) {
        msgArea.setText("There is nothing to cancel");
        return;
     }

     if (j == 0) {
        msgArea.setText
        ("To delete the hole list, klick the cancel button again");
        j = 1;
        return;
     }

     j = 0;

     quantityList.clear();
     productList.clear();
     priceList.clear();
     orderPriceList.clear();
     Counter.setText("");
     TotalCost.setText("0 $");
     total = 0;
     c = 0;
     msgArea.setText ("You can now start a new order");
}  // end of method

// Method for Send push button

void SendButton_Clicked(Event event) {
     msgArea.setText(" ");
     if (c < 1) {
        msgArea.setText
        ("There are no products in your orderlist");
        return;
     }

     for (int i = 0; i < 5; i++) {
         oChar[i] = '0';
     }
```

```
     orderMessage = new StringBuffer(6 + (c * 25));
     orderMessage.append(oChar, 1, 1);
     orderMessage.append(String.valueOf(orderNr));
     tempStr = String.valueOf(c);
     lgStr = tempStr.length();
     orderMessage.append(oChar, 1, (2-lgStr));
     orderMessage.append(String.valueOf(c));

     for (int i = 0; i < c; i++)
     {
          orderMessage.append(productList.getItem(i));
          lgStr = quantityList.getItem(i).length();
          orderMessage.append(oChar, 1, (5-lgStr));
          orderMessage.append(quantityList.getItem(i));
          lgStr = orderPriceList.getItem(i).length();
          orderMessage.append(oChar, 1, (5-lgStr));
          orderMessage.append(orderPriceList.getItem(i));

     }  //end of for loop

     omsg = orderMessage.toString();

     // now create the mqput message

     quantityList.clear();
     productList.clear();
     priceList.clear();
     orderPriceList.clear();
     Counter.setText("");
     TotalCost.setText("0 $");
     total = 0;

     FBputQ1 fbput = new FBputQ1();
     fbput.fb_mqput(omsg);

     msgArea.setText
     ("Your Order " + orderNr + " is forwarded to the HOST");
     orderNr += 1;
     orderNumber.setText(String.valueOf(orderNr));
}  // end of method

// Method for push button Get Status List

void statusButton_Clicked(Event event) {
     msgArea.setText(" ");
     GTCget gtcg = new GTCget();
     gtcg.my_mqget();
     statusList.addItem(gtcg.msgText);
```

```
}   // end method

// Initialization method

public void init() {

     super.init();
     QMenv_init();

     pmsg = fstring;      // because fruits are enabled by default

     // setup the MQ environment
     //{{INIT_CONTROLS
     setLayout(null);
     addNotify();
     resize(440,414);
     setFont(new Font("TimesRoman", Font.BOLD, 14));
     setForeground(new Color(16711680));
     setBackground(new Color(12632256));
     label1 = new java.awt.Label("Product Order List");
     label1.reshape(210,8,224,30);
     label1.setFont(new Font("TimesRoman", Font.BOLD, 24));
     label1.setForeground(new Color(0));
     add(label1);
     productList = new java.awt.List(0,false);
     add(productList);
     productList.reshape(210,38,126,180);
     productList.setForeground(new Color(0));
     Quantity = new java.awt.TextField(2);
     Quantity.setText("0");
     Quantity.reshape(161,30,35,33);
     Quantity.setFont(new Font("TimesRoman", Font.BOLD, 14));
     Quantity.setForeground(new Color(0));
     Quantity.setBackground(new Color(16777215));
     add(Quantity);
     label2 = new java.awt.Label("Quantity");
     label2.reshape(140,68,70,22);
     label2.setFont(new Font("TimesRoman", Font.BOLD, 16));
     label2.setForeground(new Color(0));
     add(label2);
     AddButton = new java.awt.Button("Add");
     AddButton.reshape(7,113,56,26);
     AddButton.setFont(new Font("TimesRoman", Font.BOLD, 14));
     AddButton.setForeground(new Color(0));
     add(AddButton);
     DeleteButton = new java.awt.Button("Delete");
     DeleteButton.reshape(70,113,56,26);
     DeleteButton.setFont(new Font("TimesRoman", Font.BOLD, 14));
```

```
DeleteButton.setForeground(new Color(0));
add(DeleteButton);
CancelButton = new java.awt.Button("Cancel");
CancelButton.reshape(133,113,60,26);
CancelButton.setFont(new Font("TimesRoman", Font.BOLD, 14));
CancelButton.setForeground(new Color(0));
add(CancelButton);
label3 = new java.awt.Label("Product Choice");
label3.reshape(20,150,150,25);
label3.setFont(new Font("TimesRoman", Font.BOLD, 18));
label3.setForeground(new Color(0));
add(label3);
Group1 = new CheckboxGroup();
FruitsRButton = new java.awt.Checkbox("Fruits", Group1, true);
FruitsRButton.reshape(35,180,105,20);
FruitsRButton.setFont(new Font("TimesRoman", Font.BOLD, 12));
FruitsRButton.setForeground(new Color(0));
FruitsRButton.setBackground(new Color(12632256));
add(FruitsRButton);
VegetablesRButton = new java.awt.Checkbox("Vegetables", Group1, false)
VegetablesRButton.reshape(35,203,105,20);
VegetablesRButton.setFont(new Font("TimesRoman", Font.BOLD, 12));
VegetablesRButton.setForeground(new Color(0));
add(VegetablesRButton);
OthersRButton = new java.awt.Checkbox("Others", Group1, false);
OthersRButton.reshape(35,225,105,20);
OthersRButton.setFont(new Font("TimesRoman", Font.BOLD, 12));
OthersRButton.setForeground(new Color(0));
add(OthersRButton);
OthersRButton = new java.awt.Checkbox("Others", Group1, false);
OthersRButton.reshape(35,225,105,20);
OthersRButton.setFont(new Font("TimesRoman", Font.BOLD, 12));
OthersRButton.setForeground(new Color(0));
add(OthersRButton);
sendButton = new java.awt.Button("Send Order");
sendButton.reshape(157,263,125,25);
sendButton.setFont(new Font("TimesRoman", Font.BOLD, 14));
sendButton.setForeground(new Color(0));
add(sendButton);
label4 = new java.awt.Label("Total $");
label4.reshape(294,225,71,22);
label4.setFont(new Font("TimesRoman", Font.BOLD, 18));
label4.setForeground(new Color(0));
add(label4);
TotalCost = new java.awt.TextField();
TotalCost.setEditable(false);
TotalCost.reshape(364,218,70,31);
TotalCost.setFont(new Font("TimesRoman", Font.BOLD, 16));
```

```
add(TotalCost);
quantityList = new java.awt.List(0,false);
add(quantityList);
quantityList.reshape(336,38,42,180);
quantityList.setForeground(new Color(0));
priceList = new java.awt.List(0,false);
add(priceList);
priceList.reshape(378,38,56,180);
productChoice = new java.awt.List(0,false);
add(productChoice);

productChoice.reshape(8,8,115,97);
productChoice.setForeground(new Color(0));
productChoice.setBackground(new Color(16777215));
Counter = new java.awt.TextField();
Counter.reshape(210,218,28,30);
Counter.setForeground(new Color(0));
Counter.setBackground(new Color(8421504));
add(Counter);
label5 = new java.awt.Label("Items");
label5.reshape(240,230,50,15);
label5.setFont(new Font("Dialog", Font.BOLD, 12));
label5.setForeground(new Color(0));
add(label5);
statusList = new java.awt.List(0,false);
add(statusList);
statusList.reshape(10,350,430,60);
statusList.setForeground(new Color(16711935));
label6 = new java.awt.Label("Order Status List");
label6.reshape(259,323,168,23);
label6.setFont(new Font("Dialog", Font.BOLD, 18));
label6.setForeground(new Color(0));
add(label6);
productButton = new java.awt.Button("Get Product List");
productButton.reshape(14,263,125,25);
productButton.setForeground(new Color(0));
add(productButton);
label7 = new java.awt.Label("Msg:");
label7.reshape(14,293,49,23);
label7.setFont(new Font("TimesRoman", Font.BOLD, 16));
label7.setForeground(new Color(0));
add(label7);
msgArea = new java.awt.Label("");
msgArea.reshape(63,293,371,23);
msgArea.setFont(new Font("TimesRoman", Font.BOLD]Font.ITALIC, 16));
msgArea.setForeground(new Color(16711935));
msgArea.setBackground(new Color(16777215));
add(msgArea);
```

```
    statusButton = new java.awt.Button("Get Status List");
    statusButton.reshape(300,263,125,25);
    statusButton.setForeground(new Color(0));
    add(statusButton);
    label8 = new java.awt.Label("act.Order Number");
    label8.reshape(70,330,160,20);
    label8.setFont(new Font("TimesRoman", Font.BOLD, 16));
    label8.setForeground(new Color(0));
    add(label8);
    orderNumber = new java.awt.Label("100");
    orderNumber.reshape(10,330,45,20);
    orderNumber.setFont(new Font("TimesRoman", Font.BOLD, 16));
    orderNumber.setForeground(new Color(0));
    add(orderNumber);
    orderPriceList = new java.awt.List(0,false);
    orderPriceList.hide();
    orderPriceList.disable();
    add(orderPriceList);
    orderPriceList.reshape(161,210,7,8);
    orderPriceList.setForeground(new Color(0));
    //}}
}  // end of init method

// Events

public boolean handleEvent(Event event) {
  if (event.target == AddButton && event.id == Event.ACTION_EVENT) {
     AddButton_Clicked(event);
  }
  if (event.target == sendButton && event.id == Event.ACTION_EVENT) {
     SendButton_Clicked(event);
  }
  if (event.target == CancelButton && event.id == Event.ACTION_EVENT) {
     CancelButton_Clicked(event);
     }
  if (event.target == DeleteButton && event.id == Event.ACTION_EVENT {
     DeleteButton_Clicked(event);
  }
  if (event.target == VegetablesRButton && event.id == Event.ACTION_EVENT) {
     VegetablesRButton_Action(event);
  }
  if (event.target == OthersRButton && event.id == Event.ACTION_EVENT) {
     OthersRButton_Action(event);
  }
  if (event.target == FruitsRButton && event.id == Event.ACTION_EVENT) {
      FruitsRButton_Action(event);
  }
  if (event.target == productButton && event.id == Event.ACTION_EVENT) {
```

```
      productButton_Clicked(event);
    }
    if (event.target == statusButton && event.id == Event.ACTION_EVENT) {
      statusButton_Clicked(event);
    }
    return super.handleEvent(event);
}

// Controls

//{{DECLARE_CONTROLS
java.awt.Label label1;
java.awt.List productList;
java.awt.TextField Quantity;
java.awt.Label label2;
java.awt.Button AddButton;
java.awt.Button DeleteButton;
java.awt.Button CancelButton;
java.awt.Label label3;
java.awt.Checkbox FruitsRButton;
CheckboxGroup Group1;
java.awt.Checkbox VegetablesRButton;
java.awt.Checkbox OthersRButton;
java.awt.Button sendButton;
java.awt.Label label4;
java.awt.TextField TotalCost;
java.awt.List quantityList;
java.awt.List priceList;
java.awt.List productChoice;
java.awt.TextField Counter;
java.awt.Label label5;
java.awt.List statusList;
java.awt.Label label6;
java.awt.Button productButton;
java.awt.Label label7;
java.awt.Label msgArea;
java.awt.Button statusButton;
java.awt.Label label8;
java.awt.Label orderNumber;
java.awt.List orderPriceList;
//}}

// Part of init method for OrderListView

public void QMenv_init() {
     MQEnvironment.hostname = hostname;
     MQEnvironment.channel = channel;
}  // end of method
```

```
    // Start method called when applet starts to execute
    // A good time to open the Queue manager

    public void start() {
         try {
             // Create a connection to the queue manager
             qMgr = new MQQueueManager(qManager);
         }

         catch (MQException ex) {
             showStatus("Unable to connect to the Queue Manager.");
             mqclose();
         }
    }  // end of method

    // Stop method

    public void stop() {
         mqclose();
    } // end of method

    // mqclose method

    public void mqclose() {
         try {
              if (qMgr != null)
                  qMgr.disconnect();
              }
         catch (MQException ex) {
              showStatus("Error on Disconnect" + ex.reasonCode );
         }
    } // end of method
}
```

A.3 FBPUTQ1.java

```
// MQSeries Client for Java sample application

 import MQ.*;                 // Inlude the MQ package
 import java.io.*;

 public class FBputQ1 extends java.applet.Applet {

        // Definitions

        private String hostname = "mqjava";     // name of host
```

```
      private String channel  = "javach1";    // name of channel
      private String qManager = "javamqm";    // name of queue manager

   // private MQQueueManager qMgr;            // queue manager object
      private static MQQueue java_q1;         // queue 1 for put
      private static MQMessage putmsg;        // the put message
      private static MQPutMessageOptions pmo; // Put Message Options

      String msgText;

// Init method

public FBputQ1() {
   // Set up MQ environment
   MQEnvironment.hostname = hostname;
   MQEnvironment.channel  = channel;
}  // end of init

// Method to put message on javaq1

public void fb_mqput(String msgText) {
  try {
      // open a queue for mqput output ...
      int openOptions = MQC.MQOO_OUTPUT  |
                        MQC.MQOO_INQUIRE |
                        MQC.MQOO_SET;

      java_q1 = OrderListView.qMgr.accessQueue("javaq1",
                           openOptions,
                           null,          // this queue manager
                           null,          // no dynamic queue name
                           null);         // no alternate user ID

      putmsg = new MQMessage();         // get a new message
      putmsg.writeString(msgText);      // fill the buffer
      pmo = new MQPutMessageOptions();  // accept the defaults
      java_q1.put(putmsg, pmo);         // put message on queue

      // Save the messageId for mutiuser order reads
      OrderListView.MsgId = putmsg.messageId;

      java_q1.close();                 // Close the queue

  } // end of try *********

  catch (MQException ex) {
      if ( ex.reasonCode == 2002 ) {
          System.out.println("allready connected");
```

```
        }
        else {
           System.out.println("An MQ error occurred : Completion code " +
                              ex.completionCode +
                              " Reason code " + ex.reasonCode);
        }
     }

     catch (java.io.IOException ex) {
         System.out.println("Error when writing to the messagebuffer: " +         " +
                            ex);
     }

   } // end of method
}
```

A.4 FBGETQ2.java

```
// MQSeries Client for Java sample application

 import MQ.*;                 // Inlude the MQ package
 import java.io.*;

 public class FBgetQ2 extends java.applet.Applet {

        // Definitions

        private String hostname = "mqjava";     // name of host
        private String channel  = "javach1";    // name of channel
        private String qManager = "javamqm";    // name of queue manager

     // private MQQueueManager qMgr;            // queue manager object
        private static MQQueue java_q2;         // queue 1 for get
        private static MQMessage getmsg;        // the get message
        private static MQGetMessageOptions gmo; // Get Message Options

        String msgText;

   // Init method

   public FBgetQ2() {
      // Set up MQ environment
      MQEnvironment.hostname = hostname;
      MQEnvironment.channel  = channel;
   }  // end of init FBgetQ2

   // Method to get a message from javaq2
```

```
  public void fb_mqget() {
     try {
        // open a queue...
        int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                          MQC.MQOO_INQUIRE |
                          MQC.MQOO_SET;

        java_q2 = OrderListView.qMgr.accessQueue("javaq2",
                              openOptions,
                              null,           // this q manager
                              null,           // no dynamic q name
                              null);          // no alternate user id

        getmsg = new MQMessage();          // create a new message

        // Set MsgId to put value to allow muti-user access
        getmsg.messageId = OrderListView.MsgId;
//
        gmo = new MQGetMessageOptions();   // accept option defaults
        gmo.options = MQC.MQGMO_WAIT;      // set wait option
        gmo.waitInterval = 30000;          // set wait to 30 sec
        java_q2.get(getmsg,gmo,3000);      // max message size = 3000
        msgText = getmsg.readLine();       // get the message

        java_q2.close();                   // Close the queue

     }  // end of try

     catch (MQException ex) {
       if ( ex.reasonCode == 2033 || ex.reasonCode == 2002 ) {
          System.out.println ("No more messages, try later");
       }
       else {
          System.out.println ("An MQ error occurred : Completion code " +
                              ex.completionCode +
                              " Reason code " + ex.reasonCode);
       }
     }   // end of catch MQException

     catch (java.io.IOException ex) {
       System.out.println ("Error when writing to the message buffer: " +
                           ex);
     }  // end of catch
  }     // end of start of fb_mqget
}       // end of FBgetQ2
```

A.5 GTCGET.java

```
// MQSeries Client for Java sample application

 import MQ.*;                 // Inlude the MQ package
 import java.io.*;

 public class GTCget extends java.applet.Applet {

       // Definitions

       private String hostname = "mqjava";     // name of host
       private String channel  = "javach1";    // name of channel
       private String qManager = "javamqm";    // name of queue manager

    // private MQQueueManager qMgr;            // queue manager object
       String msgText;

  // Init method

  public GTCget() {
      // Set up MQ environment
      MQEnvironment.hostname = hostname;
      MQEnvironment.channel  = channel;
  }   // end of init

  // Method to get a message from javaq3

  public void my_mqget() {
      try {
         // open a queue...
         int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                           MQC.MQOO_OUTPUT |
                           MQC.MQOO_INQUIRE |
                           MQC.MQOO_SET;

         MQQueue system_default_local_queue =
             OrderListView.qMgr.accessQueue("javaq3",
                              openOptions,
                              null,           // this q manager
                              null,           // no dynamic q name
                              null);          // no alternate user id

         // get the message ...
         MQMessage retrievedMessage = new MQMessage();

         // Set MsgId to put value to allow muti-user access
```

```
          retrievedMessage.messageId = OrderListView.MsgId;
//
          MQGetMessageOptions gmo = new MQGetMessageOptions();

          system_default_local_queue.get(retrievedMessage,
                                         gmo,
                                         100);   // max message size

          // Display the message text
          msgText = retrievedMessage.readLine();
          System.out.println("Order Status is " + msgText);

          // Close the queue
          system_default_local_queue.close();

       } // end of try

       catch (MQException ex) {
            if ( ex.reasonCode == 2033 ]] ex.reasonCode == 2002 ) {
               System.out.println("Last Order Status displayed");
                 msgText = "Last Order Status displayed";
             }
             else
             System.out.println ("An MQ error occurred: Completion code " +
                                 ex.completionCode +
                                 " Reason code " + ex.reasonCode);
       }

       catch (java.io.IOException ex) {
           System.out.println
              ("Error occurred when writing to the message buffer: "  +
               ex);
       }

  } // end of start of my_mqget
}
```

Appendix B. Server Program

This appendix contains the source for the business logic running in the server and files to compile it:

- BL1.C is the source code, written in C.
- Bl1.MAK is the make file for the IBM VisualAge C++ compiler.
- BL1.DEF is the definition file.

B.1 Business Logic BL1.C

```
/*********************************************************************/
/*                                                                   */
/* BL1:   Business Logic for Java Demonstration                      */
/*                                                                   */
/*********************************************************************/
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define  INCL_DOSMISC
#define  INCL_DOSPROCESS
/*****************************************************************/
/*                   Subroutines                                 */
/*****************************************************************/
int  CHECK_FILE  (void);
int  READ_FILE   (int);
void UPDATE_FILE (void);
int  OPEN_QUEUE  (int);
/*****************************************************************/
/*                   Definitions for MQSeries                    */
/*****************************************************************/
#include  <cmqc.h>                    //  MQ header file
MQLONG    CompCode;                   //  Return codes
MQLONG    Reason;                     //  Return codes
MQHCONN   Hcon;                       //  Handle to queue manager
char      inputq[]  = "javaq1";       //  Input queue name
MQHOBJ    Hinput;                     //              handle
MQGMO     MQgmo = {MQGMO_DEFAULT};    //              get message options
MQOD      odin  = {MQOD_DEFAULT};     //              object descriptor
char      outputq1[] = "javaq2";      //  Output queue name
char      outputq2[] = "javaq3";      //  Output queue name
MQHOBJ    Houtput;                    //               handle
MQPMO     MQpmo = {MQPMO_DEFAULT};    //               put message options
MQOD      odout = {MQOD_DEFAULT};     //               object descriptor
```

```
MQMD      MQmd  = {MQMD_DEFAULT};     //  Message Descriptor
MQLONG    O_options;                  //  MQOPEN options
/*************************************************************/
/*              Inventory for demo program                   */
/*************************************************************/
#define ITEMS  10                     // Length of the following tables
struct  items  { char item[16];       // Product name
                 char price[6]; };    // Product price
                                      // Table 1: Fruits
struct  items  Fruit[ITEMS] = {  {"Apples         ","00100" },    /*
                                 {"Bananas        ","00050" },    /*  2 */
                                 {"Coconuts       ","00200" },    /*  3 */
                                 {"Grapes         ","00150" },    /*  4 */
                                 {"Lemons         ","00020" },    /*  5 */
                                 {"Oranges        ","00100" },    /*  6 */
                                 {"Peaches        ","00200" },    /*  7 */
                                 {"Pears          ","00200" },    /*  8 */
                                 {"Strawberries   ","00360" },    /*  9 */
                                 {"Watermelon     ","00040" } };  /* 10 */
                                      // Table 2: Vegetables
struct  items  Veggy[ITEMS] = {  {"Broccoli       ","00120" },    /*
                                 {"Carrots        ","00080" },    /*  2 */
                                 {"Cucumbers      ","00050" },    /*  3 */
                                 {"Lettuce        ","00150" },    /*  4 */
                                 {"Red cabbage    ","00100" },    /*  5 */
                                 {"Leeks          ","00300" },    /*  6 */
                                 {"Potatoes       ","00060" },    /*  7 */
                                 {"Okra           ","00120" },    /*  8 */
                                 {"Green beans    ","00120" },    /*  9 */
                                 {"Zuchini        ","00090" } };  /* 10 */
                                      // Table 3: Other products
struct  items  Other[ITEMS] = {  {"Coca Cola      ","00100" },    /*
                                 {"Orange juice   ","00200" },    /*  2 */
                                 {"Perrier water  ","00300" },    /*  3 */
                                 {"Olive Oil      ","01200" },    /*  4 */
                                 {"Venegar        ","00400" },    /*  5 */
                                 {"Tofu           ","00100" },    /*  6 */
                                 {"Bread          ","00200" },    /*  7 */
                                 {"Butter         ","00200" },    /*  8 */
                                 {"Milk           ","00120" },    /*  9 */
                                 {"Kitchen knife  ","01750" } };  /* 10 */
```

```
/*************************************************************/
/*             Definitions for inventory files               */
/*************************************************************/
char      fnfruit[]   = "fruit.dat";  // File names
char      fnveggy[]   = "veggies.dat";
char      fnother[]   = "other.dat";
char      fn[20];                     // Current file name
FILE      *fp;
char      quantity[6] = "00999";      // Default quantity
char      file_id[4];                 // File ID
char      file_items[2];              // Number of products in file
char      file_product[16];           // Product name
char      file_quantity[6];           // Quantity on hand
char      file_price[6];              // Price from product file

typedef   struct _MSG1                // Structure for product files
{
  MQCHAR  ID[4];                      // ID ( 1, 2 or 3)
  MQCHAR  count[2];                   // Number of products
  MQCHAR  message[500];               // Up to 20 product entries
} MSG1;
MSG1      msg1;                       // Buffer containing one file
int       file_number;                // File ID (1, 2 or 3);
/*************************************************************/
/*           Input / Output Buffers and Work Areas           */
/*************************************************************/
char      szMsgId[25];                // Message ID (MQ)
char      szCorrelId[20];             // Correlation ID (MQ)

MQBYTE    buffer[1000];               // Message buffer
MQLONG    buflen;                     // Buffer length
MQLONG    messlen;                    // Message length (received)
int       recordID;                   // ID in message from client
int       count;                      // Number of items in msg from client
int       offset;                     // Offset in message buffer

char      order_name[16];             // Name of product from order ms
char      order_quantity[6];          // Quantity ordered

char      wkfld[11];                  // Work fields
int       i, ij, ik, il;
int       records;
unsigned int length;
char      *loc;
int       nogo;
```

```
/*************************************************************************/
/*                                                                       */
/*                        P r o g r a m                                  */
/*                                                                       */
/*************************************************************************/
int main( )
{
   printf("Program BL1 starting...\n");
   if (CHECK_FILE () == 1) exit (1);  // Create inventory file if it
                                      // does not exists.
/*************************************************************/
/*   Connect to queue manager                                */
/*************************************************************/
   MQCONN(" ",                        // default queue manager
          &Hcon,                      // connection handle
          &CompCode,                  // completion code
          &Reason);                   // reason code
   if (CompCode == MQCC_FAILED)
   {
     printf("MQCONN ended with reason code %ld\n", Reason);
     exit(Reason);
   }

/*************************************************************/
/*   Open input queue for read                               */
/*************************************************************/
   strcpy(odin.ObjectName,inputq);    // Name of input queue
   O_options = MQOO_INPUT_AS_Q_DEF    //  (open queue for input
             + MQOO_FAIL_IF_QUIESCING;//   but not if MQM stopping)

   MQOPEN(Hcon,                       // Connection handle
          &odin,                      // Object descriptor for queue
          O_options,                  // Open options
          &Hinput,                    // Handle to queue
          &CompCode, &Reason);        // Return codes
   if (Reason != MQRC_NONE)
       printf("MQOPEN ended with reason code %ld\n", Reason);
   if (CompCode == MQCC_FAILED)
   {
     printf("Error: unable to open queue %s for input\n",inputq);
     exit(Reason);
   }
```

```
/*****************************************************************/
/*   Loop to read messages from input queue                      */
/*****************************************************************/
   CompCode = MQCC_OK;                // Loop control
   while (CompCode != MQCC_FAILED)
   {
     buflen = sizeof(buffer) - 1;     // Buffer size for GET
     MQgmo.Options = MQGMO_WAIT;      // Wait for next message
     MQgmo.WaitInterval = 10000;      //      10 second max. wait
     memcpy(MQmd.MsgId, MQMI_NONE, sizeof(MQmd.MsgId));
     memcpy(MQmd.CorrelId, MQCI_NONE, sizeof(MQmd.CorrelId));

     MQGET (Hcon,                     // Connection handle
            Hinput,                   // Object handle to queue
            &MQmd,                    // Message descriptor
            &MQgmo,                   // Get message options
            buflen,                   // Buffer length
            buffer,                   // Message buffer
            &messlen,                 // Message length
            &CompCode, &Reason);      // Return codes
     if (Reason != MQRC_NONE) {
       if (Reason == MQRC_NO_MSG_AVAILABLE)
           printf("No more messages\n");
       else {
         printf("MQGET ended with reason code %ld\n", Reason);
         if (Reason == MQRC_TRUNCATED_MSG_FAILED)
             CompCode = MQCC_FAILED;
       }
     }
     /*************************************************************/
     /*   Display each message received                           */
     /*************************************************************/
     if (CompCode != MQCC_FAILED)
     {
       length = strlen (MQmd.MsgId);         // Message ID
       loc    = strchr(MQmd.MsgId,' ');
       if (loc != NULL) length = loc - MQmd.MsgId;
       strncpy (szMsgId,MQmd.MsgId,length);
       szMsgId[length] = '\0';
       length = strlen (MQmd.CorrelId);      // Correlation ID
       loc    = strchr(MQmd.CorrelId,' ');
       if (loc != NULL) length = loc - MQmd.CorrelId;
       strncpy (szCorrelId,MQmd.CorrelId,length);
       szCorrelId[length] = '\0';
       buffer[messlen] = '\0';               // Add terminator
       printf ("Received MsgId=%s, CorrelId=%s, Length=%d\n",
                szMsgId, szCorrelId, messlen);
       printf("<%s>\n", buffer);
```

```
/*********************************************************/
/*   Obtain message ID and number of products in message   */
/*********************************************************/
  ik = 0;
  for (ij = 0; ij < 4; ij++) {          // Possible blanks
      msg1.ID[ij] = buffer[ij];         // copy to outp
      if (buffer[ij] != ' ') {
          wkfld[ik] = buffer[ij];
          ik++;
      }
  }
  wkfld[ik] = '\0';
  recordID = atoi(wkfld);               // Order number
  wkfld[0] = buffer[4];
  wkfld[1] = buffer[5];
  wkfld[2] = '\0';
  count    = atoi(wkfld);               // Number of products
  printf("Received record ID %d,  items=%d\n", recordID, count);
/*********************************************************/
/*   ID = 1 or 2 or 3:                                   */
/*   Read inventory from disk and send to client         */
/*********************************************************/
  if (recordID == 1 ||                  // get fruit
      recordID == 2 ||                  // get vegetable
      recordID == 3   ) {               // get others
      if (READ_FILE (recordID) == 1) exit(1);
      /*****************************************************/
      /*     Send file as message to requestor             */
      /*****************************************************/
      if (OPEN_QUEUE(1) == 1) exit(1);
      MQmd.Persistence = MQPER_NOT_PERSISTENT;
      MQPUT  (Hcon, Houtput, &MQmd, &MQpmo,   // Output and input
              256, &msg1,                     // use same MQmd
              &CompCode, &Reason);
      if (Reason != MQRC_NONE)
          printf("MQPUT ended with reason code %ld\n", Reason);
      else printf("File %s sent.\n", fn);
      fclose(fp);
      /*****************************************************/
      /*   Close output queue                             */
      /*****************************************************/
      MQCLOSE (Hcon,                    // Connection handle
               &Houtput,                // Object handle for queue
               MQCO_NONE,               // No close options
               &CompCode, &Reason);     // Return codes
      if (Reason != MQRC_NONE)
          printf("MQCLOSE ended with reason code %ld\n", Reason);
  }
```

```
/*********************************************************/
/*   ID > 99:                                            */
/*   Process an order                                    */
/*********************************************************/
  else
  if (recordID > 99) {                  // Order messsage

     printf ("Process order %d with %d items\n",recordID, count);
     nogo = 0;
     UPDATE_FILE ();

     if (OPEN_QUEUE(2) == 1) exit(1);

     MQmd.Persistence = MQPER_NOT_PERSISTENT;
     sprintf (buffer,"Order %d with %d items processed.",recordID, count);
     length = strlen(buffer);
     MQPUT  (Hcon, Houtput, &MQmd, &MQpmo,
             length, buffer,
             &CompCode, &Reason);
     if (Reason != MQRC_NONE)
         printf("MQPUT ended with reason code %ld\n", Reason);
     if (nogo != 0) {
        sprintf(buffer," %d item(s) on back order (%s).",
                nogo, recordID);
        length = strlen(buffer);
        MQPUT  (Hcon, Houtput, &MQmd, &MQpmo,
                length, buffer,
                &CompCode, &Reason);
        if (Reason != MQRC_NONE)
            printf("MQPUT ended with reason code %ld\n", Reason);
     }
      /*************************************************/
      /*   Close output queue                          */
      /*************************************************/
      MQCLOSE (Hcon,                    // Connection handle
               &Houtput,                // Object handle for queue
               MQCO_NONE,               // No close options
               &CompCode, &Reason);     // Return codes
      if (Reason != MQRC_NONE)
          printf("MQCLOSE ended with reason code %ld\n", Reason);
  }
/*********************************************************/
/*   Wrong record ID                                     */
/*********************************************************/
  else {
```

```
          if (OPEN_QUEUE(2) == 1) exit(1);
          strcpy (buffer," Wrong message ID --- discarded.");
          length = strlen(buffer);
          MQPUT  (Hcon, Houtput, &MQmd, &MQpmo,
                  length, buffer,
                  &CompCode, &Reason);
          if (Reason != MQRC_NONE)
              printf("MQPUT ended with reason code %ld\n", Reason);
          MQCLOSE (Hcon,                    // Connection handle
                   &Houtput,                // Object handle for queue
                   MQCO_NONE,               // No close options
                   &CompCode, &Reason);     // Return codes
          if (Reason != MQRC_NONE)
              printf("MQCLOSE ended with reason code %ld\n", Reason);
        }
       /*************************************************************/
       }                                 // end if (CompCode != MQCC_FAILED)
     }                                   // end while (CompCode != MQCC_FAILED)
  /*****************************************************************/
  /*   Close input queue                                           */
  /*****************************************************************/
     MQCLOSE (Hcon,                          // Connection handle
              &Hinput,                       // Object handle for queue
              MQCO_NONE,                     // No close options
              &CompCode, &Reason);           // Return codes
     if (Reason != MQRC_NONE)
     {
        printf("MQCLOSE ended with reason code %ld\n", Reason);
     }
 /*****************************************************************/
 /*   Disconnect from queue manager                               */
 /*****************************************************************/
    MQDISC (&Hcon,                           // Connection handle
            &CompCode, &Reason);             // Return codes
    if (Reason != MQRC_NONE)
    {
       printf("MQDISC ended with reason code %ld\n", Reason);
    }
 /*****************************************************************/
 /*   Program ended successfully                                  */
 /*****************************************************************/
    printf("Program BL1 ended OK\n");
    return(0);
 }                                           // end MAIN
 /*****************************************************************************/
```

```
/*********************************************************************/
/*                    S U B R O U T I N E S                          */
/*********************************************************************/
/*          Check if file exists. If not, create it.                 */
/*********************************************************************/
int  CHECK_FILE ()
 {
   for (ij = 0; ij < 3; ij++) {              // Check for up to three files

       if (ij == 0) { strcpy (fn,fnfruit);
                      strncpy (file_id,"0001",4);
                    }
       if (ij == 1) { strcpy (fn,fnveggy);
                      strncpy (file_id,"0002",4);
                    }
       if (ij == 2) { strcpy (fn,fnother);
                      strncpy (file_id,"0003",4);
                    }
       strncpy (file_items,"10",2);              // Fixed 10 entries
       if ( (fp = fopen(fn,"r")) == NULL) {      // If file does not exist
             if ( (fp = fopen(fn,"w")) == NULL) {
                   printf("Cannot create file %s. Abort.\n", fn);
                   return (1);
             }
             il = fwrite (file_id, 4, 1, fp);    // File ID
             il = fwrite (file_items, 2, 1, fp); // Number or items

             for (ik = 0; ik < ITEMS ; ik++) {   // Get product from table
                 if (ij == 0) { strncpy (file_product,Fruit[ik].item,15);
                                strncpy (file_price,Fruit[ik].price,5);
                              }
                 if (ij == 1) { strncpy (file_product,Veggy[ik].item,15);
                                strncpy (file_price,Veggy[ik].price,5);
                              }
                 if (ij == 2) { strncpy (file_product,Other[ik].item,15);
                                strncpy (file_price,Other[ik].price,5);
                              }
                 il = fwrite (file_product,15,1,fp);  // Product name
                 il = fwrite (quantity,5,1,fp);       // Default quantity
                 il = fwrite (file_price,5,1,fp);     // Price
             }
             printf("File %s created\n", fn);
       }
       fclose (fp);
   }
   return (0);
 }
```

```
/*********************************************************************/
/*                  Read one of three product files                  */
/*********************************************************************/
int  READ_FILE (flda)
     int flda;
 {
     if (flda == 1)  strcpy (fn,fnfruit);
     if (flda == 2)  strcpy (fn,fnveggy);
     if (flda == 3)  strcpy (fn,fnother);
     if ( ( fp = fopen(fn,"r+")) == NULL ) {       // Open for read/write
        printf ("Problem opening file %s\n",fn);
        return (1);
     }
     ik = fread (msg1.ID, 4,1,fp);                 // File ID
     ik = fread (msg1.count, 2,1,fp);              // Number of items
     if (msg1.count[0] != ' ') {
        strncpy (wkfld,msg1.count,2);
        wkfld[2] = '\0';
     }
     else {
        wkfld[0] = msg1.count[1];
        wkfld[1] = '\0';
     }
     records = atoi (wkfld);
     ik = fread (msg1.message, 25, records, fp);   // read all products
     return (0);
 }
/*********************************************************************/
/*                  Open output queue                                */
/*********************************************************************/
int  OPEN_QUEUE (flda)
     int flda;
 {
   if (flda == 1) strcpy(odout.ObjectName,outputq1);
   else           strcpy(odout.ObjectName,outputq2);
   MQOPEN(Hcon,                        // Connection handle
          &odout,                      // Object descriptor for queue
          MQOO_OUTPUT,                 // Open options
          &Houtput,                    // Handle to queue
          &CompCode, &Reason);         // Return codes
   if (Reason != MQRC_NONE)
       printf("MQOPEN ended with reason code %ld\n", Reason);
   if (CompCode == MQCC_FAILED) {
     printf("Error: unable to open queue %s for output\n",odout.ObjectName);
     return(1);
   }
   return(0);
```

```
 }
/*********************************************************************/
/*                     Update inventory in files                     */
/*********************************************************************/
void UPDATE_FILE ()
{
 int pass, i1, i2, i3, done, qf, qo;
 int foffset, filechange;

 done = 0;                                      // Count items processed
 /*******************************************************************/
 /* For each pass read a file from disk and compare each item in the file  */
 /* with all items in the order message.                            */
 /*******************************************************************/
 for (pass = 1; pass < 4; pass++) {
     READ_FILE(pass);                           // Read one of 3 files
     filechange=0;                              // Indicator for update
     foffset = 0;                               // Offset for msg1.message
     /***************************************************************/
     /* Obtain a product from the inventory file                    */
     /***************************************************************/
     for (i1 = 0; i1 < records; i1++) {         // Check each record in file
         strncpy (file_product, msg1.message+foffset,15);
         file_product[15] = '\0';
         strncpy (file_quantity, msg1.message+foffset+15, 5);
         file_quantity[5] = '\0';
         printf("Fil=%s&&& items=%s\n",file_product, file_quantity);
         /***********************************************************/
         /* Obtain a product from the order message                 */
         /***********************************************************/
         offset = 6;                            // First item in msg buffer
         for (i2 = 1; i2 <= count; i2++) {      // Go through order message
             strncpy (order_name, buffer+offset,15);
             order_name[15] = '\0';
             strncpy (order_quantity, buffer+offset+15,5);
             order_quantity[5] = '\0';
             /*******************************************************/
             /* Compare name from inventory file with name from order msg  */
             /*******************************************************/
             if (strncmp (file_product, order_name, 6) == 0) {
```

```
                /*************************************************************/
                /* Match: Update quantity in inventory buffer                */
                /*************************************************************/
                qf = atoi(file_quantity);
                qo = atoi(order_quantity);
                if (qf >= qo) {
                   qf = qf - qo;
//                 itoa (qf, wkfld, 10);   does not work on AIX
                   sprintf(wkfld,"%d",qf);
                   strcpy (file_quantity, "00000");
                   length = strlen(wkfld);
                   for (i3 = 0; i3 < length; i3++)
                        file_quantity[4-i3] = wkfld[length-1-i3];
                   strncpy (msg1.message+foffset+15, file_quantity, 5);
                   filechange=1;
                }
                else nogo++;
                done++;
                strcpy (buffer+offset,"*****");
                break;
             }                                    // end product matched
                /*************************************************************/
                /* No match: Check next item in order message                */
                /*************************************************************/
             else offset = offset + 25;           // Next item in msg buffer
          }                                       // end pass through msg
          foffset = foffset + 25;                 // Next item in file
       }                                          // end search product file
       /*************************************************************************/
       /* Finished processing one of the inventory files. If an item has been*/
       /* updated save file on disk.  Return when all items in order message */
       /* have been processed.                                                  */
       /*************************************************************************/
       if (filechange == 1) {
          printf("Update file %s\n",fn);
          rewind(fp);
          il = fwrite (msg1.ID, 4, 1, fp);         // File ID
          il = fwrite (msg1.count, 2, 1, fp);      // Number or items
          il = fwrite (msg1.message, 25, records, fp);
       }
       fclose(fp);
       if (done == count) return;
    }                                             // end of a pass (file)
}
/*****************************************************************************/
```

B.2 Make File

```
#* Make file for a Business Logic

CC       = icc
LINK     = icc

CFLAGS   = /Ge /Gd- /Se /Re /ss /Gm+ /Ti /Q
LFLAGS   = /De /NOE /ALIGN:16 /EXEPACK /M /DEBUG /BASE:0x10000

OBJS     = bl1.obj

LIBS     = bmqapic2.lib os2386.lib mqm.lib

all: bl1.exe

bl1.exe: bl1.obj
  $(LINK) /B"$(LFLAGS)" $(CFLAGS) $(OBJS) $(LIBS) bl1.def

bl1.obj: bl1.c
  $(CC) /c $(CFLAGS) $*.c
```

B.3 Definition File

```
NAME    bl1     WINDOWCOMPAT
PROTMODE
```

Appendix C. Diskette Contents

The diskette contains the examples developed in this book. Table 6 lists the directories and the file names.

Note: The diskette can store only file names in 8.3 format. Make sure that you change the the file name when you copy the files onto your own system.

Table 6 (Page 1 of 2). Files on Diskette

Real file name	File name on diskette	Description
\demo	**Front page for demonstration program**	
mqwelcom.html	same	HTML page that links to the demonstration program.
\java	**Client progams for Chapter 9, "The Sample Application" on page 105**	
FBgetQ2.class	FBgetQ2.cla	Class that gets messages containing order lists from queue 2.
FBgetQ2.java	FBgetQ2.jav	
FBputQ1.class	FBputQ1.cla	Class that puts messages for the server program on queue 1.
FBputQ1.java	FBputQ1.jav	
GTCget.class	GTCget.cla	Class that gets status messages from queue 3.
GTCget.java	GTCget.jav	
OrderlistApplet.html	Order.htm	HTML file that loads the applet.
OrderListView.class	Order.cla	The applet running in the client (order entry GUI)
OrderListView.jav	Order.jav	
\C	**Server progam for Chapter 9, "The Sample Application" on page 105.**	
bl1.exe	same	The server program the applet communicates with.
bl1.c	same	
bl1.def	same	Definition file for compile.
bl1.mak	same	Make file for compile.
\mqm	**MQSeries definitions for Chapter 9, "The Sample Application" on page 105**	
COMMANDS.IN	same	Some helpful RUNMQSC commands.
JAVACOMA.TST	same	Queue manager objects.
run.cmd	same	Command file to start the server application.

Table 6 (Page 2 of 2). Files on Diskette

Real file name	File name on diskette	Description
\myjava	**Java examples from Chapter 5, "Java Overview" on page 41.**	
Hello.class	Hello.cla	The ″Hello World″ application.
Hello.java	Hello.jav	
HelloWorld.class	HelloW.cla	The ″Hello World″ applet.
HelloWorld.java	HelloW.jav	
HelloWorld.html	HelloW.htm	The HTML file that loads the ″Hello World″ applet.
\html	**HTML examples from Chapter 4, "HTML Overview" on page 27**	
MQJCBAN.GIF	same	The MQSeries banner displayed in the Web page.
myhtml.html	myhtml.htm	The HTML file described in the chapter.
OTHER.HTML	OTHER.htm	The HTML file used to demonstrate links to another Web page.

Appendix D. Special Notices

This publication is intended to help network adminstrators to install and maintain Internet services, and application programmers to write programs with Java and MQSeries. The information in this publication is not intended as the specification of any programming interfaces that are provided by Java and MQSeries. See the PUBLICATIONS section of the IBM Programming Announcement for MQSeries and Java for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (″vendor″) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each

item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

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.

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

AIX
Application System/400
AS/400
BookManager
BookMaster
IBM
MQSeries
Open Blueprint
Operating System/2
OS/2
RISC System/6000
RS/6000
SupportPac

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

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

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

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

Java and HotJava are trademarks of Sun Microsystems, Inc.

SunOS, SPARCstation, Network File System, NFS
Netscape, Netscape Navigator
Netscape Communications Corporation:

Other trademarks are trademarks of their respective companies.

Appendix E. Related Publications

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

E.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 155.

- *Examples of Using MQSeries in WWW*, SG24-4882

E.2 Redbooks on CD-ROMs

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

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

E.3 Other Publications

These publications are also relevant as further information sources:

- *MQSeries Planning Guide*, GC33-1349
- *MQSeries Clients*, GC33-1632
- *MQSeries Command Reference*, SC33-1369
- *MQSeries Distributed Queuing Guide*, SC33-1139
- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Application Programming Guide*, SC33-0807

How to Get ITSO Redbooks

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

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

How IBM Employees Can Get ITSO Redbooks

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

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

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

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

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

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks:

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

  To register for information on workshops, residencies, and redbooks:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  ```

  For a list of product area specialists in the ITSO:

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

- **Redbooks Home Page on the World Wide Web**

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

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

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

  IBM employees may obtain LIST3820s of redbooks from this page.

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

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

How Customers Can Get ITSO Redbooks

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

- **Online Orders** (Do not send credit card information over the Internet) — send orders to:

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

- **Telephone orders**

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

- **Mail Orders** — send orders to:

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

- **Fax** — send orders to:

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

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

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

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

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

- **Internet Listserver**

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

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

- Invoice to customer number
- Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

DO NOT SEND CREDIT CARD INFORMATION OVER THE INTERNET.

Glossary

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

applet. A Java program that runs within the Web browser, providing executable content, but having no access to the PC environment. Applets can be embedded within existing HTML documents and allows the user to interact with them. Applet can also run without a browser, you than have to use an applet viewer.

application. A Java program that runs outside a Web browser, having access to the entire PC environment. Applications commonly run from within a shell environment, or from the command line.

asynchronous messaging. A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message.

browser. A browser is a synonym of Web browser or Client browser. It is a program with a graphical interface. Commonly used browsers are:

- IBM WebExplorer
- MS Internet Explorer
- Netscape Navigator
- Sun HotJava

class. A class is an encapsulated collection of data, and methods to operat e on the data. A class may be instantiated to produce an object that is an instance of the class.

client. In MQSeries, a client is a run-time component that provides access t o queuing services on a server for local user applications.

Email. Electronic mail (also e-mail) is a text message delivering method across the Internet or the Intranet. Plain text (ASCII format) and binary files can be send.

encapsulation. Encapsulation is an object-oriented programming technique that makes an object′s data private or protected and allows programmers to access and manipulate the data only through method calls.

Firewall. A security scheme to protect one or more computer within the Internet from intrusion by other external computer. A firewall is a invisible boundary created by software. Computer within a firewall can share resources and have access to other computer, which computer outside the firewall can′t. External requests can be examined and filtered before they are allowed to go through the firewall and than have also access to the resources within the firewall.

host. A term referring to a computer (′of every size′) that is connected to the Internet or the Intranet.

HotJava. A Web browser developed by Sun Microsystems, the first ′Java-enabled′ Web browser.

HTML. HTML (Hypertext Markup Language) is a language used to define information that is to be displayed on the World Wide Web.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

Internet. The largest computer network of the world. It is a network of networks (a network can be a single computer) linked together to form a collective entity. They all communicate via the TCP/IP protocol. Other names are: Superhighway, Datahighway, The Net or CyberSpace.

interface. An interface is a special type of class which contains only abstract methods and no instance variables. An interfaces provides a

common set of methods that can be implemented by subclasses of a number of different classes.

Intranet. It follows the same rules as the Internet, but is not an open network. It's only operates within an enterprise network protected by a firewall against the Internet.

Java. An object-oriented, secure and portable programming language, comparable with C++ without the complex parts of C++ like pointers and memory management. Java is an interpreter language thats needs a virtual java engine to run. Many Web browser have a built-in Java engine.

Java Developers Kit (JDK). A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java developmetn tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

message. In message queuing applications, a communication sent between programs. There are persistent and non-persistent messages.

message channel. In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender and a receiver) and a communication link.

message channel agent. A program that transmits messages from a transmission queue to a communication link, or from a communication link to a destination queue.

message channel interface. The MQSeries interface to which application programs that transmit messages between an MQSeries queue manager and another messaging system must conform. A part of the MQSeries Framework.

message queue. Synonym for queue.

message queue interface. The programming interface provided by the MQSeries queue managers. It allows application programs to access message queuing services.

message queuing. A programming technique in which each program within an application communictes with the other programs by putting messages on queues.

messaging. See *synchronous messaging* and *asynchronous messaging*.

method. Method is the object-oriented programming term for a function or procedure.

MQI. Message queue interface.

MQI channel. Connects an MQSeries client to a queue manager on a server system. It transfers only MQI calls and responses in a bidirectional manner.

MQSC. MQSeries commands.

MQSeries. A family of IBM licensed programs that provide message queuing services.

MQSeries client. Part of an MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries commands (MQSC). Commands that are used to manipulate MQSeries objects.

object. (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group.
(2) in MQSeries, an object is a queue manager, a queue, or a channel.

package. A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to all the classes in the package and to all non-private A private field is not visible outside its own class .

protected. A protected field is only visible within its own class, within subclasses, or within packages of which the class is a part.

public. A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible.

queue. A queue is an MQSeries object. Message queuing applications can put messages on, and get messages from, a queue.

queue manager. A queue manager is a system program the provides message queuing services to applications.

server. (1)In MQSeries a server is a queue manager that provides message queuing services to client applications running on a remote workstation.
(2) More generally a server is a program that responds to requests for information in the particular two-program information flow model of client/server.
(3)The computer on which a server program runs.

socket. A mechanism to allow processes to communicate with one other over the Internet. They are the core functionality behind the TCP/IP protocol.

subclass. A subclass is a class that extends another. The subclass inherits the protected methods and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass′s protected methods and variables are available to the subclass.

Synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing.

Web. A short name for the World-Wide Web (WWW). You can think about the web as a spider web, which connects thousands of computer together.

Web browser. A program that formats and displays information that is distributed on the World Wide Web.

Web server. A program that serves HTML pages to the Web browser. See also Web browser.

World Wide Web (Web). The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

List of Abbreviations

API	Application Programming Interface
CGI	Common Gateway Interface
DOS	Disk Operating System
DNS	Domain Name Server
FTP	File Transfer Protocol
GIF	Graphic Interchange Format
HPFS	High Performance File System
HTML	HyperText Mark-up Language
HTTP	HyperText Transfer Protocol
ICCS	IBM Internet Connection Secure Server
ITSO	International Technical Support Organization
IVP	Installation Verification Program
JDK	Java Development Kit
MCA	message channel agent
MQ	Message Queuing
MQI	Message Queuing Interface
MQM	MQ queue manager
NNTP	Network News Transfer Protocol
PID	program identification number
PROFS	Professional Office System
SDK	Software Development Kit
SGML	Standard Generalized Mark-up Language
SMTP	Simple Mail Transfer Protocol
SLIP	Serial Line Internet Protocol
SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol / Internet Protocol
URL	Uniform Resource Locator
WWW	World Wide Web

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

ITSO Redbook Evaluation

Internet Application Development with MQSeries and Java
SG24-4896-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redeval@vnet.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_____

_____

_____

_____

_____

Printed in U.S.A.