WebSphere J2EE Application Development for the IBM iSeries Server

Build and deploy J2EE compliant applications for WebSphere 4.0

Use Application Developer to build iSeries servlets, JSPs, and EJBs

Learn how to interface to legacy applications

Bob Maatta
Luis Aused
Cliff Liang
Biswaanath Panigrahi
Lowell Thomason
Fernando Zuliani
Take Note! Before using this information and the product it supports, be sure to read the general information in “Notices” on page ix.
# Contents

Notices ................................................................. ix
Trademarks ............................................................ x

Preface ................................................................. xi
The team that wrote this redbook. ................................ xi
Special notice ......................................................... xiii
Comments welcome .................................................. xiv

Chapter 1. Introduction to J2EE. ........................................... 1
  1.1 Java 2 Enterprise Edition (J2EE) .................................. 2
     1.1.1 J2EE platform technologies .................................. 4
     1.1.2 J2EE 1.2 required standard extension APIs .................. 5
     1.1.3 J2EE package levels in WebSphere .......................... 5
  1.2 J2EE containers .................................................. 6
  1.3 J2EE components ................................................ 7
     1.3.1 Client-side components ....................................... 7
     1.3.2 Server-side components: Servlets ............................ 8
     1.3.3 JavaServer Pages: Separating presentation logic .......... 8
     1.3.4 Server-side components: EJBs ............................... 8
  1.4 J2EE services ................................................... 9
     1.4.1 Java Naming Directory Interface ............................. 9
     1.4.2 Java Database Connectivity ................................. 10
     1.4.3 Security ..................................................... 10
     1.4.4 Transactions (JTA and JTS) ................................. 11
     1.4.5 JavaBean Activation Framework (JAF) ....................... 12
  1.5 J2EE communication ............................................. 13
     1.5.1 Remote method invocation (RMI/IIOP) ....................... 13
     1.5.2 Java Messaging Service ..................................... 13
     1.5.3 JavaMail ................................................... 14
  1.6 J2EE packaging and deployment ................................ 14
     1.6.1 J2EE deployment descriptor ................................. 15
  1.7 J2EE platform roles ............................................ 17
  1.8 J2EE additional resources ..................................... 19

Chapter 2. Servlet and JSP development using VisualAge for Java .......................... 21
  2.1 Servlet support in WebSphere Advanced Edition 4.0 ............ 22
     2.1.1 IBM development environments for WebSphere applications 22
  2.2 Introduction to servlets ........................................ 23
     2.2.1 Simple servlet example ..................................... 25
  2.3 Setting up VisualAge for Java to develop and test servlets .... 26
     2.3.1 Loading the required features ............................... 26
     2.3.2 Using the WebSphere Test Environment ...................... 27
     2.3.3 Testing the servlet under VisualAge for Java Enterprise Edition 4.0 28
     2.3.4 Exporting class files to a JAR file ......................... 28
  2.4 Using JDBC to access an iSeries database ........................ 29
     2.4.1 The architecture of the sample application ................. 30
     2.4.2 JDBCItemCatalogSupport class ............................. 31
     2.4.3 JDBCCatalogSupport class .................................. 32
     2.4.4 Testing the application in the scrapbook ................... 34
Chapter 4. Introduction to WebSphere Studio Application Developer

4.1 WebSphere Studio Application Developer overview
4.2 Getting started with Application Developer
  4.2.1 Navigating in Application Developer
  4.2.2 Importing resources
  4.2.3 Customizing Application Developer
4.3 Working with Java code
  4.3.1 Adding new methods
  4.3.2 Compiling Java code
  4.3.3 Running the Java code
  4.3.4 Exporting the Java code
4.4 Conclusion
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include 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.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.
Trademarks

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

- e (logo)®
- AFP™
- AS/400®
- AS/400e™
- Balance®
- Redbooks (logo)™
- MQSeries®
- SP™
- IBM®
- OS/400®
- iSeries™
- Perform™
- FFST™
- Redbooks™
- WebSphere®
- S/390®
- zSeries™

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

- Lotus
- Word Pro
- Domino™

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

WebSphere Application Server 4.0 delivers the Java 2 Enterprise Edition (J2EE) implementation. It is the IBM strategic Web application server and a key IBM @server iSeries product for enabling e-business applications. The iSeries server and WebSphere Application Server are a perfect match for hosting e-business applications.

You can build J2EE applications using WebSphere Studio Application Developer – a new IBM application development environment. This is a follow-on product for VisualAge for Java and WebSphere Studio. It combines the best of these products into one integrated development environment.

This IBM Redbook shows customers, business partners, and ISVs how to build and deploy iSeries J2EE applications and how to use them to access iSeries resources. It also shows you how to use your iSeries server as a Java server. It is written for anyone who wants to use Java servlets, JavaServer Pages, and Enterprise JavaBeans on the iSeries server.

This redbook provides many practical programming examples with detailed explanations of how they work. The examples were developed using VisualAge for Java Enterprise Edition 4.0 and WebSphere Studio Application Developer 4.02. They were tested using WebSphere Application Server 4.0.2 Advanced Edition and Advanced Edition Single Server. To effectively use this book, you should be familiar with the Java programming language and object-oriented application development.

Throughout this redbook, we show and discuss code snippets from example programs. The example code is available for download. To understand this code better, download the files, as explained in Appendix A, and use them as a reference.

You can learn how to install, configure, and administer WebSphere 4.0 in an iSeries environment by referring to the complementary redbook WebSphere 4.0 Installation and Configuration on the IBM @server iSeries Server, SG24-6815.

Note: This Redbook reflects the IBM @server iSeries server name. Throughout this Redbook, we use the shortened version “iSeries” to refer to both AS/400e and iSeries servers.

The team that wrote this redbook

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

Bob Maatta is a Consulting Software Engineer at the IBM International Technical Support Organization, Rochester Center. He is the ITSO technical leader for iSeries e-business application development. He writes extensively and develops and teaches IBM classes worldwide on all areas of iSeries client/server and e-business application development. He has worked on numerous computer platforms including S/390, S/38, AS/400, and personal computers. In his
assignment, he specializes in Java programming and the IBM WebSphere Application Server. He is a Sun Certified Java Programmer and a Sun Certified Java Developer. He is a graduate of Michigan Technology University in Houghton, Michigan.

**Luis Aused** is an IT Specialist for IBM Global Services Spain. He has worked for IBM for over six years. He has developed several applications for the iSeries server, including a data warehouse application for an Insurance company. In the last two years, he developed several e-business applications for WebSphere on the IBM zSeries platform that interface with DB2 and MQSeries. His areas of expertise include e-business application development, the iSeries server, WebSphere, VisualAge for Java, and DB2. He holds a degree in physics from Complutense University, Madrid, Spain.

**Cliff Liang** is a Senior Consultant at ASTECH Solutions, Inc., a Toronto based consulting firm and IBM Business Partner. Before joining ASTECH Solutions, Inc. in 2001, he worked at IBM China as a Senior IT Specialist for the iSeries brand. His expertise includes the iSeries server, database performance, Java, and the IBM Framework for e-business. He is a graduate of the University of Science & Technology in China with a bachelor's degree in electrical engineering.

**Biswanath Panigrahi** is an Advisory IT Specialist working in PartnerWorld for Developers at Bangalore, India. He works with the iSeries brand and is responsible for supporting Techline for ASEAN/SA. He has worked at IBM for over four years and has over five years of professional IT experience. He currently specializes in creating e-business solutions and Web enabling existing application using various development tools. His area of expertise include the iSeries server, WebSphere Application Server, Domino, Java, client/server programming, and Web-based development. He holds a master's degree in computer application from Orissa University of Agriculture and Technology, Bhubaneswar, India.

**Lowell Thomason** is a Staff Software Engineer at the IBM iSeries Software Support Center in Rochester, Minnesota. He is responsible for supporting AS/400 and iSeries e-business applications for both U.S. and international customers. He has four years of experience in technical support, problem determination, customer relations, application development, custom coding, and e-commerce solutions. He holds a bachelor of science degree in computers from Mayville State University, Mayville, North Dakota. His areas of expertise include WebSphere Application Server, Java, Net.Data, HTML, and SQL.
Fernando Zuliani is an IBM Certified Consulting IT Specialist focused on the technical sales and support of the WebSphere Suite of Products. His job is to support the Americas, with specific focus on Latin American sales. He has 14 years of experience with IBM. In his current job, he performs WebSphere Proof-of-Concept (POC) scenarios for customers in IBM Latin America. These POC scenarios consist of in-depth workshops with presentations and hands-on exercises that map to customer needs. He has presented extensively at IBM events worldwide.

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

Jim Beck
Mike Burke
Larry Hall
Dan Hiebert
Kevin Paterson
George Silber
Art Smet
Frances Stewart
Mike Turk
Lisa Wellman
IBM Rochester Laboratory

Special notice

This publication is intended to help anyone who wants to use IBM WebSphere Application Server Version 4.0 Advanced Edition or WebSphere Application Server Version 4.0 Advanced Single Server Edition in the iSeries environment. The information in this publication is not intended as the specification of any programming interfaces that are provided by the IBM WebSphere Application Server Advanced Edition product. See the PUBLICATIONS section of the IBM Programming Announcement for the IBM WebSphere Application Server Standard Edition product for more information about what publications are considered to be product documentation.
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:

- Use the online Contact us review Redbook form found at:
  ibm.com/redbooks
- Send your comments in an Internet note to:
  redbook@us.ibm.com
- Mail your comments to
  IBM Corporation, International Technical Support Organization
  Dept. JLU  Building 107-2
  3605 Highway 52N
  Rochester, Minnesota 55901-7829
Introduction to J2EE

Java 2 Platform, Enterprise Edition (J2EE) defines a standard that applies to all aspects of architecting and developing multi-tier server-based applications. It defines a standard architecture composed of an application model, a platform for hosting applications, a compatibility test suite (CTS), and a reference implementation.

The primary concern of J2EE is the platform specification. It describes the runtime environment for a J2EE application. This environment includes application components, containers, resource manager drivers, and databases. The elements of this environment communicate with a set of standard services that are also specified.

J2EE makes all Java enterprise APIs and functionality available and accessible in an integrated manner. This integration helps to simplify complex problems in the development, deployment, and management of multi-tier server-centric enterprise solutions. WebSphere Application Server Version 4.0 is fully J2EE 1.2 compliant.
1.1 Java 2 Enterprise Edition (J2EE)

The Java 2 platform is a Sun specification for enterprise application development. Sun licenses the technology of J2EE to other companies, also known as the Web Technology Providers, such as IBM. Figure 1-1 shows the three editions that make up the Java 2 platform.

![Java 2 platform editions](image)

Each of the platforms are explained here:

- **Java 2 Platform, Micro Edition (J2ME)**
  
  A highly optimized Java runtime environment targeting a wide range of consumer products, including pagers, cellular phones, screen phones, digital set-top boxes, and car navigation systems.

- **Java 2 Platform, Standard Edition (J2SE)**
  
  The essential Java 2 SDK, tools, runtimes, and APIs for developers writing, deploying, and running applets and applications in the Java programming language. It includes earlier Java Development Kit versions JDK 1.1 and JRE 1.1.

- **Java 2 Platform, Enterprise Edition (J2EE)**
  
  Combines a number of technologies in one architecture with a comprehensive Application Programming Model and compatibility test suite for building enterprise-class server-side applications.

  Java 2 Enterprise Edition is a set of related specifications, a single standard for implementing and deploying enterprise applications. J2EE is an extension of the Java 2 Platform, Standard Edition. J2EE makes all Java enterprise APIs and functionality available and accessible in an integrated manner. Companies can use J2EE as a reference to identify product capabilities before they invest in a technology provider vendor. As shown in Figure 1-2, the standard includes:

  - An Application Programming Model for enterprise application developers and architects.
  - A platform to run the applications.
  - A compatibility test suite to verify that a platform complies with the standard.
– A reference implementation to show the capabilities of J2EE and provide an operational definition.

– Containers that are run-time environments that provide components with specific services. For example, Web containers provide run-time support to clients by processing requests by invoking JavaServer Page (JSP) files and servlets and returning results from the components to the client. Similarly, Enterprise JavaBean (EJB) containers provide automated support for transaction and state management of enterprise bean components, as well as lookup and security services.

Figure 1-2 J2EE architecture

The J2EE specification is a consensus and collaboration from numerous major enterprise software vendors. The current J2EE specification level is 1.3. Since this redbook covers WebSphere Application Server 4.0, which supports J2EE 1.2, we focus on the WebSphere Application Server 4.0 supported J2EE 1.2 level. WebSphere 4.0 supports some of the J2EE 1.3 specification.

Benefits of J2EE
J2EE defines a simple standard that applies to all aspects of architecting and developing large scale applications. J2EE naturally encourages the adoption of a multiple tier architecture and a strict separation between business logic and the presentation layer.

In addition, the architectural approach suggested by the J2EE standards lends itself to the implementation of highly scalable infrastructures. An example is WebSphere Application Server 4.0, where the application workload can be transparently distributed across a number of concurrent processes or even across a number of parallel systems. J2EE also provides the technology to facilitate the integration of existing applications with newly developed J2EE applications.

The specifications are generic enough to allow an end user to choose among a wide array of middleware, tools, hardware, operating systems, and application designs. J2EE has created a whole new market for application servers, development tools, connectors, and components.

Multi-tier application environments
The J2EE specification supports a distributed application model. Applications can be spread across multiple platforms or tiers. Typically, we divide J2EE applications into three tiers. The J2EE application parts as shown in Figure 1-3 are presented in the J2EE components.
The first tier contains the presentation logic. Typically this is done by a servlet or JavaServer Page. The middle tier contains the business logic. Typically this is controlled by servlets or Enterprise JavaBeans.

The third tier contains the database and legacy applications.

Clients typically interface to the application through a Web browser, a Java application, or a pervasive computing device.

1.1.1 J2EE platform technologies

From a pure technological standpoint, you can recognize three areas (shown in Figure 1-4) within the J2EE specifications.

**Components**

Components are provided by the application developers and include servlets, JavaServer Pages, and Enterprise JavaBeans. These components live inside J2EE containers that are provided by the middleware vendors, such as IBM with the WebSphere Application Server. A J2EE container needs to provide support for a number of services and communications. See 1.3, “J2EE components” on page 7, for details.

**Services**

The services are functions that are accessible to the components via a standard set of APIs. For example, a component has to access a relational database by using the Java Database Connectivity (JDBC) APIs, while it can use the Java Naming Directory Interface (JNDI) APIs to access the
naming services. These APIs need to be supported by the container. Each level of the J2EE specification calls for supporting a specific level of each set of APIs. For example, the current level of J2EE supported by WebSphere 4.0 is 1.2. This implies support for JDBC level 2.0. See 1.4, “J2EE services” on page 9.

Communication
The essence of J2EE is the definition of a distributed, object-oriented infrastructure. Components need to communicate with each other in this distributed world. Therefore, the containers need to provide the appropriate communication mechanisms to make this happen. Examples of communications included in the J2EE standards are RMI/IIOP for remote method calls, JavaMail for programmatic access to e-mail, and Java Messaging Service (JMS) for accessing messaging technologies. Refer to 1.5, “J2EE communication” on page 13, for more details.

1.1.2 J2EE 1.2 required standard extension APIs

J2EE application components run in runtime environments provided by the containers that are part of the J2EE platform. The J2EE platform supports four separate types of containers (Figure 1-5), one for each J2EE application component type. The containers include application client containers, applet containers, Web containers for servlets and JSPs, and Enterprise JavaBean containers.

The J2EE platform also includes a number of Java standard extensions. Table 1-1 indicates which standard extensions are required to be available in each type of container. It also shows the required version of the standard extension.

<table>
<thead>
<tr>
<th>API</th>
<th>Applet</th>
<th>Application client</th>
<th>Web</th>
<th>EJB</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC 2.0</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>JTA 1.0</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>JNDI 1.2</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Servlet 2.2</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>JSP 1.1</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>EJB 1.1</td>
<td>N</td>
<td>Y (1)</td>
<td>Y (2)</td>
<td>Y</td>
</tr>
<tr>
<td>RMI/IIOP 1.0</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>JMS 1.0</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>JavaMail 1.1</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>JAF 1.0</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

(1) - Application clients can only use the enterprise bean client APIs.
(2) - Servlets and JSP pages can only use the enterprise bean client APIs.

1.1.3 J2EE package levels in WebSphere

Table 1-2 shows the details of servlet, JSP, and EJB changes included in the J2EE component level of WebSphere.
Table 1-2  J2EE component API levels in WebSphere

<table>
<thead>
<tr>
<th>API</th>
<th>WAS 3.5.2+</th>
<th>WAS 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servlet</td>
<td>2.1, 2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>JSP</td>
<td>0.91, 1.0, 1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>EJB</td>
<td>1.0</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 1-3 show the details of JDBC, JTA/JTS, JNDI, JAF, XML4J, and XSL changes included in J2EE Services level of WebSphere.

Table 1-3  J2EE service API levels in WebSphere

<table>
<thead>
<tr>
<th>API</th>
<th>WAS 3.5.2+</th>
<th>WAS 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>1.0, 2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>JTA/JTS</td>
<td>1.0, 1.0.1, 1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>JNDI</td>
<td>1.2</td>
<td>1.2.1</td>
</tr>
<tr>
<td>JAF</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>XML4J</td>
<td>2.0.15</td>
<td>3.1.1</td>
</tr>
<tr>
<td>XSL</td>
<td>1.0.1</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 1-4 shows the details of RMI/IIOP, JMS, and Java Mail changes included in the J2EE communication level of WebSphere.

Table 1-4  J2EE communication API levels in WebSphere

<table>
<thead>
<tr>
<th>API</th>
<th>WAS 3.5.2+</th>
<th>WAS 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI/IIOP</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>JMS</td>
<td>1.0.1</td>
<td>1.0.1</td>
</tr>
<tr>
<td>Java Mail</td>
<td>N/A</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Note: WebSphere Application Server 4.0 also includes some J2EE 1.3 features such and J2C and JMS Transaction (JMS/XA).

WebSphere 4.0 JDK level
WebSphere 4.0 implements JDK level 1.3, while WebSphere 3.5 used JDK 1.2.2. One of its major benefits is improved performance. The IBM Java Runtime Environment (JRE) 1.3 provides up to a 22 percent improvement over the JRE 1.2.2. Improvements in the JVM, JIT compiler, garbage collection, and threads management account for the performance improvements.

1.2 J2EE containers

Containers provide the runtime support for the application components. A container provides a total view of the underlying J2EE APIs to the application components. Interposing a container between the application components and the J2EE services allows the container to inject services defined by the components' deployment descriptors, such as declarative transaction management, security checks, resource pooling, and state management.
Each component runs inside a container that is provided by the J2EE platform provider. A typical J2EE product provides a container for each application component type: application client container, applet container, Web component container, and enterprise bean container as shown in Figure 1-5. The services that are offered depend on the type of container.

1.3 J2EE components

Components are provided by application developers and include applets, servlets, JavaServer Pages, and Enterprise JavaBeans. These components live inside J2EE containers. A J2EE container needs to provide support for a number of services and communications. Figure 1-5 shows the J2EE component object model.

1.3.1 Client-side components

Application clients and applets are components that run on the client.

Application clients
Client-side components are Java programs that are typically graphical user interface (GUI) programs that run on a desktop computer. Application clients offer a user similar experience to that of native applications and have access to all of the facilities of the J2EE middle tier.

Client-side components run inside a J2EE compliant client container. A client container provides security, communication and other required J2EE services. Application clients can interact with:

- EJBs through RMI/IIOP
- Web components through HTTP/HTTPS
The deployment of client side components is platform specific, since details are unspecified by the J2EE specifications.

**Applets**

Applets are GUI components that typically run in a Web browser. They can run in a variety of other applications or devices that support the applet programming model. Applets can be used to provide user interfaces for J2EE applications.

Applets were Java's first attempt to solve some of the problems created by the common gateway interface (CGI) programming model. They created an entirely new set of challenges including network bandwidth requirements, poor performance, security, and lack of function. They are not widely accepted. Servlets provide an alternative to applets.

### 1.3.2 Server-side components: Servlets

Servlets are Java classes that allow application logic to be embedded in the HTTP request-response process. The J2EE 1.2 specification requires support for the servlet 2.2 API, as described in Table 1-1 on page 5. For more details about servlets, see 2.2, "Introduction to servlets" on page 23.

### 1.3.3 JavaServer Pages: Separating presentation logic

JSPs enable server-side scripting using Java. Server-side scripting is not an alternative to client-side scripting. Client-side scripting, JavaScript or Java applets, is important for performing input validity tests and other computation eliminating interaction with the server.

Although it is certainly possible to create a servlet that "does everything" (controller, model, and view logic), this is highly undesirable from both an initial coding and a maintenance perspective. JSPs provide a clean way to separate the View parts (presentation logic) from the rest of the Web application:

- This means that Java programmers can focus on building highly valuable, reusable component frameworks in the object space, and don't have to deal with HTML.
- Web developers can focus on building a catchy, attractive Web site, and don't have to deal with the discipline of programming. Even better, they can own the JSPs, eliminating the problem of programmers and Web designers tromping on each others' work.
- The JSP specification allows an <HTML> oriented syntax, with clear <BEAN> tags to help the Web designer understand how to inject dynamic content from the Java components running in the object space.

JSPs are responsible for generating the output HTML going back to the Web client. They bridge the gap between Java and HTML, because the source JSP code is parsed and converted to pure a Java servlet, and then loaded and run just like any other servlet under the control of WebSphere. The only responsibility of this JSP/servlet is to generate output HTML, fulfilling the role of the View part of the application. For more information about JSPs, see 2.6, "JSP support in WebSphere Version 4.0" on page 52.

The J2EE 1.2 specification requires support for JSP 1.1, as described in Table 1-1 on page 5.

### 1.3.4 Server-side components: EJBs

The Enterprise JavaBean standard is a server-side Java-based component architecture for building multi-tier, distributed object applications. Components are pre-developed application code used to assemble applications.
EJBs have the same programming model as the client-side JavaBeans programming model, which makes it easy for Java programmers to use EJBs to build applications. By building reusable business objects as EJBs, programmers can focus on pure business logic and let the open standards based (EJB specification) EJB server manage the way the EJBs are mapped back to persistent data stores.

The J2EE 1.2 specification requires support for EJB 1.1, as described in Table 1-1 on page 5. For more information about Enterprise JavaBeans, see Chapter 6, “Introduction to Enterprise JavaBeans” on page 215.

1.4 J2EE services

The services are functions that are accessible to the components through a standard set of APIs. For example, a component has to access a relational database by using the JDBC APIs, while it can use the JNDI APIs to access the naming services. These APIs need to be supported by the container. Each level of the J2EE specification calls for supporting a specific level of each set of APIs. For example, the current level of J2EE supported by WebSphere 4.0 is 1.2, which requires support for JDBC level 2.0.

1.4.1 Java Naming Directory Interface

JNDI is an integral part of J2EE to lookup J2EE objects and resources. JNDI APIs provide naming and directory functionality to Java programs, allowing components to store and retrieve named Java objects. Naming services provide name-to-object mappings.

JNDI is independent of any specific directory access protocol, allowing easy deployment of new directory services and manipulation of Java instances by name.

Containers provide two levels of naming schemes:

- **Global**: The actual JNDI namespace.
- **Local**: Read-only, accessible to components. Local names are bound to their global counterparts at deployment.

The J2EE 1.2 specification requires support for the JNDI 1.2 specification, as described in Table 1-1 on page 5. WebSphere 4.0 provides an integrated JNDI 1.2 compliant name service.

**Using JNDI**

To access its naming context, a component creates a javax.naming.InitialContext object.

WebSphere 4.0 provides a new naming service named `com.ibm.websphere.naming.WsnInitialContextFactory`. WebSphere 3.5.2+ uses `com.ibm.ejs.ns.jndi.CNInitialContextFactory`. It is deprecated in WebSphere 4.0.

System-provided objects, such as UserTransaction objects, are stored in `java:comp/env` in the JNDI name space. User-defined objects are stored in subcontexts of `java:comp/env`. Here are a couple of examples:

- `java:comp/env.ejb`
- `java:comp/env/jdbc`
1.4.2 Java Database Connectivity

JDBC provides database-independent connectivity to a variety of data stores. The J2EE 1.2 specification requires:

- **JDBC 2.0 Core APIs**: Basic database services
- **JDBC 2.0 Extension APIs**: Advanced functionality
  - Connection Pooling
  - Transactional capabilities

The JDBC 2.0 Core and Extension APIs are supported in WebSphere 3.5 and above. Functionality is provided by a combination of WebSphere Application Server and a compliant JDBC driver.

1.4.3 Security

J2EE access control involves authentication to verify the user's identity:

- **Basic authentication**: The Web server authenticates a principal using the user name and password obtained from the Web client.

- **Digest authentication**: User name and a password are transmitted in an encrypted form.

- **Form-based authentication**: The Web container can provide an application-specific form for logging in.

- **Certificate authentication**: The client uses a public key certificate to establish its identity and maintain its own security context.

J2EE access control also involves authorization to determine if the user has permission to access the requested resource:

- Authorization is based on roles, which contain users or groups of users.

- Permissions are mapped by the deployer.

The security runtime consists of three core components, as shown in Figure 1-6.

A security plug-in is attached to a Web server. The plug-in performs initial security checks when users request Web resources, such as HTML files or servlets, from Web browsers over HTTP. The security plug-in challenges the Web client to authenticate, and contacts the EJS Web Collaborator to check the client's authentication information and check authorization.

An EJS Web collaborator is attached to every application server that contains a Web container. For every method call to a servlet, the EJS Web collaborator contacts the security application to perform the authorization check.

An EJS security collaborator is attached to every application server that contains an Enterprise JavaBean container. For every method call to an enterprise bean, the security collaborator contacts the security application to perform the authorization check and enforce the delegation policy.
1.4.4 Transactions (JTA and JTS)

The Java Transaction API (JTA) allows applications to access transactions in a manner that is independent of specific implementations. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system:

- The transactional application
- The J2EE server
- The manager that controls access to the shared resources affected by the transactions

The J2EE 1.2 specification requires support for JTA 1.0, as described in Table 1-1 on page 5.

Java Transaction Service (JTS) specifies the implementation of a transaction manager that supports JTA and implements the Java mapping of the Object Management Group Object Transaction Service 1.1 specification. A JTS transaction manager provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and propagation of information that is specific to a particular transaction instance.

JTA is the API, and JTS is the implementation. JTA is oriented at application developers, and JTS is oriented at middleware providers.
There are two ways to start a transaction:

- **Programmatic:** In the Java code, using the `javax.transaction.UserTransaction` interface.
- **Declarative:** Using EJB Container Managed Transaction.

### Container-managed transactions

The EJB container is responsible for managing transaction boundaries, dictated by the transaction attribute modes specified for the EJB method as defined in the EJB deployment descriptor. Two transactional specifiers, the transaction attribute and the isolation attribute, are set in the deployment descriptor. The values chosen for these specifiers give the container the information it needs to generate the support code.

The transaction attributes are:

- **Required:** Use client transaction context, if present; otherwise create a new one.
- **Required New:** Always create a new transaction.
- **Not supported:** Don’t propagate the client transaction, if present.
- **Supported:** Use the client transaction, if present.
- **Mandatory:** Use the client transaction, if present; otherwise, throw a `TransactionRequiredException` exception.
- **Never:** If client has transaction, throw an exception (new in EJB 1.1)

Concurrent transaction isolation is dictated by the isolation level. The isolation level is no longer in the EJB 1.1 specification. However, WebSphere Application Server 4.0 continues to support the isolation level as an IBM extension. The isolation levels that are supported are Read, Repeatable-Read, Read-Committed, and Serializable.

Two JDBC drivers are available for the iSeries server that support JTA 1.0:

- **IBM Toolbox for Java JDBC driver:**
  `com.ibm.as400.access.AS400JDBCConnectionPoolDataSource`
- **IBM Developer Kit for Java (native JDBC driver):**
  `com.ibm.db2.jdbc.app.DB2StdXADatasource`

When running on the iSeries server, such as under WebSphere Application Server, use the native driver. It performs better. The toolbox JDBC driver should be used to access iSeries database tables from remote systems.

### 1.4.5 JavaBean Activation Framework (JAF)

JAF is a MIME type support used by JavaMail to handle data in e-mail messages. It’s not intended for typical application use. However, advanced e-mail interactions may require using it. Refer to 1.5.3, “JavaMail” on page 14.

The J2EE 1.2 specification requires support for JAF 1.0, as described in Table 1-1 on page 5.
1.5 J2EE communication

The essence of J2EE is the definition of a distributed, object-oriented infrastructure. Components need to communicate with each other in this distributed world. Therefore, the containers need to provide the appropriate communication mechanisms to make this happen. An example of the communications included in the J2EE standards are RMI/IIOP for remote method calls, JavaMail for programmatic access to e-mail, and JMS for accessing messaging technologies.

1.5.1 Remote method invocation (RMI/IIOP)

Remote method invocation (RMI) turns local method calls into remote method calls. From an application view, it just makes a method call. The infrastructure handles calling the remote method and receiving the response. EJB clients use remote method invocation over Internet Inter-ORB Protocol (RMI/IIOP) to communicate with other EJBs.

RMI support provides the rmic compiler, which generates:
- Client and server stubs that work with any Object Request Broker (ORB).
- An IDL file compatible with the RMI interface. To create a C++ server object, an Application Component Provider uses an IDL compiler to produce the server stub and skeleton for the server object.
- A CORBA API and ORB.

An Application Component Provider defines the interface of a remote object in Interface Definition Language (IDL). Then it uses an IDL compiler to generate client and server stubs that connect object implementations to an ORB, a library that enables CORBA objects to locate and communicate with one another. ORBs communicate with each other using the Internet Inter-ORB Protocol (IIOP).

The J2EE 1.2 specification supports RMI/IIOP level 1.0, as described in Table 1-1 on page 5.

1.5.2 Java Messaging Service

Java Messaging Service is a reliable interface for asynchronously sending and receiving messages. Messaging communication is:
- Peer to peer: Any client can send/receive messages to or from another client.
- Loosely coupled: The sender and receiver do not have to be available at the same time to communicate.
- Asynchronous: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- Reliable: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or receive duplicate messages.
- Point-to-point messaging: A client sends a message to the message queue of another client. Each message has one receiver.
- Publish-subscribe messaging: Clients subscribe to a well-known node called a topic. All subscribers receive a message sent to the topic. The topic adjusts as publishers and subscribers activate and deactivate.
The J2EE 1.2 specification requires support for JMS 1.0, as described in Table 1-1 on page 5. JMS 1.0 requires JMS transactions interacting with a single JMS resource provider only (one-phase commit). J2EE 1.3 requires support for distributed JMS Transaction (JMS/XA) two-phase commit. WebSphere Application Server Version 4.0 Advanced Edition supports JMS/XA. JMS/XA is not available in WebSphere Application Server Version 4.0 Advanced Single Server Edition. JMS/XA requires MQSeries.

1.5.3 JavaMail

JavaMail provides APIs for reading, composing, and sending e-mail. The APIs model a mail system, allowing Web components to interact with an e-mail system.

APIs require service providers to implement specific protocols:
- Send Mail: SMTP
- Receive/Store Mail: POP3, IMAP
- JAF to handle non-plain text mail content (MIME, URL, file attachments)

The J2EE 1.2 specification requires support for JavaMail 1.1, as described in Table 1-1 on page 5.

1.6 J2EE packaging and deployment

J2EE components are grouped in modules. Each module has its own deployment descriptor. Applications can include one or more modules, as shown in Figure 1-7 and as explained here:
- EJB modules group related EJBs in a single module and are packaged in Java Archive (JAR) files. The EJB JAR file contains all the EJBs and a deployment descriptor.
- Web modules group servlet class files, JSPs, HTML files, and images. They are packaged in Web Application Archive (WAR) files. The WAR file contains the Web components and a deployment descriptor.
- Application client modules are also packaged in Java Archive (JAR) files. The application client JAR contains all the Java classes of the application client and a deployment descriptor.

A J2EE application is packaged in an Enterprise Archive (EAR) file. The application has a deployment descriptor that allows configuration to a specific container’s environment when deployed, as shown in Figure 1-7.
1.6.1 J2EE deployment descriptor

Deployment descriptors are used to communicate the needs of application components to the deployer. The deployment descriptor is a contract between the application component provider or assembler and the deployer. The application component provider or assembler is required to specify the application component's external resource requirements, security requirements, environment parameters, and so on in the component's deployment descriptor. The J2EE product provider is required to provide a deployment tool that interprets the J2EE deployment descriptors and allows the deployer to map the application component's requirements to the capabilities of a specific J2EE product and environment.

A module's deployment descriptor, an XML-based file, dictates interaction with the container. The deployment descriptor describes the components of the module and the environment that the container is to provide for the components. Each module and ear file has a deployment descriptor.

When using the IBM WebSphere Software Platform foundation and tools, such as WebSphere Application Assembly Tool (AAT) or WebSphere Studio Application Developer, the deployment descriptor can be automatically created and verified by the WebSphere deployment tools.

The deployment descriptor can also be manually created or edited. However, this is not recommended because it is tedious and error prone.
EJB deployment descriptor
The deployment descriptor file associated with EJB components is named *ejb-jar.xml*. There is only one deployment descriptor for all the EJBs in the JAR file. Previously, in the EJB 1.0 specification, each enterprise bean had its own deployment descriptor.

The EJB deployment descriptor defines the following information for each EJB:

- Home interface, remote interface and bean name
- For session beans, the session type
- For entity beans, the persistence type
  - For container-managed persistence (CMP), the primary key and container managed fields
- Transaction type
- Environment entries
- EJB references

The EJB deployment descriptor defines the following common assembly information:

- The security roles used in the EJB, the mapping of roles to methods
- For container managed transactions, the mapping of EJB methods to transaction attributes

WAR deployment descriptor
The deployment descriptor file associated with Web application components (WAR) is named *web.xml*. It defines the following information.

- Servlets, JSP, static resources: Servlet URL pattern, class/file, attributes
- Security constraints: For each URL pattern, the assigned security roles for methods
- Login configuration
- EJB and resource references
- Security roles for this Web module

EAR deployment descriptor
The deployment descriptor file associated with an enterprise application (EAR) modules is named *application.xml*. It defines the following information:

- All the modules packages in the EAR file: EJB, WAR, Application Client
- Defined security roles

Figure 1-8 shows an EAR deployment descriptor example that defines modules for EJB and Web components.
1.7 J2EE platform roles

It is important to understand the J2EE roles because it helps to better understand the WebSphere tooling and how it addresses the separation of roles. Figure 1-9 and the following list describe the J2EE roles with a distinct focus on products, applications, and runtime:

- **Product provider**: The company that designs and makes available for purchase the J2EE platform, APIs, and other features defined in the J2EE specification. Product providers are typically operating system, database system, application server, or Web server vendors who implement the J2EE platform. IBM is a J2EE product provider with WebSphere 4.0.

- **Tools provider**: The person or company who makes development, assembly, and packaging tools used by component providers, assemblers, and deployers. IBM is a Tool Provider with VisualAge for Java and WebSphere Studio Application Developer.

- **Component provider**: The company or person who creates Web components, enterprise beans, applets, or application clients for use in J2EE applications.

- **Application assembler**: The company or person who gets application component JAR files from component providers and assembles them into a J2EE application EAR files. The assembler or deployer can edit the deployment descriptor directly or use tools that correctly add and edit XML tags. A software developer performs the following tasks to deliver an EAR file containing the J2EE application:
  - Assembles EJB JAR and Web components (WAR) files created in the previous phases into a J2EE application (EAR) file.
  - Specifies the deployment descriptor for the J2EE application.
  - Verifies that the contents of the EAR file are well formed and comply with the J2EE specification.

- **Deployer**: Deploys or installs the J2EE application (.ear) into the J2EE server platform.

- **System administrator**: Administers the computing and networking infrastructure where J2EE applications run, and oversees the runtime environment.
Figure 1-9  J2EE platform roles

Figure 1-10 describes the J2EE packaging and deployment tasks associated with the J2EE roles:

- **Application component provider**: Specifies component deployment descriptors and packages components into modules.

- **Application assembler**: Resolves dependencies between deployment descriptor elements in different modules and assembles modules into larger deployment units.

- **Deployer**: Customizes deployment descriptor elements for the environment and installs deployment units into servers.

Figure 1-10  J2EE packaging and deployment tasks
1.8 J2EE additional resources

You can find additional J2EE information, such as the J2EE Specification, J2EE Blueprints, J2EE Reference Implementation, Java PetStore application, and articles and discussions on J2EE at: http://java.sun.com/j2ee
Servlet and JSP development using VisualAge for Java

Servlets are server side Java programs that run inside request/response-oriented servers, such as WebSphere Application Server, and extend them in some manner. For example, a servlet may be responsible for taking data in an HTML order entry form and applying the business logic used to update a company's order database. Servlets are to servers as applets are to browsers.

JavaServer Pages (JSP) extend the servlet architecture. They provide an easy way to separate business logic from the presentation of information. They allow you to access server-side components from Web pages while separating the presentation of dynamic content from the generation of that content. You do not need to know the Java programming language to use JavaServer Pages. JSPs give you the ability to access the feature set of Java in an easy-to-use tagging framework that generates dynamic content for the Web.

This chapter covers iSeries servlet and JSP application development for the WebSphere Application Server V4.0 environment using VisualAge for Java. The servlet and JSP support are the same for WebSphere Application Server Version 4.0 Advanced Single Server Edition and WebSphere Application Server Version 4.0 Advanced Edition.
2.1 Servlet support in WebSphere Advanced Edition 4.0

Like version 3.5, the WebSphere Application Server 4.0 environment exists independently of the HTTP server using its resources. WebSphere Application Server 4.0 also provides an embedded HTTP server. This internal HTTP server can be used for testing purposes but should not be used in a production environment (Figure 2-1.)

![Diagram of WebSphere V4.0: Typical servlet scenarios](image)

As you look at Figure 2-1, notice how WebSphere Application Server works. For both WebSphere Application Server Version 4.0 Advanced Single Server Edition and WebSphere Application Server Version 4.0 Advanced Edition, if you are using an external HTTP server, there is an additional step in the process. This step is the Web server interface to the application server. In this case, the HTTP server and Java application server run as separate jobs in separate subsystems. If you use the internal HTTP server, there is only the application server running in its subsystem.

WebSphere Application Server V4.0 implements the Java Servlet 2.2 and JavaServer Pages 1.1 specification. The JavaServer Web Development Kit (JSWDK) is the reference implementation for the JavaServer Pages technology and the Java servlet API. The JSWDK is available at no charge from the Sun Microsystems Web site. For a complete definition of the Sun Java servlet and JSP API specification, refer to the Sun online documentation at the following Web sites:

- [http://java.sun.com/products/servlet/2.2/](http://java.sun.com/products/servlet/2.2/)

2.1.1 IBM development environments for WebSphere applications

There are a number of IBM application development environments that can be used for developing WebSphere applications. Two of the most popular ones for an iSeries environment are:

- **VisualAge for Java Enterprise Edition 4.0**: Provides a servlet and Enterprise JavaBean development environment
In writing this chapter, we used VisualAge for Java Enterprise Edition 4.0 as our development environment. We required the ability to easily integrate servlets with both the IBM Toolbox for Java classes and Enterprise JavaBeans. While IBM WebSphere Studio is a simpler tool to use to achieve servlet/JSP generation, it would have required us to manually modify the generated servlets and import them into VisualAge for Java for integration into our ultimate integrated JSP, servlet, and EJB environment.

We also used a new IBM development tool – WebSphere Studio Application Developer V4.0 (Application Developer). It integrates the functionalities of both VisualAge for Java and WebSphere Studio and provides an advanced development environment for J2EE application development. It improves your productivity by providing an easy way to package J2EE compliant applications. We use Application Developer to build Web applications in Chapter 5, “Building Java servlets and JSPs with WebSphere Studio Application Developer” on page 175.

2.2 Introduction to servlets

Before you develop servlet applications, you need some background information about the basic concepts behind servlets. Figure 2-2 shows an overview of the servlet architecture and how servlets communicate with a browser. Communication between a browser and a servlet application follows this sequence:

1. The client (browser) sends a request to the HTTP server.
2. The HTTP server forwards the request to the servlet.
3. The servlet receives the request and generates a response by accessing resources and passes the response back to the HTTP server. The response usually contains HTML or other data that can be processed by the client.
4. The HTTP server sends the response to the calling client (browser).
5. The browser renders the data.

Servlets are modules that run inside request/response-oriented servers, such as the IBM WebSphere Application Server for iSeries. The servlets extend the servers in some manner. For example, a servlet may be responsible for taking data from an HTML order form and applying the company’s business logic to update a company’s order database.
Servlets can provide services for HTTP servers or extensions to HTTP servers. They can perform functions that are equivalent to CGI programs, server-side includes, and server-side APIs (NSAPI and ISAPI). Servlets can also be used as a powerful substitution for CGI scripts. Although servlets can provide services outside of the HTTP environment, the HTTP servlet is of most interest.

Servlet support is provided in two packages:
- `javax.servlet.*`: Provides support for generic, protocol-independent servlets
- `javax.servlet.http.*`: Provides support for HTTP-specific functionality

The classes and interfaces are organized as shown in Figure 2-3.

![Servlet hierarchy](image)

Servlets are a standard extension to Java, which means that servlets are part of the Java language, but they are not part of the core Java API. Therefore, while they may work with any Java Virtual Machine, servlet classes may not be bundled with all Java Virtual Machines.

You can obtain the two packages that are required to develop servlets, `javax.servlet` and `javax.servlet.http`, at the Javasoft Web site at: [http://www.javasoft.com](http://www.javasoft.com)

With VisualAge for Java, these packages are available in the Sun JSDK class libraries project.

Servlets are usually used in the HTTP environment. For this reason, the HttpServlet class is extended, which, in turn, extends the GenericServlet class. To respond to a client request, the HttpServlet must be extended and override one or more of the following methods:
- `doGet()`: To support HTTP GET requests
- `doPost()`: To support HTTP POST requests
- `doDelete()`: To support HTTP DELETE requests
- `doPut()`: To support HTTP PUT requests

Usually, a servlet only overrides the `doGet()` and `doPost()` methods.
2.2.1 Simple servlet example

When accessed from a Web browser, the servlet in Example 2-1 generates the HTML page shown in Example 2-2.

Example 2-1  MyHelloWorldServlet

```java
package nservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyHelloWorldServlet extends javax.servlet.http.HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<head><title>My Hello World</title></head>");
        out.println("<body>");
        out.println("<P><CENTER><B><FONT SIZE=+3 COLOR="RED">Hello World</FONT></B></P>");
        out.println("</body></html>");
    }
}
```

Example 2-2  Hello World HTML source

```html
<html>
<head><title>Hello World</title></head>
<body>
<p><center><font size=+3 color=red>Hello World</font></p>
</body></html>
```

The code starts with three import statements. The `java.io` package is a standard package in the core Java platform. However, the two other packages, `javax.servlet` and `javax.servlet.http`, are particular to the servlet API. The `javax.servlet` package provides the general classes and interfaces for servlets. The `javax.servlet.http` package provides HTTP-specific classes and interfaces for servlets.

This servlet extends the `javax.servlet.http.HttpServlet` and overrides the `doGet()` method. By extending the abstract `HttpServlet` class of the `javax.servlet.http` package, the servlet gains access to specialized methods for handling the HTTP protocol. Since `HttpServlet` is an abstract class, classes that extend the `HttpServlet` class must override at least one method of the `HttpServlet` class. `MyHelloWorldServlet` overrides the `doGet()` method.

When the server receives a GET request for the servlet, the server calls the `service()` method of the servlet, passing it two objects as parameters:

- `HttpServletRequest` (the request object)
- `HttpServletResponse` (the response object)
The `service()` method of the servlet routes the GET request to the `doGet()` method of the servlet. The servlet communicates with the server and ultimately with the client through these request and response objects.

The `HttpServletRequest` object represents the request of the client. This object supplies the servlet with methods to retrieve HTTP protocol specified header information, the parameters for this request, information about the client, and so forth.

The `HttpServletResponse` object represents the servlet response to the client. The servlet invokes the methods of the response object to send the response that it has prepared back to the client. The servlet can also use these methods to set HTTP response headers.

The servlet then calls the `getWriter()` method of the response object to get a print writer for writing formatted text responses. It uses the returned `PrintWriter` object to generate the HTML text.

When the servlet is called from the browser, you see the servlet output as shown in Figure 2-4.

![Hello World](image)

**Figure 2-4  MyHelloWorldServlet running on the iSeries server**

### 2.3 Setting up VisualAge for Java to develop and test servlets

VisualAge for Java offers an Integrated Development Environment (IDE). It provides some very useful capabilities for developing and testing servlets and JavaServer Pages.

VisualAge for Java Enterprise Edition 4.0 provides a test version of the WebSphere Application Server. It is called the `WebSphere Test Environment` (WTE). It includes an application server that is integrated right into the integrated development environment. It includes a servlet engine, a persistent name server, a JSP execution monitor, and a DataSource configuration wizard.

#### 2.3.1 Loading the required features

To work with VisualAge for Java, you must first load the necessary features from the repository into the VisualAge for Java work space. To perform a feature installation, follow these steps:

1. Select `File-> Quick Start-> Features-> Add Features`.
2. Select the following features:
   - IBM WebSphere Test Environment
   - IBM Enterprise Toolkit for AS/400
3. Click `OK` to load the features into the workspace.
2.3.2 Using the WebSphere Test Environment

To use the WebSphere Test Environment in VisualAge for Java 4.0, complete the following steps:

1. Install VisualAge for Java Enterprise Edition, Version 4.0. Then, install VisualAge for Java Enterprise Edition's latest patches. During the Feature Install installation process, you are prompted to indicate the document root on your system (the document root is where your Web resources, including HTML files and JSP files, are stored). Because there is minimal error checking, make sure that the document root directory exists and is valid.

   **Note:** If you want to change the document root later, you can change it in the doc.properties file. This file is located in the \\ide\project_resources\IBM WebSphere Test Environment\properties\server\servlet\httpservice\ subdirectory.

2. After the installation process is completed, restart VisualAge for Java. The features are automatically imported into the repository.

3. Launch the WebSphere Application Server Test Environment. To do this, in the VisualAge for Java Workbench, select **Workspace-> Tools-> WebSphere Test Environment** (Figure 2-5).

![Figure 2-5 Starting the WebSphere Test Environment](image)

4. The WebSphere Test Environment starts. Since this is the first time that it is started, it will take more time than normal to start. Be patient.

5. Once the WTE starts, as shown in Figure 2-6, you can click **Start Servlet Engine** to start the servlet engine.
You can now test servlets inside VisualAge for Java. You can use the VisualAge for Java debugger to monitor and debug the servlet Java code interactively.

### 2.3.3 Testing the servlet under VisualAge for Java Enterprise Edition 4.0

Once the WebSphere Test Environment starts and the servlet engine is active, you can run the servlet in a Web browser. The servlet, JSP files, and HTML files from the designated document root can be served. To run the MyHelloWorldServlet, enter either of the following URLs:

- `http://127.0.0.1:8080/servlet/nservlets.MyHelloWorldServlet`

The MyHelloWorld servlet page appears as shown in Figure 2-7.

### 2.3.4 Exporting class files to a JAR file

In WebSphere Application Server V3.5, you can export the class files to the iSeries Integrated File System (IFS) directly and deploy them into the WebSphere Application Server environment. In WebSphere Application Server V4.0, however, this is not supported. You use the WebSphere Application Assembly Tool (AAT) to create a J2EE Web application that can
be installed on WebSphere Application Server 4.0. For the Java classes in VisualAge for Java, we first export the files to a JAR file and then we can use it as the source for AAT. See Chapter 3, “WebSphere V4.0 assembly and deployment tools” on page 75, for more information about using AAT.

Here is how to export Java classes into a JAR file:

1. Select the **MyHelloWorldServlet** class file in the VisualAge for Java workbench.
2. Right-click and select **Export**.
3. In the Export SmartGuide, select the **Jar File** radio button.
4. On the next page, enter the JAR file name and directory (here we store it in c:\ws\ws\ws\ws.jar), as shown in Figure 2-8.
5. Click **Finish**. The JAR file is stored on the local drive.

In Chapter 3, “WebSphere V4.0 assembly and deployment tools” on page 75, we use AAT to assemble it into a J2EE Web application.

![Figure 2-8 Exporting Java classes into a JAR file](image)

### 2.4 Using JDBC to access an iSeries database

This example builds a servlet that uses JDBC to access a database table on the iSeries server. The table that we access is named **ITEM**. It has the columns shown in Table 2-1. It is found in a library named **APILIB**.
Table 2-1 Item file description

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Length</th>
<th>Decimals</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IID</td>
<td>Item ID</td>
<td>6</td>
<td>0</td>
<td>Character</td>
</tr>
<tr>
<td>INAME</td>
<td>Item Name</td>
<td>24</td>
<td>0</td>
<td>Character</td>
</tr>
<tr>
<td>IPRICE</td>
<td>Price</td>
<td>5</td>
<td>2</td>
<td>Packed</td>
</tr>
<tr>
<td>IDATA</td>
<td>Item Information</td>
<td>50</td>
<td>0</td>
<td>Character</td>
</tr>
</tbody>
</table>

Figure 2-9 shows the output of the servlet. Please note that, by calling the servlet in this manner, the HTTP request that the servlet receives is the default GET request.

![Figure 2-9 The results of running ItemServlet](image)

2.4.1 The architecture of the sample application

This section shows you the benefit of a proper application architecture. That is, you can use the same functionality for applets, servlets, and applications with only minor changes. To demonstrate this, we structure the example application as shown in Figure 2-10.
Figure 2-10  Application architecture

The application consists of multiple packages:

- **access**: This package contains the JDBCItemCatalog class that is responsible for database access. The database access is implemented using JDBC. The class contains one method to connect to the database, `connectToDB()`, and two methods, `getRecord()` and `getAllV()`, to retrieve data from the databases.

- **nservlets**: This package contains classes that provide the end-user interface. The servlet examples implement the HttpServlet abstract class. The ItemServlet class in this package manages all the user interface-oriented work, such as getting and interpreting the user requests that are received in the form of HTTP requests and generating the response in the form of HTML. To obtain the items information that is included in the generated HTML, the ItemServlet class forwards the database access work to the JDBCItemCatalog class.

### 2.4.2 JDBCItemCatalogSupport class

The JDBCItemCatalogSupport class is the super class for the JDBC-based access classes. It adds a few methods that are shared among several JDBC-based access classes.

**JDBCItemCatalogSupport.getRows() method**

Only one JDBCItemCatalogSupport method is relevant for the JDBCItemCatalog class. The `getRows()` method is shown in Example 2-3.

#### Example 2-3  JDBCItemCatalogSupport.getRows() method

```java
public java.util.Vector getRows(java.sql.ResultSet aResultSet) {
    java.util.Vector aDataVector = new Vector();

    try {
        while (aResultSet.next()) {
            String[] data = new String[4];
            data[0] = aResultSet.getString(1); // Item ID
            data[1] = aResultSet.getString(2); // Item Name
            data[2] = aResultSet.getBigDecimal(3, 2).toString(); // price
            data[3] = aResultSet.getString(4); // Item Information
            // add the data element (String[]) to the vector
            aDataVector.addElement(data);
```
To make our code more generic in nature, we use vectors to pass information. The `getRows()` method accepts an SQL ResultSet object and returns a vector. It loops through the ResultSet object. Each row of the ResultSet object is converted into an array of strings. The array of strings is added to the vector.

### 2.4.3 JDBCItemCatalog class

The JDBCItemCatalog class extends JDBCCatalogSupport and defines three private variables as shown in Example 2-4:

- **dbConnection**: A `java.sql.Connection` object that represents a session with a specific database. Within the context of a Connection object, SQL statements are executed and results are returned.
- **psAllRecord**: A `java.sql.PreparedStatement` object that represents a precompiled SQL statement that is stored in the PreparedStatement object. This object can then be used to efficiently run this statement multiple times. We use this PreparedStatement object to retrieve all the Item database rows.
- **psSingleRecord**: A `java.sql.PreparedStatement` object used to retrieve one row from the Item database.

**Example 2-4   JDBCItemCatalog class**

```java
package access;
import java.util.*;
import java.sql.*;
public class JDBCItemsCatalog extends JDBCCatalogSupport{

private java.sql.Connection dbConnection;
private java.sql.PreparedStatement psAllRecord;
private java.sql.PreparedStatement psSingleRecord;
```

**JDBCItemCatalog.connectToDB() method**

The `connectToDB()` method (Example 2-5) handles the connection to the iSeries server.

**Example 2-5   JDBCItemCatalog.connectToDB() method**

```java
public String connectToDB(String url, String userid, String password,String driver) {
    // Create a properties object for JDBC connection
    Properties jdbcProperties = new Properties();
    // Set the properties for the JDBC connection
```
jdbcProperties.put("user", userid);
jdbcProperties.put("password", password);
jdbcProperties.put("naming", "sql");
jdbcProperties.put("errors", "full");
jdbcProperties.put("date format", "iso");
jdbcProperties.put("extended dynamic", "true");
jdbcProperties.put("package", "ServTest");

try {
    try {
        Driver dr = (Driver)Class.forName(driver).newInstance();
    } catch (Exception ex) {... return "cannot register JDBC driver";}
    dbConnection = DriverManager.getConnection(url, jdbcProperties); //toolbox
    psAllRecord = dbConnection.prepareStatement("select * from item");
    psSingleRecord = dbConnection.prepareStatement("SELECT * FROM ITEM WHERE IID = ?");
} catch (SQLException ex) {... return "connection failed sql exception";}
return "connection successful";

The connectToDB() method accepts the four parameters shown in Table 2-2 and returns a String containing a short information message to indicate a successful connection or an error message.

Table 2-2   The parameters of the JDBC driver

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>JDBC driver</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>A database URL in the form jdbc:subprotocol:subname</td>
<td>Toolbox</td>
<td>jdbc:as400://as05/APILIB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Native</td>
<td>jdbc:db2://as05/APILIB</td>
</tr>
<tr>
<td>userid</td>
<td>A valid user ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>A valid password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>driver</td>
<td>The name of the class that implements the JDBC driver to use</td>
<td>Toolbox</td>
<td>com.ibm.as400.access.AS400JDBCDriver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Native</td>
<td>com.ibm.db2.jdbc.app.D82Driver</td>
</tr>
</tbody>
</table>

To make it easier to manage the properties of the JDBC connection, we use a java.util.Properties object and set the different JDBC connection properties, such as SQL naming convention, date format, and so on.

Before a JDBC connection can be made, you need to load the JDBC driver and register it with the driver manager. These two operations can be done in one statement:

Class.forName(driver);

The JDBC documentation states, "When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager."

The next step is to use the driver manager to get a JDBC connection. We use the prepareStatement() method of the connection object to create two SQL-prepared statements. One SQL statement (psALLRecord) is used to retrieve all the rows from the ITEM table, and the second (psSingleRecord) is used to retrieve one row from the ITEM table. To choose the specific row dynamically, while gaining the performance benefits of compiled SQL statements, we use a parameter marker in the SQL statement.
**JDBCItemCatalog.getAllV() method**
The `getAllV()` method executes the SQL query previously prepared in the `psAllRecord` preparedStatement object and uses the `getRows()` method inherited from the `JDBCCatalogSupport` class (Example 2-6) to convert the returned result set into a vector.

```java
Example 2-6   JDBCItemCatalog.getAllV() method
public java.util.Vector getAllV() {
    java.util.Vector aDataVector = new Vector();
    java.sql.ResultSet aResultSet = null;
    try {
        aResultSet = psAllRecord.executeQuery();
        aDataVector = getRows(aResultSet);
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    return aDataVector;
}
```

**JDBCItemCatalog.getRecord() method**
The `getRecord()` method sets the part number value for the parameter marker using the `setString()` method of the `psSingleRecord` preparedStatement object. Then, it executes the SQL query and uses the `getRows()` method, shown in Example 2-7, to convert the returned result set into a vector.

```java
Example 2-7   JDBCItemCatalog.getRecord() method
public java.util.Vector getRecord(String partNo) {
    java.util.Vector aDataVector = new Vector();
    java.sql.ResultSet aResultSet = null;
    try {
        psSingleRecord.setString(1, partNo);
        aResultSet = psSingleRecord.executeQuery();
        aDataVector = getRows(aResultSet);
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    return aDataVector;
}
```

### 2.4.4 Testing the application in the scrapbook
Before we implement the servlet, we want to make sure the data access bean works properly. There is a function in VisualAge for Java called the **scrapbook**, which can be used to test the class without a user interface.

We open the scrapbook from the Workbench window menu and enter the code shown in Example 2-8.
Example 2-8  Code in scrapbook

```java
access.JDBCItemsCatalog aItemsCatalog = new access.JDBCItemsCatalog();
aItemsCatalog.connectToDB("jdbc:as400://system/teamxx","user","password",
   "com.ibm.as400.access.AS400JDBCDriver");
java.util.Vector itemsVector = aItemsCatalog.getAllV();
java.util.Enumeration items = itemsVector.elements();
while (items.hasMoreElements()) {
   String[] aItem = ((String[]) items.nextElement());
}
```

Let's go through the code. First, it creates a new instance of the JDBCItemsCatalog class named aItemsCatalog. Then it executes the `connectToDB` method of the JDBCItemsCatalog class. Once connected, it executes the `getAllV` method of the JDBCItemsCatalog class to retrieve all the rows from the Items Table and return them in a vector named itemsVector. Then it uses the Enumeration `nextElement` method to read each row from an Enumeration and store the information in a String array named aItem and print them on the console.

As shown in Figure 2-11, to execute this code, we need to enter the valid values for:

- **URL**
- **User**
- **Password**
- **JDBC driver**

![Scrapbook in VisualAge for Java](image)

**Figure 2-11  Scrapbook in VisualAge for Java**

We open the Console window of VisualAge for Java (from Window menu) to see the output messages.

We select all the text and press either Ctrl-E to run the code or use the pop-up menu and select the Run option. As shown in Figure 2-12, you see the rows retrieved from the database and displayed on the console, which means the data access bean works.
2.4.5 ItemServlet class

The ItemServlet class shown in Example 2-9 is the servlet implementation. It imports the generic servlet support package, javax.servlet, and the HTTP servlet support package, javax.servlet.http. This class extends the SuperServlet support class.

The SuperServlet class is a support class that we provide in the nservlets package. We use it to help control the logging of messages and to read information from an external properties file. It provides these methods:

- **getServerRoot()**: Returns the base URL of the server upon which this servlet is running, making it easy to dynamically build URLs.
- **flexLog()**: Writes a log entry either to the servlet log or to the system log as defined in the class initialization.
- **getMainPropertiesFile()**: Returns a Properties object loaded from the config.properties file.
- **outputHeader()**: Writes the HTML Header to the specified PrintWriter.

The ItemServlet class keeps a reference to the JDBCItemsCatalog class that does all the database-related work.

```java
Example 2-9  ItemServlet class
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.util.Properties;

public class ItemServlet extends SuperServlet {
    // Implementation
```
private access.JDBCItemsCatalog aJDBCItemsCatalog;

getItemServlet.init() method
The init() method is called automatically only once during the servlet life cycle by WebSphere Application Server when it loads the servlet. It is guaranteed to finish before any service requests are accepted. After successful initialization, WebSphere does not call the init() method again, unless it reloads the servlet after it has unloaded and destroyed it.

The init() method shown in Example 2-10 starts by calling super.init(config) and passing it a ServletConfig object. This call is very important because the init method should save the ServletConfig object so that it can be returned by the getServletConfig method. Calling the super.init() method saves a reference to the config object for future use.

Example 2-10 ItemServlet.init() method

```java
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    aJDBCItemsCatalog = new access.JDBCItemsCatalog();
    String url = getInitParameter("url");
    String userid = getInitParameter("user");
    String password = getInitParameter("password");
    String driver = getInitParameter("driver");
    if ((url == null) || (userid == null) || (driver == null) || (password == null)) {
        throw new ServletException("Could not retrieve all the Initial Parameters, cannot continue");
    }
    String returnValue = aJDBCItemsCatalog.connectToDB(url, userid, password, driver);
    log("ItemServlet: return from connect = " + returnValue);
    return;
}
```

To connect to the database, we perform these steps:
1. Load the initial parameters for the database URL, user ID, password, and JDBC driver.
2. Connect to the database using the JDBCItemsCatalog.connectToDB() method.

The javax.servlet.GenericServlet.getInitParameter() method returns a string containing the value of the named initialization parameter. Or it may return null if the requested parameter does not exist.

After the init() method returns successfully, the ItemServlet servlet is loaded, the database connection is set, and the two SQL-prepared statements are prepared. The servlet service method is now ready to accept requests from the doPost() method.

getItemServlet.doPost() method
The doPost() method shown in Example 2-11 is called by the servlet service() method to service HTTP POST requests. This method performs the following steps:
1. Sets the response MIME type to text/html.
2. Gets the response PrintWriter to write the HTML output.
3. Gets the value of the request partno input parameter.
The value can be *ALL to indicate that you want to get all the items from the table or a specific item number. If the parameter is null, it defaults to *ALL.

4. Writes the HTML header.
   This work is done by the outputHeader() method from the SuperServlet class.

5. Retrieves the data from the database by handing the work to the proper JDBCItemsCatalog method.
   If the partno parameter is *ALL, the getAllV() method is called. Otherwise, the getRecord() method is called. In both cases, a vector of String arrays is returned. Each String array describes one item and includes item ID, name, price, and description.

6. Writes the HTML body by calling the outputItemInformation() method.

---

Example 2-11  ItemServlet.doPost method

```java
public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {
    // set the MIME type to text/html
    response.setContentType("text/html");
    // create the output stream
    PrintWriter out = response.getWriter();
    try {
        // get the value from the input field named partno (see HTML file)
        // and check if value is valid

        //Only get the first parameter
        String parameter = null;
        if (request.getParameter("partno") == null)
            parameter = "*ALL";
        else
            parameter = request.getParameter("partno");
        Vector parts = null;
        flexLog("ItemServlet: parameter passed in from HTML is: " + parameter);
        // write the HTML header to the output stream
        outputHeader(out, getServletInfo());
        // retrieve all data from the database
        if (parameter.toUpperCase().equals("*ALL")) {
            parts = aJDBCItemsCatalog.getAllV();
        } else
            parts = aJDBCItemsCatalog.getRecord(parameter);

        // write the HTML table to the output stream
        outputItemInformation(out, parts);
        out.close();
        return;
    } catch (Throwable e) {
        printError(out, e);
    }
} // end of doPost()
```
Chapter 2. Servlet and JSP development using VisualAge for Java

**ItemServlet.doGet method**
The doGet() method, shown in Example 2-12, forwards the HTTP GET requests to the doPost method. Since GET is the default HTTP request, invoking the servlet from a URL, such as http://sysname:port/webapp/OrderEntry/ItemServlet, causes the doGet() method to be called. The doGet() method then calls the doPost() method. In this case, the partno parameter is null and defaults to *ALL, which eventually creates an HTML page with information about all the items in the Item file.

**Example 2-12  ItemServlet.doGet() method**

```java
public void doGet(HttpServletRequest request,
                   HttpServletResponse response) throws IOException {
    doPost(request, response);
    return;
}
```

**ItemServlet.outputItemInformation() method**
The method presented in Example 2-13 generates HTML pages that contain a four-column-wide table and populates this table with information from the parts Vector parameter.

**Example 2-13  ItemServlet.outputItemInformation() method**

```java
private void outputItemInformation(PrintWriter out, Vector partsVector) throws IOException {
    flexLog("ItemServlet: outputItemInformation()...");
    Enumeration parts = partsVector.elements();
    out.println("<TABLE BORDER>");
    out.println("<P><CENTER><B><FONT SIZE=+3>Here are the results of your query:</FONT></B></P>");
    out.println("<TR>");
    out.println("<TH>ITEM ID</TH><TH>Item Name</TH><TH>Price</TH><TH>Description</TH>");
    out.println("</TR>");
    while (parts.hasMoreElements()) {
        String[] aPart = ((String[]) parts.nextElement());
        out.print("<TD>" + aPart[0] + "</TD>");
        out.print("<TD>" + aPart[1] + "</TD>");
        out.print("<CENTER>$" + aPart[2] + "</CENTER></TD>");
        out.print("<TD>" + aPart[3] + "</TD>");
        out.println("</TR>");
    } // end while
    out.println("</TABLE>");
    out.println("</BODY></HTML>");
    flexLog("ItemServlet: outputItemInformation() executed.");
} // end outputPartsInformation()
```

**2.4.6 Running the ItemServlet inside VisualAge for Java**

Once we finish the code, we can test the ItemServlet inside VisualAge for Java. The ItemServlet is different than the MyHelloWorld servlet because it uses initial parameters to set runtime values.
VisualAge for Java uses an XML file named `default_app.webapp` to control running servlets in the WebSphere Test Environment. We need to update it to reflect our system parameters. It is found in the `<VAJ install root>/ide/project_resources/IBM WebSphere Test Environment/hosts/default_host/default_app/servlets` directory.

1. Locate the `default_app.webapp` file, and open it with a text-based editor (like Notepad).
2. Add the XML tags for the ItemServlet.

See Example 2-14.

**Example 2-14  Adding a servlet to the default_app.webapp file**

```xml
<servlet>
  <name>ItemServlet</name>
  <description>Item Servlet</description>
  <code>nservlets.ItemServlet</code>
  <servlet-path>/webapp/OrderEntry/ItemServlet</servlet-path>
  <init-parameter>
    <name>url</name>
    <value>jdbc:as400://AS05/apilib</value>
  </init-parameter>
  <init-parameter>
    <name>driver</name>
    <value>com.ibm.as400.access.AS400JDBCDriver</value>
  </init-parameter>
  <init-parameter>
    <name>user</name>
    <value>team43</value>
  </init-parameter>
  <init-parameter>
    <name>password</name>
    <value>win4ibm</value>
  </init-parameter>
  <autostart>true</autostart>
</servlet>
```

We make sure the parameters like iSeries server name, userid, and password are correct. We save the changes and stop the WebSphere Test Servlet Engine in the WebSphere Test Environment Control Center. Once it is stopped, select Servlet Engine-> Edit Class Path. Ensure that IBM Enterprise Toolkit for AS/400 and WebSphere Workshop (the project that includes the access and nservlets packages) are selected. Restart the Servlet Engine.

Open a browser to access the servlet from the following URL:

```
http://localhost:8080/webapp/OrderEntry/ItemServlet
```

The result is shown in Figure 2-13.
2.4.7 Exporting the servlet from VisualAge for Java

Similar to MyHelloWorldServlet, before we deploy to WebSphere Application Server Version 4.0, we need to export the code into a JAR file. The JAR file can be used by the Application Assembly Tool to build a Web application. In the VisualAge for Java Workbench, select the access and nservlets packages (use the Ctrl key to select both) and export them. The steps are similar with the steps in 2.3.4, “Exporting class files to a JAR file” on page 28. We export to a JAR file named wswsitem.jar. In Chapter 3, “WebSphere V4.0 assembly and deployment tools” on page 75, we use AAT to create a Web application that we install in WebSphere Application Server 4.0.

2.5 Database connection pools

It is essential to understand and feel comfortable using database connection pools when creating servlets that run under WebSphere Application Server Version 4.0 Advanced Edition or WebSphere Application Server Version 4.0 Advanced Single Server Edition.

WebSphere Application Server 4.0 uses the JDBC 2.0 core and optional packages (formerly JDBC 2.0 standard extension) style and APIs for database connection pooling. Both iSeries JDBC drivers, the native and IBM Toolbox for Java drivers, are JDBC 2.0 compliant.

2.5.1 DataSource version

Here is a simple example that shows how to use DataSource objects. As shown in Example 2-15, the class is named DataBaseConnectivity.
Example 2-15  DataSource example: Imports and variable definitions

```java
package WebSphereV3;

import java.sql.*;
import java.util.*;
import javax.sql.*;
import com.ibm.ejs.dbm.jdbcext.*;
import javax.naming.*;

public class DataBaseConnectivity
{
    // Hard code datasource connectivity information. The information
    // could be brought in at runtime (from an external property file
    // identified) or passed into a method.

    // Use to communicate with connection manager.

    static String userId = "Jeff";
    static String userPassword = "Jeff";
    static String v3DataSourceName = "jdbc/MyDataSource";
    private Context v3Ctx = null;
    private DataSource v3DS = null;
    private Connection dataConn = null;
}
```

Example 2-15 shows the import statements for the DataSource support. These are explained in Table 2-3.

<table>
<thead>
<tr>
<th>Import statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.sql</td>
<td>This package provides the JDBC DB2 optional package support for JNDI and distributed transaction support.</td>
</tr>
<tr>
<td>com.ibm.ejs.dbm.jdbcext</td>
<td>This package provides WebSphere connection pooling support. This package is deprecated in WebSphere 4.0, but is still used in VisualAge for Java.</td>
</tr>
</tbody>
</table>

Notice the two objects required to support the new JDBC 2.0 style lookup for the DataSource support:

- **Context**: Used to access the Java servers JNDI naming service.
- **DataSource**: Used to define the data source.

For simplification, we set the properties using variables.

Example 2-16 shows the default constructor method.

Example 2-16  DataSource example: Retrieving a JDBC connection from the DataSource

```java
public DataBaseConnectivity()
{
    try
    {
        Hashtable parms = new Hashtable();
        parms.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        v3Ctx = new InitialContext(parms);
        v3DS = (DataSource)v3Ctx.lookup(v3DataSourceName);
```
Note the following points as shown in Example 2-16:

- We use JNDI to access the DataSource (or any other object) from the naming space. We create a Hashtable object to hold the necessary parameters to access the naming space.

- We specify com.ibm.ejs.ns.jndi.CNInitialContextFactory as the Context factory object we want to use. VisualAge for Java 4.0 does not support the new WebSphere 4.0 Context Factory (com.ibm.websphere.naming.WsnInitialContextFactory), so we need to use this Context factory to test inside VisualAge for Java. WebSphere Studio Application Developer V4.0 supports the new WebSphere 4.0 Context factory. WebSphere Application Server 4.0 supports both context factories, although CNInitialContextFactory is deprecated.

- We get a Context object for the EJS JNDI interface.

- Using the Context object, we do a JNDI lookup for the logical name of the DataSource.

- The Context lookup returns a Java object, which we cast to a DataSource object. All DataSource names exist in the “jdbc” subcontext of the root.

As shown in Example 2-17, we use the getConnection() method to get the JDBC connection.

```java
Example 2-17  DataSource example getConnection method

public Connection getConnection() throws Exception, SQLException
{
    try
    {
        if (!dataConn.isClosed())
        {
            // Get valid new data connection, if this one has been closed.
            dataConn = v3DS.getConnection(userId, userPassword);
            System.out.println("getConnection: Data Connection rebuilt");
            return dataConn;
        }
        else
        {
            return dataConn;
        }
    }
    catch (SQLException sqle)
    {
        System.out.println("getConnection: SQL Exception caught " + sqle.getMessage());
        sqle.printStackTrace();
        throw sqle;
    }
    catch (Exception e)
    {
        System.out.println("getConnection: Exception caught " + e.getMessage());
    }
}
```
When the JDBC connection is requested, we ensure that we still have the exclusive use of the connection and that it has not been reclaimed by the DataSource or closed by the caller. If the connection has been closed, we attempt to request a new connection. The WebSphere Application Server Version 4.0 DataSource objects have associated time-outs that you can change if you have long running servlets.

**JDBCCatalogSupport class**

Next, we change the JDBC example discussed in 2.4, "Using JDBC to access an iSeries database" on page 29, to use DataSource connection pooling support. We start by adding DataSource support into the JDBCCatalogSupport class and its sub-class JDBCPoolCatalog as illustrated in Figure 2-14.

![Diagram](image)

*Figure 2-14  Connection pool example*

The JDBCCatalogSupport class is the super class for the JDBC-based access classes. It has a few methods that are shared among several JDBC-based access classes.

We start with import statements required for the DataSource support as shown in Example 2-18.

```java
package access;

import java.util.*;
import java.sql.*;
import javax.sql.*;
import com.ibm.ejs.dbm.jdbcext.*;
import javax.naming.*;
```

Since we now work directly with JDBC connections, we enhance the freeConnection and getConnection methods to take advantage of this. As shown in Example 2-19, the freeConnection method now simply closes the JDBC connection and the underlying DataSource connection pooling mechanism reclaim the connection.
Example 2-19  The freeConnection method

```java
public void freeConnection(Connection dataConn)
{
    try {
        if (!dataConn.isClosed())
        {
            dataConn.close();
        }
    }
    catch (SQLException e)
    {
        System.out.println("release connection: "+ e.getMessage());
    }
    return;
}
```

As shown in Example 2-20, within the getConnection method, we now return a JDBC connection object. Notice that the parameter list now consists of the DataSource object and a user ID and password only.

Example 2-20  The getConnection method

```java
public Connection getConnection(DataSource v3DS, String userId, String userPassword)
{
    Connection dataConn = null;
    try {
        System.out.println("JDBCPoolCatalog: retrieving connection");
        dataConn = v3DS.getConnection(userId, userPassword);
    }
    catch (SQLException e)
    {
        System.out.println(e.getMessage());
    }
    return dataConn;
}
```

Only one JDBCCatalogSupport method is relevant for the JDBCPoolCatalog class. The getRows() method is shown in Example 2-21.

Example 2-21  JDBCCatalogSupport.getRows() method

```java
/**
 * getAll method comment.
 */
public java.util.Vector getRows(java.sql.ResultSet aResultSet) {
    java.util.Vector aDataVector = new Vector();
    try {
        //System.out.println("JDBCCatalogCatalog: getRows");
        while (aResultSet.next()) {
            String[] data = new String[4];
            data[0] = aResultSet.getString(1); // Item ID
            data[1] = aResultSet.getString(2); // Item Name
            data[2] = aResultSet.getBigDecimal(3).toString(); // price
            aDataVector.addElement(data);
        }
    }
    return aDataVector;
}
```
The `getRows()` method accepts an SQL ResultSet object and returns a vector. It loops through the ResultSet object. Each row of the ResultSet object is converted into an array of strings. The array of strings is added to the vector.

**JDBCPoolCatalog class**

Next, we look at the JDBCPoolCatalog class. We start with import statements to match those required for the DataSource support as shown in Example 2-22.

```
import java.util.*;
import java.sql.*;
import javax.sql.*;
import com.ibm.ejs.dbm.jdbcext.*;
import javax.naming.*;

public class JDBCPoolCatalog extends JDBCCatalogSupport
{
    private String connUserId = null;
    private String connUserPassword = null;
    private Context v3Ctx = null;
    private DataSource v3DS = null;

    We use the `connectToDB()` method, shown in Example 2-23, to get the JDBC connection. We pass in the DataSource name, the user ID, and password.

```

```
Example 2-23   The connectToDB method

generic String connectToDB(String dataSourceName, String userid, String password)
{
    try
    {
        connUserId = userid;
        connUserPassword = password;
        Hashtable parms = new Hashtable();
        parms.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        v3Ctx = new InitialContext(parms);
        v3DS = (DataSource) v3Ctx.lookup(dataSourceName);
    }
    catch (Exception e)
    {
        System.out.println("JDBCPoolCatalog: (ConnectToDB) " + e.getMessage());
    }

```
In reference to Example 2-23, note these points:

- We use JNDI to get access to the DataSource (or any other object) from the naming space. We create a Hashtable object to hold the necessary parameters to access the naming space.
- We specify com.ibm.ejs.ns.jndi.CNInitialContextFactory as the Context factory object we want to use.
- We get a Context object for the EJS JNDI interface.
- Using the Context object, we run a JNDI lookup for the logical name of the DataSource.
- The Context lookup returns a Java object, which we cast to a DataSource object. All Datasource names exist in the "jdbc" subcontext of the root.

In the getAllV() method shown in Example 2-24, we call the getConnection() method to get the JDBC connection.

**Example 2-24   The getAllV method**

```java
public java.util.Vector getAllV()
{
    java.util.Vector aDataVector = new Vector();
    java.sql.ResultSet aResultSet = null;
    Connection dataConn = null;
    try
    {
        dataConn = getConnection(v3DS, connUserId, connUserPassword);
    }
    catch (Exception e)
    {
        System.out.println("Exception caught: " + e.getMessage());
        e.printStackTrace();
        // throw e;
    }
    PreparedStatement psAllRecord = null;
    try
    {
        psAllRecord = dataConn.prepareStatement("select * from ITEM");
        aResultSet = psAllRecord.executeQuery();
        aDataVector = getRows(aResultSet);
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
    }
    finally
    {
        try
        {
            if (null != psAllRecord)
            {
```
Example 2-25 shows the getRecord method.

Example 2-25 The getRecord method
public java.util.Vector getRecord(String partNo)
{
    java.util.Vector aDataVector = new Vector();
    java.sql.ResultSet aResultSet = null;
    Connection dataConn = getConnection(v3DS, connUserId, connUserPassword);
    PreparedStatement psSingleRecord = null;
    try
    {
        psSingleRecord = dataConn.prepareStatement("SELECT * FROM ITEM WHERE IID = ?");
        psSingleRecord.setString(1, partNo);
        aResultSet = psSingleRecord.executeQuery();
        aDataVector = getRows(aResultSet);
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
    }
    finally
    {
        try
        {
            if (null != psSingleRecord)
            {
                psSingleRecord.close();
            }
            freeConnection(dataConn);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
    return aDataVector;
}

Finally, we show the servlets that use the JDBC Pool class. Example 2-26 shows the init() method for the ItemPoolServlet.
Example 2-26  The init method

```java
public void init(ServletConfig config) throws ServletException {
    super.init(config, SuperServlet.SYSTEM);
    aJDBCItemCatalog = new access.JDBCPoolCatalog();

    // Get JDBC properties
    String datasource = getInitParameter("datasource");
    String userid = getInitParameter("user");
    String password = getInitParameter("password");
    flexLog("DatSource: " + datasource);
    flexLog("UserID: " + userid);
    flexLog("Password: " + password);
    String returnValue = aJDBCItemCatalog.connectToDB(datasource, userid, password);
    flexLog("ItemPoolServlet: return from connect = " + returnValue);
    return;
}
```

We set the servlet initialization parameters to retrieve the name of the DataSource object and pass this, along with the user ID and password, into the database connection method.

### 2.5.2 Running the ItemPoolServlet inside VisualAge for Java

Running this servlet inside VisualAge for Java is similar to the ItemServlet discussed earlier. We need to set up the initial parameters for data source, user, and password. But before that, we configure the data source in the VisualAge for Java WebSphere Test Environment.

We start the WebSphere Test Environment if it is not already started. Click the **Persistent Name Server** and start it. Once it is started, select **DataSource Configuration** to add a new data source, as shown in Figure 2-15.

![Figure 2-15  DataSource configuration](image)

On the following page (shown in Figure 2-16), we create the DataSource named NativeDS and specify the following parameters:
DataSource name: NativeDS
Database Driver: com.ibm.as400.access.AS400JDBCDriver
Database URL: jdbc:as400://sysname/teamxx

We replace sysname with the name of the iSeries server, and replace teamxx with the correct library name.

Figure 2-16  Add DataSource window

The new DataSource appears in the WebSphere Test Environment Control Center as shown in Figure 2-17.

Figure 2-17  DataSource Configuration

Similar to the process in 2.4.6, “Running the ItemServlet inside VisualAge for Java” on page 39, we need to update the default_app.webapp file to reflect the correct DataSource name, user ID and password with a text editor (like Notepad). This is shown in Example 2-27.
Example 2-27  The default_app.webapp file

<servlet>
   <name>ItemPoolServlet</name>
   <description>ItemPoolServlet</description>
   <code>nservlets.ItemPoolServlet</code>
   <servlet-path>/webapp/OrderEntry/ItemPoolServlet</servlet-path>
   <init-parameter>
      <name>datasource</name>
      <value>jdbc/NativeDS</value>
   </init-parameter>
   <init-parameter>
      <name>user</name>
      <value>Team43</value>
   </init-parameter>
   <init-parameter>
      <name>password</name>
      <value>win4ibm</value>
   </init-parameter>
   <autostart>false</autostart>
</servlet>

Once we save the file, we return to the VisualAge for Java WebSphere Test Environment and stop and restart the servlet engine. Now we can access the servlet via the following URL:

http://localhost:8080/webapp/OrderEntry/ItemPoolServlet

You see the results of the query as shown in Figure 2-18.

![Item Pool Servlet - Microsoft Internet Explorer](image)

Here are the results of your query:

<table>
<thead>
<tr>
<th>ITEM ID</th>
<th>Item Name</th>
<th>Price</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>WEB SPHERE RED BOOK</td>
<td>$30.00</td>
<td>SG24-5635</td>
</tr>
<tr>
<td>000002</td>
<td>Radio Controlled Plane</td>
<td>$96.00</td>
<td>DELUXE MODEL</td>
</tr>
<tr>
<td>000003</td>
<td>Change Machine</td>
<td>$32.55</td>
<td>HANDLES SEVERAL CURRENCIES</td>
</tr>
<tr>
<td>000004</td>
<td>Baseball Tickets</td>
<td>$91.14</td>
<td>MINNESOTA TWINS</td>
</tr>
<tr>
<td>000005</td>
<td>Twelve Pac USED Pensil</td>
<td>$50.58</td>
<td>YELLOW IN COLOR</td>
</tr>
<tr>
<td>000006</td>
<td>Over Under Shotgun</td>
<td>$79.66</td>
<td>12 GAUGE</td>
</tr>
<tr>
<td>000007</td>
<td>Feel Good Vitamins</td>
<td>$29.81</td>
<td>A,B,C AND D</td>
</tr>
<tr>
<td>000008</td>
<td>Cross Country Ski Set</td>
<td>$93.00</td>
<td>ENTRY LEVEL</td>
</tr>
<tr>
<td>000009</td>
<td>Rubber Baby Buggy Wheel</td>
<td>$98.71</td>
<td>4 INCH SIZE</td>
</tr>
<tr>
<td>000010</td>
<td>TISO RED BOOK SG24-2152</td>
<td>$50.00</td>
<td>ACCESSING THE AS/400 WITH JAVA</td>
</tr>
<tr>
<td>000011</td>
<td>TISO RED BOOK SG24-2163</td>
<td>$20.87</td>
<td>BUILDING AS/400 APPLICATIONS WITH JAVA</td>
</tr>
</tbody>
</table>

Figure 2-18  ItemPoolServlet
2.5.3 Exporting the ItemPoolServlet servlet from VisualAge for Java

Similar to ItemServlet, before we deploy our new servlet to WebSphere Application Server V4.0, we have to export it into a JAR file that can be used by the Application Assembly Tool to build a Web application. In the VisualAge for Java Workbench, select the access and nservlets packages and export them to a JAR file. See 2.3.4, “Exporting class files to a JAR file” on page 28. We export to a file named wswspool.jar.

2.6 JSP support in WebSphere Version 4.0

WebSphere Application Server 4.0 (both Advanced Single Server Edition and Advanced Edition) implements the Sun Microsystems JavaServer Pages Specification 1.1 and removes the support for the JSP 0.91 and 1.0 specifications. For JSP files written to the 0.91 specification, you must migrate them to the JSP 1.1 specification. There are functional differences between the tags in the JSP 1.0 and JSP 1.1 specification. You do not need to migrate pages written to the JSP 1.0 specification to the newer JSP 1.1 specification to deploy them in WebSphere Application Server V4.0.

For a complete definition of the Sun JSP specification, refer to the Sun online documentation at: http://java.sun.com/products/jsp/download.html

2.6.1 JSP life cycle

The JSP life cycle is similar to the standard servlet life cycle. JSPs are compiled into servlets by a JSP engine and are then run by the standard servlet engine on the server platform. For those working with JSPs for the first time, it can be difficult to visualize just what is happening, when, and where. To help you understand this better, refer to the activity diagram in Figure 2-19, which explains the typical JSP life cycle.

Note: The diagram is by no means exhaustive. You should refer to the WebSphere documentation and JSP specification for further details.

Activity diagrams: The activity diagram format is a simplified Unified Modeling Language (UML) activity diagram. The vertical columns are called swim lanes and denote one strand of application processing. This process can be a single object, a Java thread, or even a user.

Concurrent (or nearly concurrent) processes are shown side by side in swim lanes. Horizontal bars, or synchronization bars, are used to show points where application flow diverges into separate lanes or converges back again. In this example, a good illustration of this is in lane 3, where the JSP is compiled if it is not already compiled.

In this case, the swim lanes segregate actions that are performed by a particular section of the application architecture. Timing is top to bottom and includes some degree of parallelism. For a good, quick introduction to UML notation, see UML Distilled: A Brief Guide to the Standard Object Modeling Language by Martin Fowler.
The steps shown in the activity diagram in Figure 2-19 are explained here in greater detail:

1. At the client side browser, the user enters a URL that references a JSP file. The JSP file goes to the Web server (in this example, it is HTTP server for iSeries, but it could be any other iSeries-based Web server) and requests the named page.

2. When the HTTP server receives a request for a JSP, it passes that request to WebSphere Application Server. WebSphere Application Server passes the request to the JSP engine (PageCompile). Depending on the application architecture used, one of two courses of action take place:
   - If the JSP is the first page of an application, or the page only contains HTML, the JSP is invoked (steps 3, 3a, and 4), the HTML is generated (step 6), and a page is served to the client. In this case, the JSP is a little different from a normal HTML page.
   - If the page contains input controls, and the browser request is an action, the page is processed by the JSP and servlet engines as described below.
3. The served page may contain input fields and buttons. We assume here that the user performs a task, such as filling in the item number and clicking the List all items button in the ItemServlet example.

4. The JSP engine receives the process request. If the JSP has not yet been compiled, or the JSP has changed since the last time it was compiled (based on a file time stamp), step 3a occurs, and the JSP is compiled into a servlet.

5. From this point, the JSP is a servlet and is processed exactly like any other servlet.

6. Any data access that is typically carried out by data access JavaBeans is performed in the iSeries Java Virtual Machine. The results are returned to the JSP or servlet.

7. If any access was requested, the JSP or servlet receives the results of the data access. Otherwise, it receives a request to produce an output based on a bean passed within the data. The JSP or servlet then generates an HTML file based on the logic contained within the data results or bean.

8. The HTML is returned to the HTTP server and presented to the client browser.

9. The client browser only deals with the HTML from the Web server. Therefore, the user is unaware of the complexity of the operations being carried out on the server, making the client browser truly a thin client.

After it is compiled, the servlet is stored in memory on the server. When subsequent requests for that page are made, the server checks to see if the JSP file has changed. If it has not changed, the server uses the servlet stored in memory to generate the response to the client. Because the compiled servlet is stored in memory, the response is very fast. If the JSP file has changed, the server automatically recompiles the page file and replaces the servlet in memory. As with standard servlets, if the initial load time is long, the JSP can be compiled and loaded at server system start time.

### 2.6.2 JSP design

For simplification of National Language Support (NLS) or the good design strategy of separating the presentation layer from the business logic, perform business logic within a servlet and pass the response data to a JavaServer Page for display as shown Figure 2-20.

![Servlet and JSP architecture](image)

*Figure 2-20  Servlet and JSP architecture*
Under the servlet 2.1 and 2.2 API, the \texttt{setAttribute} method of the HttpServletRequest class is used to pass objects between a servlet and a JSP. A new interface, \texttt{javax.servlet.requestDispatcher}, is used for each servlet or JSP to forward the response and request objects to the specific servlet or JSP. This is similar to servlet filtering and shares some of the same restrictions. If a servlet retrieves a ServletOutputStream object or PrintWriter object from its response object, then an \texttt{IllegalStateException} exception is thrown in the application.

### 2.6.3 JSP servlet interface example

This section looks at a very simple example that demonstrates how a servlet interfaces with a JSP. In this example, CallJSP is a servlet that passes information to a JSP named \texttt{outputjsp.jsp} for display. As shown in Figure 2-21, the user client calls the CallJSP servlet. It creates an object named \texttt{DataBean}, which it passes to the \texttt{outputjsp} JavaServer Page. DataBean is a very simple class that provides methods to set a value and get a value. The JSP displays the information (that is passed to it) back to the user client.

![Figure 2-21 Servlet calling a JSP](image)

Example 2-28 shows the \texttt{doGet} method of the CallJSP servlet.

**Example 2-28** First part of the \texttt{doPost} method of the ItemJSPServlet

```java
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, java.io.IOException {
    System.out.println("Inside CallJSP");
    DataBean dataBean = null; // Create an instance of DataBean
    try {
        dataBean = (DataBean) Beans.instantiate(this.getClass().getClassLoader(), "nservlets.DataBean");
    } catch (Exception ex) {
        throw new ServletException("CallJSP: doGet: Can't create bean of class DataBean: " + ex.getMessage());
    }
    // Set some Bean properties (content generation)
```
```java
if (request.getParameter("value") == null)
    dataBean.setValue1("Something Important");
else
    dataBean.setValue1(request.getParameter("value"));
request.setAttribute("dataBean", dataBean);
RequestDispatcher rd = getServletContext().getRequestDispatcher("/outputjsp.jsp");
rd.forward(request, response);
```

This method performs the following actions:

- Instantiates an object named `dataBean`, which is based on the `DataBean` class from the `nservlets` package.
- Uses the `DataBean setValue1()` method to set the information to be passed to the JSP. If the `CallJSP` servlet is called with a parameter, it passes the parameter value in the `dataBean` object. Otherwise, it passes the default value of `Something Important`.
- Uses the `setAttribute()` method of the `HttpServletRequest` object to set the object to pass to the JSP.
- Creates a `RequestDispatcher` object named `rd` using the `getServletContext().getRequestDispatcher()` method setting the name of the JSP to call.
- Uses the `forward` method of the request dispatcher object to call the JSP passing the request and response objects.

Example 2-29 shows the source of the JSP 0.91 version of the `outputjsp.jsp` JavaServer Page.

**Example 2-29  Contents of the 0.91 version of `output.jsp`**

```xml
<BEAN name="dataBean" type="nservlets.DataBean" create="no" scope="request">
</BEAN>

<HTML>
<HEAD><TITLE>Simple JSP with Bean</TITLE></HEAD>
<BODY>
<CENTER>
<IMG SRC="/html/as400.gif" BORDER=2 X-SAS-UseImageWidth X-SAS-UseImageHeight HEIGHT=120 WIDTH=120/>
</CENTER>
<%
    out.println("The value of the parameter is "+dataBean.getValue1());
%
</CENTER>
</BODY>
</HTML>
```

This JSP performs these tasks:

- Defines a Bean named `dataBean` based on the `nservlets.DataBean` class. A parameter of `create=no` is used because the Bean is passed to the JSP by the servlet in the request object.
- Uses HTML tags to display a graphic.
- Uses a JSP scriplet to create Java code that uses the `DataBean getValue1()` method to retrieve and display the information passed in the `dataBean` object.
When running the JSP on WebSphere Application Server V4.0, it fails with a compiler syntax error because WebSphere V4.0 only implements the JSP 1.1 specification. You should migrate the JSP page to be compatible with the JSP processor of WebSphere Application Server. Example 2-30 shows the JSP page we use after modification.

Example 2-30  Contents of the 1.1 version of outputjsp.jsp

```
<jsp:useBEAN id="dataBean" class="nservlets.DataBean" scope="session" create="no" />

<HTML>
<HEAD><TITLE>Simple JSP with Bean</TITLE></HEAD>
<BODY>
<CENTER>
<IMG SRC="as400.gif" BORDER=2 X-SAS-UseImageWidth X-SAS-UseImageHeight HEIGHT=120 WIDTH=120></CENTER>

<% System.out.println("Inside outputjsp");
   nservlets.DataBean theDataBean = (nservlets.DataBean) request.getAttribute("dataBean");
   out.println("The value of the parameter is " + theDataBean.getValue1()); %>

</CENTER>
</BODY>
</HTML>
```

2.6.4 Running the CallJSP servlet inside VisualAge for Java

Similar to the process in 2.4.6, “Running the ItemServlet inside VisualAge for Java” on page 39, we need to update the default_app.webapp file in VisualAge for Java test environment with a text editor (like Untapped). This is shown in Example 2-31.

Example 2-31  The default_app.webapp file

```
<servlet>
 <name>CallJSP</name>
 <description>CallJSP</description>
 <code>nservlets.CallJSP</code>
 <servlet-path>/webapp/OrderEntry/CallJSP</servlet-path>
 <autostart>false</autostart>
</servlet>
```

Once we save the file, we copy the outputjsp.jsp file into the <VAJ install root>/ide/project_resources/IBM WebSphere Test Environment/hosts/default_host/default_app/web directory. The WebSphere Test Environment looks for the JSP file in this directory. This next part may be a bit more challenging. Since we use a image file in the JSP (as400.gif), we must also copy this file to the directory. However, because the context for this application is /webapp/OrderEntry, we cannot simply copy the image file to the
Web directory because the http server will not find it according to the context. We have to create a subdirectory under the Web directory named webapp. We then create a subdirectory under the webapp directory named OrderEntry. Now we copy the image file into the OrderEntry directory, as shown in Figure 2-22.

![Figure 2-22  Copying the image into WebSphere Testing Environment](image)

We restart the servlet engine in the WebSphere Test Environment and access the servlet by using the following URL:

http://localhost:8080/webapp/OrderEntry/CallJSP

The result is shown in Example 2-23.

![Figure 2-23  CallJSP result](image)

We could change the URL to:

http://localhost:8080/webapp/OrderEntry/CallJSP?value=Test

In this case, we see the result page in which the message becomes:

The value of the parameter is Test
2.6.5 Exporting the CallJSP servlet from VisualAge for Java

Similar to ItemServlet, before we deploy our new servlet to WebSphere Application Server V4.0, we have to export it into JAR file that can be used by the Application Assembly Tool to build a Web application. In the VisualAge for Java Workbench, select the access and nservlets packages (use the Ctrl key to select both) and export them to a JAR file. See 2.3.4, “Exporting class files to a JAR file” on page 28. We export to a file named wswjsp.jar.

We don't include the JSP file in the JAR file. Later, we use AAT to add it into the Web application.

2.7 Session management

A session is a series of requests and their associated responses originating from the same user, at the same browser. HTTP is a stateless protocol. This means each request that arrives carries only its own request context. The individual requests are not related. This causes problems when we want to develop stateful applications. For example, for an online shopping application, we may surf different Web pages and decide to buy items from several different pages. Without session tracking, we would have to pay for each item individually. This is not natural. We want to put all the goods into a shopping cart and pay for them one time.

Sessions can be used to solve this problem. We can use a session object to store and maintain a Web “shopping cart”. A shopping cart is a good example of using session tracking.

2.7.1 Session tracking solutions

There are several ways to maintain a state across a series of requests. For all of them, the basic theory is simple, that is to send a token back and forth for the identity, as shown in Figure 2-24.

![Figure 2-24  Session tracking](image)

Now the question is where to put the token in the HTTP data stream. According to the HTTP frame shown as Figure 2-25, we can put this token either in the HTTP body, header, or Start Line (URL).
There are three common approaches for session tracking:

- **Hidden fields**
- **Cookies**
- **URL rewriting**

### Hidden fields

Hidden HTML fields are not displayed by the browser. They are displayed when you choose the option to view the HTML source. Hidden fields are declared with the attribute named `type` set to `hidden` as shown in the following code fragment:

```
<input type="hidden" name="partno" value=" " size=10 maxlength=10>
```

A servlet can generate HTML with hidden input fields. When the browser submits the HTML page, the information in the hidden HTML fields is available to the program that handles the HTTP request.

The main advantage of using hidden fields for session tracking is their simplicity. They do not require any user authentication, and they are supported on most, if not all, of the browsers.

Their main disadvantage is that, by their nature, they will work only for a sequence of dynamically generated HTML.

As you can see from the previous code fragment, there is an overhead defining a hidden field for each value you want to persist across a series of requests and the type of information that each such input field can hold is limited to textual information.
In addition, hidden fields are transmitted between the client and the server, when you really need them only at the server side. A solution to these deficiencies can be to use just one hidden input field that holds the session ID, and use it as a key to access the session state information from a database file located on the server.

**Cookies**

Cookie technology was developed by the Netscape Communications Corporation. This technology enables a Web server to store and retrieve information from the client. When the cookie information is sent to the server, it is put in the HTTP header.

A Web server generates a cookie using the `Set-Cookie` header as part of the HTTP response. The cookie holds a single-name value pair and a few attributes that describe the cookie properties, such as its version and the range of URLs for which it is valid. Upon receiving a cookie, the browser stores it and includes it in any future HTTP requests made to the originating server, according to rules that match the cookie's attributes and the requested URL.

The convenience class `javax.servlet.http.Cookie` represents a cookie and makes it easier to handle cookies within a servlet.

You create a cookie with the constructor:

```java
javax.servlet.http.Cookie(name, value)
```

Then, you add it to the HTTP response with the method:

```java
javax.servlet.http.HttpServletResponse.addCookie(cookie)
```

The `addCookie` method can be called multiple times to set more than one cookie. Browsers are expected to support only 20 cookies per host, of at least 4 KB each, so it is not a good practice to use many cookies.

The `Cookie` class provides getter and setter methods to retrieve and set the cookie value and attributes. As with HTML form hidden fields, even though the information stored in the cookie is needed on the server, it is stored on the client. This requires that it be sent to the server with each request. This is not a big problem, because the cookie size and number is limited. But, you may consider storing the session ID in a cookie and keeping the session state data on the server.

Since cookies are part of the HTML header, as for all HTML header fields, you need to add all the cookies to the response object before getting the PrintWriter object.

Cookies provide a nice way to implement sessions, but they suffer from some problems, for example, some users, either by choice or mandate, cannot receive cookies from Web sites. Also most browsers allow users to turn off the ability to receive cookies. Under this situation, an alternative technique, known as **URL encoding**, is available for passing user session IDs between the user and the Web application server instance.

**URL rewriting**

URL rewriting is a technique that embeds client-specific tokens within the URL of each request. For example, we can access our callJSP servlet by the URL:

```text
http://sysname:port/OrderEntry/callJSP
```

We can also pass a specific value to the servlet by changing the URL to:

```text
http://sysname:port/OrderEntry/callJSP?value=hello
```
This request sends the parameter value to the Web container; the container can obtain the value from this URL and do further processing. This kind of approach is called URL rewriting since it changes the URL to include client specific tokens. If we are working on an online shopping application, we do not want to put confidential information on the URL. In the Sun Servlet API specification, a parameter named jsessionid is available. It sends the session identifier back and forth, while the actual session information is stored on the server.

URL rewriting requires the developer to use special encoding APIs and to set up the site page flow to avoid losing the encoded information. For example, if a servlet returns HTML directly to the requester (without using a JSP page), the servlet calls the following API to encode the returning content:

```java
out.println("<a href="");
out.println(response.encodeURL("/OrderEntry/ItemServlet"));
out.println("">Item Servlet</a>"/ajax);
```

URL rewriting limits the flow of site pages exclusively to dynamically generated pages (such as pages generated by servlets and JSPs). WebSphere inserts the session ID into dynamic pages, but cannot insert it into static pages (.htm or .html pages).

### 2.7.2 HttpSession interface

The `javax.servlet.http.HttpSession` interface is implemented by WebSphere Application Server to provide session support. Under the covers, HttpSession information is maintained either by using cookies or by URL rewriting as described previously. The WebSphere Administrative Console is used to enable session support using URL rewriting or cookies.

For example, in WebSphere Application Server Version 4.0 Advanced Edition, you can select the server from the left pane of the console and click the Services tab in the right pane as shown in Figure 2-26. There are some session services you can edit. Select Session Manager Service and click Edit Properties.

![Figure 2-26 Editing Session Manager properties](image-url)
The Session Manager Service page displays. Under the General tab, you can enable the cookies support or URL rewriting support. It is possible to select both cookie support and URL rewriting support at the same time. In this case, cookies are used and preferred over URL rewriting.

**Major changes in the servlet 2.2 API**

In the servlet 2.2 API, session scoping is per Web application. Servlets in different Web applications running in the same servlet engine cannot share session information. In the servlet 2.1 API, session scoping is per servlet engine.

**HTTPSession deprecation includes:**

- `getValue(String)` replaced by `getAttribute(String)`
- `getValueNames()` replaced by `getAttributeNames()`
- `putValue(String, Object)` replaced by `setAttribute(String, Object)`
- `removeValue(String)` replaced by `removeAttribute(String)`

**Creating sessions**

You create a session by using the following method:

```java
class javax.servlet.http.HttpServletRequest
getSession(create)
```

If the value of the boolean parameter `create` is false, the method returns the current valid session associated with the request, if one exists. Otherwise, it returns null. If the value of `create` is true and a session does not exist, it is created.

Creating a session means that the server generates a unique session ID and sends it to the client browser as a cookie or by means of URL rewriting. The browser is expected to return this session ID on all subsequent requests made to this server from the same browser. This identifies the request as part of the session.

**Using the HttpSession object**

Once a session is created, values can be saved in the session object or retrieved from the session object, using either of the following two methods:

```java
class javax.servlet.http.HttpSession
setAttribute(name, value)
getAttribute(name)
```

The `setAttribute()` method accepts a String for the name of the value and a Java object for the value. The `getAttribute()` method accepts a String for the name and it returns an object. Multiple values can be written to, or read from, the session. The `getAttributeNames()` method can be used to retrieve an Enumeration of names associated with a session. This Enumeration can be used to retrieve the objects themselves.

Objects can also be removed from the Session object by using:

```java
removeAttribute(name)
```

If there is a requirement that the sessions persist across server shutdowns, or they are used in a session cluster environment (where one WebSphere Application Server plays the role of a sessions server), the Java objects that you save to a session should implement the `Serializable` interface.
Invalidating sessions
Sessions exist until they are invalidated. Invalidating sessions is done automatically by WebSphere Application Server, if the session was not used for more than the Invalidate Time value, which defaults to 30 minutes. Sessions can also be manually invalidated by a call to the invalidate() method:
javax.servlet.http.HttpSession.invalidate()

2.7.3 ItemSessionServlet example
To demonstrate the use of sessions, we develop a shopping cart type application that uses session objects to hold an instance of a shopping cart and a vector of items queried from the database. Figure 2-27 shows the design of the application.

For this application, we use a Java ShoppingCart class that represents, as its name implies, a shopping cart. A ShoppingCart object holds the items that we choose to buy. This class is fully described later. For this context, it is enough to know about three methods that are implemented by this class:

- The ShoppingCart() constructor is used to create an instance of a ShoppingCart object.
- The setItems(items) method is used to set the content of the shopping cart.
- The getItems() method returns a Vector object that represents the content of the shopping cart.

![Servlet with Session Object](image)

Figure 2-27  ItemSessionServlet diagram

Figure 2-27 shows the flow of the application. The application starts when the user is presented with the Query Screen HTML page. When the user clicks the Search button, the following sequence of jobs is performed:
1. A session object is created.
2. The Item file is queried according to the user selection (either for a specific item or all the items).
3. The vector that contains the items is placed in the Session object.
4. The Item list HTML, shown in Figure 2-28, is generated and sent to the user browser.

![Figure 2-28 Item list HTML page](image)

From the Item List HTML page, the user can choose any one of the following steps:

- Click the **Search Again** button to return to the Query Screen HTML page.
- Click the **Add to Cart** button, which triggers the following sequence of execution.
  a. The CartServlet is invoked.
  b. The vector of items is retrieved from the session.
  c. The shopping cart object is retrieved from the session. If one does not exist, a newly created shopping cart is added to the session.
  d. The items that are selected by the user in the HTML page are added to the shopping cart.
  e. The Shopping Cart HTML page is generated, as displayed in Figure 2-29, and sent to the user.
- Clicking **Show Cart** button, which triggers the same sequence of execution as in the previous option, except for adding the selected item to the cart.
- From the Shopping Cart HTML page, the user only has the option to click the Continue button that invokes the ItemSessionServlet again.
The ItemSessionServlet class is based on the ItemPoolServlet class. It is modified to use a Session object. Only those modifications are described.

As shown in Example 2-32, the class imports the ShoppingCart class described previously. It implements a shopping cart.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import shopData.ShoppingCart.*;

public class ItemSessionServlet extends SuperServlet {
    private access.JDBCPoolCatalog aJDBCPoolCatalog;
}
```

**ItemSessionServlet doPost() method**

The ItemSessionServlet.doPost() method processes the HTML POST requests to this servlet. The request can come from two different sources, including:

- From the Query Screen HTML page, in which case we want to retrieve the items data from the database, and save them to the Session object.
- From the Shopping Cart HTML page after the user clicks the Continue button, in which case we get the Vector of items from the Session object.
This logic is implemented with the if statements shown in Example 2-33.

Example 2-33  ItemSessionServlet parameter processing

```java
if (parameter.toUpperCase().equals("CONTINUE")) {
    flexLog("Continue shopping");
    parts = (Vector) session.getAttribute("ItemSessionServlet.parts");
} else {
    // retrieve all data from the database
    if (parameter.toUpperCase().equals("*ALL")) {
        parts = aJDBCPoolCatalog.getAllV();
    } else {
        parts = aJDBCPoolCatalog.getRecord(parameter);
    }
```

Why can we use a single parameter to check whether you click the Continue button or you input the part No. in the HTML page? That’s because we give the Continue button a name “partno”, the same as the input field in HTML page.

Note how a Session object is created, as shown in Example 2-34, using the `getSession()` method with the create parameter set to `true`. This is done because we want to create a session if one does not exist (if this is the first time we are on the Item List HTML page).

Example 2-34  ItemSessionServlet doPost method

```java
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException {
    flexLog("do Post in ItemSessionServlet");
    boolean create = true;
    // Get the Session object
    HttpSession session = request.getSession(create);
    // set the MIME type to text/html
    response.setContentType("text/html");
    // create the output stream
    PrintWriter out = response.getWriter();
    try {
        // get the value from the input field named partno (see HTML file)
        // and check if value is valid
        //Only get the first parameter
        String parameter = null;
        if ((request.getParameter("partno")) == null) // partno not passed into servlet=>
            // set to *ALL
        {
            parameter = "*ALL";
        } else {
            parameter = request.getParameter("partno");
        }
        Vector parts = null;
        flexLog("ItemSessionServlet: parameter passed in from HTML is: " + parameter);
        // write the HTML header to the output stream
        outputHeader(out, getServletInfo());
        // retrieve all data from the session info if called from shopping cart
        // parts is a vector which each element is a string array length 4 with
```
// item id, item name, price, and item information each of the string elements.

// If it isn't a CONTINUE, you need to go to the database and retrieve.

if (parameter.toUpperCase().equals("CONTINUE")) {
    flexLog("Continue shopping");
    parts = (Vector) session.getAttribute("ItemSessionServlet.parts");
} else {
    // retrieve all data from the database
    if (parameter.toUpperCase().equals("*ALL")) {
        parts = aJDBCPoolCatalog.getAllV();
    } else {
        parts = aJDBCPoolCatalog.getRecord(parameter);
    }

    // save the vector to the Session Data
    flexLog("Putting the parts list in the session object; number of parts = ":
        parts.size());
    session.setAttribute("ItemSessionServlet.parts", parts);

    flexLog("Now the table goes up");
    // write the HTML table to the output stream
    outputItemInformation(out, parts);
    out.close();
} catch (Throwable e) {...
} // end of doPost()

ItemSessionServlet.outputItemInformation() method
The ItemSessionServlet.outputItemInformation() method generates the Item List HTML page. As shown in Figure 2-28, two different servlets can be invoked from this generated HTML file. This is why two HTML FORM tags are generated. The form that invokes the CartServlet contains two buttons, one to add items to the cart, and the second just to show the cart. The two buttons share the same name "command," but they have different values that are checked in the CartServlet.

We add a Select check box to each item line to enable the user to choose the items to purchase, shown as Example 2-35.

Example 2-35   ItemSessionServlet outputItemInformation() method

private void outputItemInformation(PrintWriter out, Vector partsVector) throws IOException {
    flexLog("ItemSessionServlet: outputItemInformation()...");
    Enumeration parts = partsVector.elements();

    // Add to cart button will invoke the cart servlet.
    out.println("<FORM METHOD=POST ACTION="/webapp/OrderEntry/CartServlet" >");
    out.println("<CENTER>");
    out.println("<TABLE BORDER>");
    out.println("<P><CENTER><B><FONT SIZE=+3>Here are the results of your query:"</FONT></B></CENTER></P>");
    out.println("<TR>");
    out.println("<TH>Select</TH><TH>ITEM ID</TH><TH>ItemName</TH><TH>Price</TH><TH>Quantity</TH>");
    out.println("</TR>");
    int i = 0;

while (parts.hasMoreElements()) {
    String[] aPart = ((String[]) parts.nextElement());
    out.print("<TD><CENTER><INPUT TYPE = checkbox name = index value = " + i + ">
    out.println("</TR>" + i);
    i++;
}; // end while
out.println("</TABLE*>");
out.println("</CENTER>");
out.println("<P><CENTER><INPUT TYPE=submit value="Add to Cart" name="command">< INPUT TYPE=submit value="Show Cart" name="command"></CENTER>");
out.println("</FORM>");

// Continue Shopping button - just go back to main HTML page
out.println("<FORM action="/itemSessionservlet.html" method="GET">")
out.println("<P><INPUT TYPE=submit value="Search Again" name="command"></CENTER>");
out.println("</FORM>");
out.println("</FONT></BODY></HTML>");
flexLog("ItemSessionServlet: outputItemInformation() executed.");
} // end outputPartsInformation()

CartServlet class
The CartServlet class is responsible for adding the selected items (stored in the session) to
the cart and displaying the cart content by generating an HTML page.

Most of the work is done by the doPost() method, which is described in the following section.

CartServlet.doPost() method
The cartServlet.doPost() method, shown in Example 2-36, starts by getting the Session
object associated with the HTTP request. In this case, the value of the create parameter of
the getSession() method is set to false, because the session should already have been
created by ItemSessionServlet. If the session does not exist, it is an indication of some kind of
problem.

Next, the method tries to retrieve the vector that contains the items from the session. Since all
the values saved in the session object are shared among all servlets in the session, it is
common practice to prefix the session value name with the servlet name to avoid naming
collisions. This is why the vector that contains the items name is prefixed with
ItemSessionServlet.

Example 2-36   CartServlet doPost() method
public void doPost(HttpServletRequest request, HttpServletResponse response) throws
IOException {
    flexLog("CartServlet: doPost");
    // Session should already be available
    boolean create = false;
    // Get the Session object
    HttpSession session = request.getSession(create);
// set the MIME type to text/html
response.setContentType("text/html");

// create the output stream
PrintWriter out = response.getWriter();
try {
   // Get the list of the parts queried from the database from the session object.
   flexLog("Getting the items from the session");
   Vector parts = (Vector) session.getAttribute("ItemSessionServlet.parts");
   if (parts == null) {
      flexLog("parts is null!");
   } else {
      flexLog("Parts retrieved successfully from session, it has "+ parts.size() + "
   }

   //Get the command parameter to determine if this is an add or a show cart
   String parameter = request.getParameter("command");
   flexLog("CartServlet: parameter passed in from ItemSessionServlet is: " +
parameter);
   String[] value = request.getParameterValues("index");
   ShoppingCart cart = (ShoppingCart) session.getAttribute("shopcart.selected");
   if (cart == null) {
      cart = new ShoppingCart();
      flexLog("cart did not exist in session at this time");
   } else {
      flexLog("cart retrieved from session successfully, it has " +
   cart.getItems().size() + " elements");
   }

   // If "Add to Cart" was selected and items are checked, add the items to the cart
   if ((parameter.equalsIgnoreCase("Add to Cart")) && (value != null)) {
      int j = 0;
      for (int i = 0; i < value.length; i++) {
         j = Integer.parseInt(value[i]);
         flexLog("i: "+i + " value: "+j);
         String[] data = (String[]) parts.elementAt(j);
         CartItem aCartItem = new CartItem(data[0], data[1], data[2], data[3], new
Integer(1));
         cart.getItems().addElement(aCartItem);
      }
   }

   // save the vector to the Session Data
   session.setAttribute("shopcart.selected", cart);
}

// write the HTML header to the output stream
outputHeader(out, getServletInfo());

// write the HTML table to the output stream
outputItemInformation(out, cart);
out.close();
} catch (Throwable e) {
   printError(out, e);
}
} // end of doPost()
This servlet can either add the items to the cart and display the cart, or just display the cart. This functionality is determined according to the value of the command parameter. If the command parameter value is *Add to cart*, after adding the items to the cart, the cart is saved again to the session object.

### 2.7.4 Running the ItemSessionServlet servlet inside VisualAge for Java

Similar to the servlet testing previously in VisualAge for Java, we need to update the `default_app.webapp` file in the VisualAge for Java test environment with a text editor (like Notepad). This is shown in Example 2-37.

#### Example 2-37  The default_app.webapp file

```xml
<servlet>
  <name>ItemSessionServlet</name>
  <description>ItemSessionServlet</description>
  <code>nservlets.ItemSessionServlet</code>
  <servlet-path>/webapp/OrderEntry/ItemSessionServlet</servlet-path>
  <init-parameter>
    <name>user</name>
    <value>Team43</value>
  </init-parameter>
  <init-parameter>
    <name>password</name>
    <value>win4ibm</value>
  </init-parameter>
  <init-parameter>
    <name>datasource</name>
    <value>jdbc/NativeDS</value>
  </init-parameter>
  <autostart>false</autostart>
</servlet>

<servlet>
  <name>CartServlet</name>
  <description>CartServlet</description>
  <code>nservlets.CartServlet</code>
  <servlet-path>/webapp/OrderEntry/CartServlet</servlet-path>
  <autostart>false</autostart>
</servlet>
```

Once we save the property file, we copy the HTML page, itemsessionservlet.html file, into the `<VAJ install root>\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web` directory. It is used to accept input as shown as Example 2-38. We restart the servlet engine of the WebSphere Test Environment.

#### Example 2-38  ItemSessionServlet.html page

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0//EN"
     "http://www.w3.org/TR/REC-html40/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    <meta name="GENERATOR" content="Mozilla/4.04 [en] (WinNT; I) [Netscape]">
    <title>Items Retrieval</title>
  </head>
  <body>
    <!--This file created 3:28 PM  2/11/98 by Claris Home Page version 2.0-->
    <x-sas-window top=79 bottom=699 left=12 right=845>
    </x-sas-window>
  </body>
</html>
```
<BODY BGCOLOR="#C0C0C0">
<FORM method="POST" action="/webapp/OrderEntry/ItemSessionServlet">
<CENTER><IMG SRC="as400.gif" BORDER=2 X-SAS-UseImageWidth X-SAS-UseImageHeight HEIGHT=120 WIDTH=120></CENTER>
<CENTER><HR SIZE=5></CENTER>
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>Enter *ALL to get all items from the catalog</FONT></FONT></B></CENTER>
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>or</FONT></FONT></B></CENTER>
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>Enter the item number to get only one item from the catalog</FONT></FONT></B></CENTER>
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>Press the Button to retrieve the items</FONT></FONT></B></CENTER>
<CENTER>&nbsp;</CENTER>
<CENTER><TABLE BORDER=0 WIDTH="50%" HEIGHT="1" >
<TR>
<TD WIDTH="169" HEIGHT="14">
<DIV ALIGN=right><FONT FACE="Arial,Helvetica">Item Number or *ALL</FONT>&nbsp;</DIV>
</TD>
<TD WIDTH="118"></TD>
</TR>
<TR>
<TD WIDTH="169"></TD>
<TD WIDTH="118">INPUT TYPE="text" NAME="partno" VALUE="*ALL" SIZE=10 MAXLENGTH=10--><TD>
</TR>
</TABLE></CENTER>
<CENTER><HR SIZE=5 WIDTH="100%"></CENTER>
</FORM>
This file uses the servlet <I>ItemServlet</I> to retrieve information from the AS/400 system.

Now we access the ItemSessionServlet application by entering the following URL:

http://localhost:8080/ItemSessionServlet.html

The input HTML page appears as shown in Figure 2-30.
We click the **Get Items Information** button. It shows all the items that we can select, as shown in Figure 2-31. We select several items and click the **Add to Cart** button.

The next page (Figure 2-32) shows the items we selected in a shopping cart. You can click **Continue** to select more items.
2.7.5 Exporting the ItemSessionServlet from VisualAge for Java

Similar to ItemServlet, before we deploy our new servlet to WebSphere Application Server V4.0, we have to export it into JAR file that can be used by the Application Assembly Tool to build a Web application.

In the VisualAge for Java Workbench, select the access, nservlets, and shopData packages (use the Ctrl key to select all three) and export them to a JAR file. See 2.3.4, “Exporting class files to a JAR file” on page 28. We export to a file named wswssession.jar.
WebSphere V4.0 assembly and deployment tools

In Chapter 2, “Servlet and JSP development using VisualAge for Java” on page 21, we developed several servlet and JSP applications and tested them inside VisualAge for Java V4.0. We exported the servlet examples from VisualAge for Java into JAR files:

- wsws.jar: MyHelloWorldServlet
- wswsitem.jar: ItemServlet
- wswspool.jar: ItemPoolServlet
- wswsjsp.jar: CallJSP
- wswssession.jar: ItemSessionServlet

Tools that are not J2EE compliant, like VisualAge for Java 4.0, cannot package applications into the proper format for installation under WebSphere Application Server 4.0. It is necessary to use a tool, like the Application Assembly Tool (AAT), to package them into J2EE format before they can be installed. This chapter explains how to use AAT to do this.
3.1 WebSphere 4.0 application packaging overview

WebSphere Application Server Version 4.0 is fully J2EE 1.2 compliant. To deploy an application on WebSphere Application Server 4.0, we cannot simply copy the servlet or JSP files into the server directory. J2EE requires that all your code, whether it is EJBs, JSPs, servlets, image files, JAR files, and so on, be packaged into modules with deployment descriptors provided for each module.

There are three types of modules in J2EE:

- **Web module**: This is a deployable archive file that includes Java servlets, JSP pages, JAR files, HTML files, images, and so on. We call this file a Web Archive file or WAR file. The extension of the file is .war. There is a deployment descriptor file, `web.xml` in the WEB-INF directory of the archive that contains the deployment description for the WAR file.

- **EJB module**: This is a deployable archive file that includes EJBs and the associated JAR files. The extension of the file is .jar. The deployment descriptor, `ejb-jar.xml` is also included in the META-INF directory of the archive.

- **Client module**: This is a JAR file that contains Java client classes and a deployment descriptor named `application-client.xml`.

A J2EE enterprise application consists of one or more of these three modules packaged into an Enterprise Archive (EAR) file. The EAR file also contains an XML file named `application.xml` that describes the application. The extension of the EAR file is .ear.

For enterprise applications running on WebSphere Application Server Version 4.0, we also package some WebSphere specific extensions into the EAR file, which are not included in the J2EE specification. Examples include transaction isolation attributes, Web application reloading, file serving, and servlet invoker by classname information. These extensions generate additional deployment descriptors, like `ejb-jar.xml`, `ibm-ejb-jar-ext.xmi`, and `ibm-ejb-jar-bnd.xmi`.

The complete picture of WebSphere 4.0-based application packaging is shown in Figure 3-1. To deploy applications developed with VisualAge for Java, WebSphere Studio, or other non-J2EE compliant tools, we need to package them into Web modules, EJB modules, client modules, and eventually enterprise archive files. IBM provides a tool to help with this process. It is called the *IBM Application Assembly Tool* and is available in WebSphere Version 4.0.
3.2 Application Assembly Tool overview

Application Assembly Tool is a graphical user interface (GUI) tool provided with WebSphere V4.0 to perform application assembly for WebSphere applications. The process for AAT consists of two steps:

1. Create archive files that contain all of the files that belong to a specific application. The archive files can be WAR files, EJB modules, and client modules.

2. Configure the runtime behavior of the application. This generates the XML-based deployment descriptors that list the contents and characteristics of the modules and contain instructions for how the modules are deployed in the runtime environment.

AAT has rich functions. Some of the following features are very useful for building applications that will be deployed on WebSphere 4.0:

- Create/Edit J2EE applications (EARs) from J2EE modules.
- Create/Edit J2EE modules.
  - Web modules (.war files) for servlets, JSPs, HTML
  - EJB modules (.jar files) for EJB and associated JAR files
  - Client modules (.jar files) for Java application client classes
- Modify the deployment descriptor information.
- Modify the binding information attributes.
- Modify the IBM extension attributes.
- Convert EJB JAR files from the EJB 1.0 specification to the EJB 1.1 specification.

AAT is shipped with the WebSphere 4.0 client support. Because it does not run on the iSeries server, you must install it on a workstation.

To start AAT, open a command prompt session and change directories to the <WebSphere install>\AppServer\bin directory, enter `assembly` to start the AAT.
The AAT welcome page displays as shown in Figure 3-2. You can click one of the icons to directly go to the specific function, or click Cancel to use the main menu.

![Application Assembly Tool welcome page](image)

**Figure 3-2  Application Assembly Tool welcome page**

From the main menu, you can either use the menu items or simply click the smarticons, as shown in Figure 3-3.

![Application Assembly Tool smarticons](image)

**Figure 3-3  Application Assembly Tool smarticons**

Some icons are greyed out. This means these icons take effect only if you invoke some specific functions.

### 3.3 Application packaging and deploying scenario

In this section, we package the JAR files created with VisualAge for Java using AAT. The JAR files are:
► *wsws.jar*: Contains MyHelloWorldServlet
► *wswsitem.jar*: Contains ItemServlet
► *wswspool.jar*: Contains ItemPoolServlet
► *wswsjsp.jar*: Contains CallJSP servlet

We package the JAR files into Web applications and deploy the Web applications on WebSphere Application Server. We test the deployed applications using the internal WebSphere HTTP server.

We also build an enterprise application based on the EJB code developed in Chapter 8, “iSeries EJB application development scenario” on page 245. We go through the steps of building individual EJB modules, Web modules, and client modules, and then generate the EAR file. We deploy it on WebSphere Application Server Version 4.0 Advanced Edition and test it through the internal WebSphere HTTP server.

### 3.3.1 Packaging MyHelloWorldServlet

Here are the steps to deploy the MyHelloWorldServlet example:

1. Start the Application Assembly Tool.
2. Once the welcome page displays, select **Web Module** and click **OK**.
3. In the Application Assembly window, right-click **Web Components** and select **New** (Figure 3-4).

![Figure 3-4 Creating a new Web component](image)

4. In the New Web Component window (Figure 3-5), enter **MyHelloWorld** as the Component name. Select the **Servlet** radio button and click the **Browse** button to add the servlet name.
5. In the Select file for Class name window, click **Browse** to locate the JAR file (c:\wsws\wsws.jar) that you exported from VisualAge for Java. In our example, we placed the JAR files in the wsws directory.

6. After you select the JAR file, you return to the Select file for Class name window (Figure 3-6). Select the **nservlets** package in the left pane and then select the **MyHelloWorldServlet.class** file in the right pane (make sure you select the class file). Click **OK**.
This takes you back to New Web Component window. The window now contains a component name and a class name as shown in Figure 3-7.
7. Click **OK** to save the settings. Now you see MyHelloWorld listed as a Web Component, as shown in Figure 3-8.

![Application Assembly Tool](image)

**Figure 3-8  AAT with a Web Component**

8. Create a servlet mapping that is similar to an alias in WebSphere 3.5 for this servlet. Right-click **Servlet Mapping** in the left panel and select **New**. The New Servlet Mapping window displays, as shown in Figure 3-9.
9. Enter MyHello for the URL pattern and select MyHelloWorld as the servlet. Click OK to save the settings.

10. The configuration of MyHelloWorldServlet is complete. It is time to deploy the Web application. Select File-> Save As. We save our work in the wsws directory as wsws.war.

Next we use the WebSphere Application Server Version 4.0 Advanced Single Server Edition Administrative Console to install the Web module so it can be run under WebSphere Application Server Single Server.

### 3.3.2 Installing the MyHello Web module under the Single Server

To begin, start the Web-based console. Then follow these steps:

1. Click the plus (+) sign in front of Nodes to reveal the topology. You see the iSeries server name for Node name. Drill down to the next level and click **Enterprise Applications**. All the installed applications are displayed, as shown in Figure 3-10.
2. Install the Web module from this page. Click the **Install** button to start the process. When the Application Installation Wizard page displays, use the **Browse** button to locate the `wsws.war` file in the `wsws` directory. As shown in Figure 3-11, enter `MyHelloWorld` for both **Application Name** and **Context Root** and click **Next**.

Figure 3-10  Enterprise applications in WebSphere 4.0 Single Server Edition

Figure 3-11  Installing a Web Module under WebSphere 4.0 Single Server Edition
3. Click **Next** again when you see the Specifying Virtual Host names and Precompiled JSP option for the Web Modules page. When you see the confirmation dialog, click the **Finish** button.

4. Click the **Save** button on the top of this page to save the configuration. Select the **Save Configuration** radio button and click **OK**, as shown in Figure 3-12.

```
Figure 3-12   Saving the configuration for WebSphere 4.0 Single Server Edition
```

5. Stop and restart the application server to make the application available. Select **Application Servers** in the left pane and click **Stop** in the right pane, as shown in Figure 3-13.

```
Figure 3-13   Stopping the application server
```

6. You may see the warning dialog, shown in Figure 3-14. Click **OK** and wait for the application to stop. You will lose the session in your browser.
7. Start WebSphere Application Server. Open an iSeries 5250 session and ensure that your application server has ended. You can use the WRKACTJOB command to check the QEJBAES4 subsystem. Then follow these steps:
   a. Enter qsh to start a Qshell session.
   b. Enter the following command to change directories:
      ```
      cd /qibm/proddata/webasaes4/bin
      ```
   c. Enter the following command to start the WebSphere instance (where yourinstance is the name the WebSphere instance):
      ```
      strwasinst -instance yourinstance
      ```

8. Once the application server starts, test the new Web application. Open a browser session and enter the URL as:

   ```
   http://sysname:port/MyHelloWorld/MyHello
   ```

   Replace sysname with your server name and port with the internal WebSphere HTTP server port. Figure 3-15 shows the results.
9. If you want to use an external HTTP server to test this application, you need to re-generate the Web server plug-in configuration.

From the Administrative Console, expand **Nodes** and drill down to **Application Servers**. Expand **Application Servers** and select the instance application server in the left pane. In the properties page on the right, click **Web Server Plug-in Configuration**, as shown in Figure 3-16.

10. Click **Generate** on the Web Server Plug-in Configuration page. It re-generates the plug-in configuration for the external HTTP server.
Now we can access our application by the external HTTP port:

http://sysname:port/MyHelloWorld/MyHello

Here *sysname* is the iSeries server name and *port* is the external HTTP server port.

**Tip:** If the HTTP server fails to find the servlet, you may need to restart the external HTTP server.

### 3.3.3 Packaging ItemServlet

Next, we package ItemServlet into a Web module with Application Assembly Tool. This is very similar to the steps in 3.3.1, “Packaging MyHelloWorldServlet” on page 79. We focus on the differences here.

1. Start Application Assembly Tool.
2. Click **Cancel** on the AAT welcome page.
3. From the main menu, select **File-> Open** to open the wsws.war file in the wsws directory. This WAR file was created in 3.3.1, “Packaging MyHelloWorldServlet” on page 79.
4. Right-click **Web Components** and select **New** to add a new component.
5. Enter *ItemServlet* in the input field for Component Name and locate the ItemServlet.class from the wswsitem.jar file for the class name. The result should look like the example in Figure 3-17.

![New Web Component](image)

*Figure 3-17 Adding a Web Component for ItemServlet*

6. Click **OK** to save the settings.
7. Create the Servlet Mapping by right-clicking the Web component and selecting **New**. We enter *ItemServlet* for the URL pattern and select *ItemServlet* for the servlet.
Once we finish this, we see the difference from the MyHelloWorld application. ItemServlet uses initialization parameters for userid, password, JDBC url and JDBC driver.

8. Now we configure the parameters through AAT. Select **Web Components->ItemServlet**.
   a. Right click **Initialization Parameters** and select **New**.
   b. As shown in Figure 3-18, enter *driver* as the Parameter Name and
      `com.ibm.db2.jdbc.app.DB2Driver` as the Parameter Value. Click **Apply**.

   ![New Initialization Parameter](image)

   **Figure 3-18** Setting the initialization parameters

   Use the same technique to enter the following parameter values:
   - **User**: Your user ID
   - **Password**: Your password
   - **URL**: *jdbc:<db2://sysname/library* where *sysname* is the iSeries server name and *library* is the library name for the ITEM table.

   A total of four parameters are configured as shown in Figure 3-19.
9. ItemServlet uses some classes from the access package. The next step to add these supporting classes. Select File-> Class Files-> Add Files.

10. In the Add Files window, click Browse to locate the wswsitem.jar file in the wsws directory.

11. Select the access package in the left pane and then select all the classes in the right pane. To do this, simply select the first class and then, holding down the Shift key, click to select the last class. All classes are then selected, as shown in Figure 3-20.

12. Click Add to add the files.
13. Using the same technique, add the SuperServlet class to the supporting classes. To do this, click the `<nservlets>` package in the left pane, click the `SuperServlet.class` file in the right pane, and click `Add`.

14. Click `OK` to save the settings and return to the AAT main page.

15. Now the ItemServlet is complete. We can export it to a WAR file. Select `File-> Save As` and save the work in the wsws directory as wsws.war. Click `Yes` to the confirmation dialog and exit AAT.

### 3.3.4 Installing the ItemServlet Web module on the Single Server

We have done most of the work with AAT. The installation on WebSphere Application Server is relatively simple and similar to the steps in 3.3.2, “Installing the MyHello Web module under the Single Server” on page 83.

1. Open the WebSphere Administrative Console through the browser. Click the plus (+) sign in front of `Nodes`. Drill down to `Enterprise Applications` and click it. Then click the `Install` button on the right pane.

2. In the Application Installation Wizard page, use the Browse button to locate the wsws.war file for the Path. Enter `OrderEntry` for both Application Name Context Root (it’s case sensitive), as shown in Figure 3-21. Click the `Next` button.
3. Click the **Next** button again on the Specifying Virtual Host names and precompiled JSP option for Web Modules dialog.

4. Click **Finish** on the confirmation dialog.

5. Don’t forget to click the **Save** button on top of the page to save the settings.

6. As in 3.3.2, “Installing the MyHello Web module under the Single Server” on page 83, you should stop and restart the application server to make the application available.

7. After the server is restarted, access the application by entering the following URL:

   http://sysname:port/OrderEntry/ItemServlet

   Here *sysname* is the iSeries server name and *port* is the internal HTTP server. The results are shown in Figure 3-22.
If you want to use the external HTTP server, follow steps in 3.3.2, “Installing the MyHello Web module under the Single Server” on page 83.

### 3.3.5 Running ItemServlet from an HTML file

So far, we have accessed the servlet directly from the browser. In most cases, the interface we deal with is an HTML page or JSP page. In this section, we use an HTML page for data input. When you click the Submit button, the data is sent to the ItemServlet, and the servlet accesses and displays the database information.

The HTML page is shown in Example 3-1.

*Example 3-1  ItemServlet.html page*

```
<HTML>
<HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
    <META NAME="GENERATOR" CONTENT="Mozilla/4.04 [en] (WinNT; I) [Netscape]">
    <TITLE>Items Retrieval</TITLE>
</HEAD>
<BODY BGCOLOR="#C0C0C0">
  <FORM method="POST" action="/OrderEntry/ItemServlet">
    <CENTER><IMG SRC="as400.gif" BORDER=2 X-SAS-UseImageWidth X-SAS-UseImageHeight HEIGHT=120 WIDTH=120></CENTER>
    <CENTER><HR SIZE="5"></CENTER>
    <CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>Enter *ALL to get all items from the catalog</FONT></B></FONT></CENTER>
</FORM>
</BODY>
</HTML>
```
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>or</FONT></FONT></B></CENTER>
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>Enter the item number to get only one item from the catalog</FONT></FONT></B></CENTER>
<CENTER><B><FONT COLOR="#0000AF"><FONT SIZE=+2>Press the Button to retrieve the items</FONT></FONT></B></CENTER>
<CENTER>&nbsp;</CENTER>
<TABLE BORDER=0 WIDTH="50%" HEIGHT="1">
  <TR>
    <TD WIDTH="169" HEIGHT="14">
      <DIV ALIGN=right><FONT FACE="Arial,Helvetica">Item Number or *ALL</FONT>&nbsp;</DIV>
    </TD>
    <TD WIDTH="118" HEIGHT="14"><!-- Add the input field after this line -->
      <INPUT TYPE="text" NAME="partno" VALUE="*ALL" SIZE=10 MAXLENGTH=10></TD>
  </TR>
  <TR>
    <TD WIDTH="169"> </TD>
    <TD WIDTH="118"><INPUT TYPE="submit" NAME="Submit" VALUE="Get Items Information"></TD>
  </TR>
</TABLE></CENTER>
<HR SIZE="5"></HR>
</FORM>

This file uses the servlet <I>ItemServlet</I> to retrieve information from the AS/400 system.

In the FORM tag, the ItemServlet is specified to process the request from this HTML page. The input field “partno” has a default value of *ALL, which means if you don’t enter a value, the servlet will retrieve all records from the ITEM table.

Next we use AAT to add the HTML file to the Web archive file (wsws.war):

1. Use AAT to open the <strong>wsws.war</strong> file.
2. Select <strong>Files-> ResourceFiles-> Add Files</strong>, as shown in Figure 3-23.
3. As shown in Figure 3-24, locate the directory that includes the HTML pages and images. Click the directory on the left pane, and select all the files in the right pane. Click Add to add these files, and click OK to save the settings.

4. In the main AAT window, select File-> Save to save the changes to the wsws.war Web module.
5. Before re-deploying this Web module on WebSphere Application Server, uninstall the old OrderEntry application. Open the browser-based WebSphere console.

6. Select **Nodes** and drill down until you can select **Enterprise Applications** in the left pane.

7. Select the **OrderEntry** application in the right panel and click **Stop**. See Figure 3-25.

8. Once the application is stopped, select the **OrderEntry** application and click **Uninstall**.

9. On the next page, accept the defaults as shown in Figure 3-26 and click **Uninstall** again.
10. We uninstalled the OrderEntry application from WebSphere Application Server. Now, we install it again. The steps to install the application are exactly the same as those explained 3.3.4, “Installing the ItemServlet Web module on the Single Server” on page 91. After you install the application, don’t forget to restart the server.

11. Now you can access the HTML page by entering the following URL:

   http://sysname:port/OrderEntry/ItemServlet.html

   Here sysname is the iSeries server name and port is the internal HTTP server port. The page shown in Figure 3-27 appears.
12. The default value for Part No. is *ALL. We can change it to different value, for example 0000001 and click the **Get Items Information** button. The record is returned from ItemServlet and displayed on the browser as shown in Figure 3-28.

### 3.3.6 Packaging ItemPoolServlet

Packaging ItemPoolServlet into the Web module with Application Assembly Tool requires almost the same steps as those in 3.3.3, “Packaging ItemServlet” on page 88.
We add it to the wswhs.war file. We define a new Web component named ItemPoolServlet to point to the servlet named nservlets.ItemPoolServlet in the wswspool.jar file. We then define a Servlet Mapping that maps ItemPool to the ItemPoolServlet component.

Then we define the initialization parameters for the ItemPoolServlet component. Three parameters are required as listed in Table 3-1.

Table 3-1  ItemPoolServlet initialization parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>datasource</td>
<td>jdbc/NativeDS</td>
</tr>
<tr>
<td>user</td>
<td>A valid user ID</td>
</tr>
<tr>
<td>password</td>
<td>A valid password</td>
</tr>
</tbody>
</table>

Finally we add supporting classes by selecting Files-> Classes Files->Add Files. We add all the classes from the access package of the wswspool.jar file.

3.3.7 Installing the ItemPool Web module on the Single Server

The process to install ItemPool Web module on WebSphere Application Server is similar to the one outlined in 3.3.4, “Installing the ItemServlet Web module on the Single Server” on page 91.

We first stop the OrderEntry application and uninstall it. We then install the OrderEntry application again according to the steps in 3.3.4, “Installing the ItemServlet Web module on the Single Server” on page 91.

Before we stop the server and restart it, we need to create the JDBC driver for iSeries and define the data source used by the ItemPool application. An example of creating a data source for WebSphere Application Server Version 4.0 Advanced Single Server Edition is shown in Chapter 3 in the redbook WebSphere 4.0 Installation and Configuration on the IBM zSeries Server, SG24-6815.

After creating the data source, we stop and restart the server according to the steps in 3.3.4, “Installing the ItemServlet Web module on the Single Server” on page 91. After the server is started, we access the ItemPool application by using the following URL:

http://sysname:port/OrderEntry/ItemPool

The output is shown in Figure 3-29.
3.3.8 Packaging and deploying CallJSP

Packaging CallJSP into a Web module with Application Assembly Tool is similar to the process that is explained in 3.3.6, “Packaging ItemPoolServlet” on page 98. We do not go through it in detail here.

After you define the component and Servlet Mapping, you must add supporting classes, which include the DataBean class in the nservlets package. To add JSP pages into a Web module, select **Files->Resource Files->Add Files** as shown in Figure 3-30.
Figure 3-30   Adding JSP pages into the Web module

Select the directory that includes the output.jsp page and add it into this Web module. Save the new WAR file.

The process to install it on WebSphere Application Server is exactly the same as the one that is explained in 3.3.7, “Installing the ItemPool Web module on the Single Server” on page 99. After the installation, restart the server and access the application by using the following URL:

http://sysname:port/OrderEntry/callJSP

The result page is shown as Figure 3-31.
You can also call it with a parameter, such as:

http://sysname:port/OrderEntry/callJSP?value=hello

### 3.3.9 Packaging and deploying ItemSessionServlet

Since the steps of packaging and deploying ItemSessionServlet are similar to the steps that are outlined in the previous sections, we do not go through them again. The Session Manager of WebSphere is enabled to support cookies by default, so it is not necessary to make any changes to WebSphere Application Server settings.

### 3.3.10 Installing the OrderEntry application on Advanced Edition

In this section, we install the Web application under WebSphere Application Server Version 4.0 Advanced Edition. We use the Advanced Edition Administrative Console, which is not browser based, but a client application.

1. To start the Advanced Edition console, open a workstation command prompt window and change directory as shown here:
   
   cd \WebSphere\install\AppServer\bin

2. Enter the following command to start the WebSphere Administrative Console:

   adminclient sysname port

   Here **sysname** is name of iSeries server name and **port** is the administrative server port. Wait until you see the message **Console Ready** in the Administrative Console, as shown in Figure 3-32. Now, you can start to deploy the code.
3. Since the OrderEntry application needs a data source to access the iSeries database, we need to define a data source. For information about how to create a data source, see Chapter 3 in the redbook *WebSphere 4.0 Installation and Configuration on the IBM @server iSeries Server*, SG24-6815.

4. Install the OrderEntry application using the Administrative Console. To do this, copy the wsws.war file from the local workstation drive to the IFS directory (named wsws) on the iSeries. The console can now access this file for the installation.

5. On the console, select the wizards icon and click **Install Enterprise Application**, as shown in Figure 3-33.
6. The Install Enterprise Application Wizard page appears as shown in Figure 3-34. Make sure the **install stand-alone module** radio button is selected. Click the **Browse** button next to Path to locate the wsws.war file in the wsws IFS directory. Enter **OrderEntry** for Application name and **/OrderEntry** for the Context root.

7. Once you enter all these parameters, click the **Next** button, until you see the Completing the Application Installation window as shown in Figure 3-35. Click **Finish** and then **OK** on the confirmation dialog to install the application.
8. Stop and restart the application server to make the application available. As shown in Figure 3-36, right-click the server and select **Stop**. After it stops, select **Start** to start it again.

9. Once the application server is ready, we can test through either the internal HTTP server or an external HTTP server. For the external HTTP server, there is an additional step to perform. You must re-generate the plug-in configuration. To do this, use the topology pane on the left to display the node, right-click and select **Regen Webserver Plugin**, as shown in Figure 3-37. A completion message shows if the re-generation is successful.
10. Test the OrderEntry application by entering the following URL:

   http://sysname:port/OrderEntry/ItemServlet

   Here sysname is the name of iSeries server and port is the internal or external HTTP server port for the WebSphere Advanced Edition server instance. The output is displayed in the browser, as shown in Figure 3-38.

   **Figure 3-38   ItemServlet result**

   **Here are the results of your query:**

<table>
<thead>
<tr>
<th>ITEM ID</th>
<th>Item Name</th>
<th>Price</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>WEBSPHERES REDBOOK</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>000002</td>
<td>Radio_Controlled_F</td>
<td>$96.86</td>
<td>DELUXE MODEL</td>
</tr>
<tr>
<td>000003</td>
<td>Change_Machine</td>
<td>$52.55</td>
<td>HANDLES SEVERAL CURRENCES</td>
</tr>
<tr>
<td>000004</td>
<td>Baseball_Tickets</td>
<td>$91.14</td>
<td>MINNESOTA TWINS</td>
</tr>
<tr>
<td>000005</td>
<td>Twelve_Man_Two_Pencilt</td>
<td>$50.58</td>
<td>YELLOW IN COLOR</td>
</tr>
</tbody>
</table>

**Tip:** If the external HTTP server fails to find the servlet, you may need to restart the HTTP server.
We can also test ItemPool or callJSP by using the following URLs:

http://sysname:port/OrderEntry/ItemPool
http://sysname:port/OrderEntry/callJSP

### 3.3.11 Packaging the MyHelloWorldApp enterprise application

Deploying applications that contain EJBs is different than deploying applications that contain only servlets or JSPs. They must be deployed using EAR files.

In Chapter 10, “Building Java applications with Enterprise JavaBeans” on page 315, we develop the MyHelloWorldApp application, which uses Java clients, servlets, and EJBs. We use WebSphere Studio Application Developer for the development. With Application Developer, it is not necessary to go through Application Assembly Tool to package the modules. Application Developer is capable of creating EAR files directly that can be deployed on WebSphere Application Server 4.0.

Since Application Developer is a new tool, many Java developers still work with non-J2EE compliant tools like VisualAge for Java to develop EJB-based applications. In this case, AAT is the next step to create an EAR file for deployment on WebSphere Application Server 4.0. In this section, we show how to combine EJBs, servlets, and Java applications into an .ear file.

Here, it is understood that we have already developed the same MyHelloWorldApp application with VisualAge for Java 4.0 and now want to deploy it. We follow these steps:

1. Export the EJB part. As shown in Figure 3-39, click the EJB tab in VisualAge for Java. Select the **EJB group**, and then right-click and select **Export-> EJB 1.1 JAR**.

![Figure 3-39 Exporting the EJB group to the EJB 1.1 Jar](image)

Export to the MyEJBs.jar file in the wsws directory. We export both class and Java source into the JAR file.
2. Export the EJB client application. Just like exporting servlets, simply right-click the **HelloWorldClient** application and select **Export**, as shown in Figure 3-40. Save it as **HelloWorldClient.jar** in the wsws directory.

![Figure 3-40  Exporting HelloWorldClient to a JAR file](image)

3. Export the servlet. We select the **HelloEJBServlet** servlet and save it to **HelloEJBServlet.jar** file in the wsws directory.

We now have three JAR files:
- **MyEJBs.jar**: Includes all the EJBs
- **HelloWorldClient.jar**: Includes the EJB client application
- **HelloEJBServlet.jar**: Includes the servlet

4. Create an enterprise application based on these three JAR files. We create modules for each JAR file and then bundle them together. Figure 3-41 shows the flow we follow to create the enterprise application:
   a. Create an EJB module from the EJBs exported from VisualAge for Java (**MyEJBs.jar**).
   b. Create an Application client module from the applications exported from VisualAge for Java (**HelloWorldClient.jar**).
   c. Create a Web module from the servlets exported from VisualAge for Java (**HelloEJBServlet.jar**).
   d. Combine the modules into an enterprise application.
   e. Deploy the Enterprise Application in WebSphere Application Server 4.0. We deploy to WebSphere Application Server Version 4.0 Advanced Edition.
Creating the EJB module for MyEJBs.jar

First we create an EJB module to hold the EJBs that we exported. Then we follow these steps:

1. Start Application Assembly Tool (AAT). In the welcome page, click Cancel.
2. Open the MyEJBs.jar file in the wsws directory.
3. Select Session Beans-> MyHelloWorld as shown in Figure 3-42.

Figure 3-41 Creating an enterprise application
4. You can now see the details about the EJB. Select the **Bindings** tab and make sure that the JNDI name is **MyHelloWorld**.

5. We need to generate the EJB deployment code. This includes all the remote method invocation (RMI) code necessary to allow the EJB to communicate through an RMI/IIOP interface. Select **File-> Generate Code for Deployment**. In the Generate code for deployment window (Figure 3-43), click the **Generate Now** button.
6. While the code generation is being processed, the Generate Now button is grayed out. Wait for the code generation to finish. When it is done, scroll down the messages pane and ensure that you have no errors.

7. Close the Generate code for deployment window and check the wsws directory for Deployed_MyEJBs.jar file. It contains all the code necessary to actually run the EJBs.

Creating the client module for HelloWorldClient.jar
Next, we create a client application to hold the Java application. Follow these steps:

1. In AAT, select File-> Wizards->Create Application Client Wizard.

2. In the wizard page (Figure 3-44), enter MyHelloWorldClient for Display Name and x:\wsws\MyHelloWorldClient.jar for File Name. Replace x with the drive letter where the wsws directory is stored. Then click Next.
3. In the Create Client Application: Adding Files window, click Add to display the Add File window.

4. In the Add Files window (Figure 3-45), click the Browse button to locate the HelloWorldClient.jar file in the wsws directory.

5. Select the HelloWorldClient.class file in the right pane and click Add to add it to the selected files pane. Click OK to return to the previous page and then click Next.

6. In the Specifying Additional Application Client Module Properties window (Figure 3-46), we need to specify the main class that client container can invoke. Click the Browse button.
Figure 3-46 Specifying additional application properties

7. In the Select file for Main class window, select the **HelloWorldClient.class** file and click **OK**. We now see that the Main Class has been filled in. Click **Next**.

8. In the Choosing Application Icons window, click **Next** again.

9. Once you see the Adding EJB References window, click the **Add** button. The Add EJB References window appears (Figure 3-47).

Figure 3-47 Adding EJB references
10. Enter **MyHelloWorld** in the Name field. Click the **Browse** button next to Home to locate the Deployed_MyEJBs.jar. As shown in Figure 3-48, select **MyHelloWorldHome.class** as the Home interface class and click **OK**.

![Figure 3-48 Selecting MyHelloWorldHome.class for Home](image)

11. Use the same technique to select **MyHelloWorld.class** as the Remote Interface from Deployed_MyEJBs.jar file. Make sure the EJB type is set to **Session**.

12. As shown in Figure 3-49, the EJB Home, Remote, and Type parameters are set. Click **OK** to save the settings.

![Figure 3-49 EJB references result](image)
13. In the Create Application Client Wizard window, click **Next**.
14. Click **Next** again on the Adding Resource References window.
15. Click **Finish** to create the EJB application client module.
16. Set the EJB bindings for the client application. In the AAT window, click **EJB References** on the left pane and click the **Bindings** tab on the right for the MyHelloWorld bean. As shown as Figure 3-50, enter **MyHelloWorld** for the JNDI name and click **Apply**.
17. The application client is now complete. We save it by selecting **File-> Save** to save the work as MyHelloWorldClient.jar in the wsws directory.

![Figure 3-50 Setting the bindings for an application client](image)

**Creating a Web module for HelloEJBServlet.jar**

Next, we create a Web module that contains the HelloEJBServlet:

1. Select **File-> New-> Web Module**.
2. In the main AAT window, right-click **Web Components** and select **New** to create a component based on HelloEJBServlet.
3. In the New Web Component window, shown as Figure 3-51, enter **HelloEJBServlet** for Component name. Click the **Browse** button to locate the **HelloEJBServlet.jar** file in the wsws directory.
4. In the Select file for classname window, shown as Figure 3-52, select the `HelloEJBServlet.class` servlet from `tservlets` package. Click OK.
5. The New Web Component window now contains a component name and a Class name. Click OK to save the settings.

6. Define the Servlet Mappings. We define HelloEJBServlet as the URL pattern for the HelloEJBServlet servlet.

7. Define the EJB references:
   a. Right-click EJB Reference and select New.
   b. In the EJB Reference window, shown as Figure 3-53, enter MyHelloWorld in the Name field.
   c. Click the Browse button next to Home. Locate the Deployed_MyEJBs.jar file in the wsws directory.
   d. Select MyHelloWorldHome.class for Home interface.
   e. Use the same technique to select MyHelloWorld.class for Remote interface.
   f. Make sure the Type is Session.
   g. Click the Bindings tab. Enter MyHelloWorld as the JNDI name, and click OK to save the settings.

8. The Web module packaging is complete. Select File-> Save As to save it as HelloEJBServlet.war in the wsws directory.

Creating the MyHelloWorldApp enterprise application

We now have three modules:
- Deployed_MyEJBs.jar as the EJB module
- MyHelloWorldClient.jar as the client module
- HelloEJBServlet.war as a Web module
We also set the EJB local bindings in the Web module and client modules. We need to assemble them together as an enterprise application:

1. In AAT, select **File -> Wizard -> Create Application Wizard**.

2. In the main Create Application wizard window (Figure 3-54), enter `MyHelloWorldApp` for Display Name and `x:\wsws\MyHelloWorldApp.ear` for File Name. Replace `x` with the drive letter where the `wsws` directory is stored.

   ![Figure 3-54 Specifying enterprise application properties](image)

3. Keep clicking the **Next** button until you reach the Adding EJB Modules window. Click **Add** and locate the `Deployed_MyEJBs.jar` file in the `wsws` directory. Click **Open**.

   ![Figure 3-55 Confirming the values for the EJB modules](image)

   In the Confirm values window, shown as Figure 3-55, leave Alternate DD blank (represents an alternate Deployment Descriptor). Click **OK** and then **Next**.

4. Add the Web modules. In the Adding Web Modules window, use the same technique to select the `HelloEJBServlet.war` file in the `wsws` directory. When the Confirm values window (Figure 3-56) appears, leave Alternate DD blank and enter `/HelloWorld` for the Context root. Click **OK** and then **Next**.
5. In the Adding Application Clients Modules window, use the same technique to select the `MyHelloWorldClient.jar` file in the wsws directory. When the Confirm values window (Figure 3-57) appears, leave Alternate DD blank. Click OK and then Next.

6. We have now added the three modules. Click Finish to save the settings.

7. You return to the AAT main window with the newly created enterprise application, as shown in Figure 3-58. Click the Bindings tab and set the application name to MyHelloWorldApp. Click Apply to save the change.

8. Save the configuration by selecting File-> Save to save it to the MyHelloWorldApp.ear file in the wsws directory.
3.3.12 Installing the MyHelloWorldApp application on Advanced Edition

In this section, we install the MyHelloWorldApp enterprise application on WebSphere Application Server Version 4.0 Advanced Edition using the Administrative Console.

1. Copy the **MyHelloWorldApp.ear** file from our local drive to the iSeries IFS wsws directory, so the Administrative Console can access it when creating enterprise applications.

2. As in 3.3.10, “Installing the OrderEntry application on Advanced Edition” on page 102, open a command prompt window to start the Administrative Console. Wait until you see the Console Ready in the Console message.

3. In the Console, select the wizard icon and click **Install Enterprise Application**.

4. The Specifying the Application or Module window (Figure 3-59) appears. Make sure that the **Install Application** radio button is selected. Click the **Browse** button next to Path to locate the **MyHelloWorldApp.ear** in the wsws directory. Enter **MyHelloWorldApp** for the Application name.

![Figure 3-59 Specifying the application parameters](image)

5. Keep clicking **Next** until you reach the Completing the Application Installation window (Figure 3-60). Click **Finish** to complete the installation. Click **No** to the Regenerate the application dialog and click **OK** on the confirmation dialog.
6. As in 3.3.10, “Installing the OrderEntry application on Advanced Edition” on page 102, stop and restart the application server. To support this new application with an external HTTP server, you must regenerate the plug-in configuration.

### 3.3.13 Testing the MyHelloWorldApp application

Now that we have successfully installed the MyHelloWorldApp application on WebSphere Application Server Version 4.0 Advanced Edition, we need to test it. Since we have two kinds of EJB clients – application clients and servlets – we test both of them.

**Testing the application client**

Open a workstation command prompt session and change the directory to:

```
\<WebSphere install>\AppServer\bin
```

Enter the following command:

```
launchclient x:\wsws\myhelloworldapp.ear -CCBootstrapHost=sysname -CCBootstrapPort=port
```

Here `x` is the drive where the `wsws` directory is stored, `sysname` is the name of iSeries server, and `port` is the administrative server bootstrap port.

As shown in Figure 3-61, if everything is working, the following messages are written to the command session:

Running the J2EE Application Client HelloEJB Sample

--lookup was successful

--narrow was successful

--create was successful

Message from HelloEJB: Hello from teamxx

HelloClient ran successfully
Figure 3-61  Running the application client for MyHelloWorldApp

Testing the servlet

Now we test the servlet. Open a browser and enter the URL:

http://sysname:port/HelloWorld/HelloEJBServlet

Here sysname is the name of iSeries server and port is your HTTP server port (it can be either an external HTTP server port or internal HTTP server port).

If everything works properly, the message appears in the browser, as shown in Figure 3-62.

![Image](http://as05s543/HelloWorld/HelloEJBServlet - Microsoft Internet Explorer)

Message from Hello EJB: Hello from Team43

Figure 3-62  Testing servlet for MyHelloWorldApp

To view the output from the System.out.println statements, we can look in the WebSphere log file that is located in the /QIBM/UserData/WebASAdv4/xxxx/logs directory (xxxx is your WebSphere instance name). Use a workstation editor to view the ServerName_stdout.log file and scroll to the end of the file. As shown in Figure 3-63, you see the messages from the servlet.
Now we have finished installing and testing the MyHelloWorldApp enterprise application.

### 3.3.14 Packaging the OrderEntryApp enterprise application

In Chapter 10, “Building Java applications with Enterprise JavaBeans” on page 315, we develop another enterprise application, OrderEntryApp, which integrates Java clients, servlets, and EJBs. The MyHelloWorldApp application is very simple and only uses only one session bean. In OrderEntryApp, we use multiple session beans and entity beans (both container-managed persistence and bean-managed persistence) to simulate a more complex transaction-based application. It is shown in Figure 3-64.
OrderEntryClerk and OrderPlacement are session beans. ItemEB, CustomerEB, StockEB, OrderEB, and DistrictEB are entity beans. We use Java applications and servlets as EJB clients. We can also treat session beans as clients of entity beans, where session beans represent business logic and entity beans represent data. When we package the application with AAT, we configure the bindings for Java application clients and servlets to find the EJBs. We also need to configure the bindings for session beans to find the entity beans.

Assume we have developed the OrderEntryApp application with VisualAge for Java 4.0, not with WebSphere Studio Application Developer. Application Developer is capable of packaging .ear files directly. We export our code into JAR files. We create four JAR files:

- OrderEntryBeans.jar: Includes all the EJB beans
- OrderEntryClient.jar: Includes the EJB client application
- OrderEntryWar.jar: Includes the servlet
- nonejb.jar: Includes supporting classes for the EJBs

Next, we create an enterprise application based on these JAR files. We create modules for each JAR file and then bundle them together.

Figure 3-65 shows the flow of the application packaging. Since the packaging process is very similar with the one explained in 3.3.11, “Packaging the MyHelloWorldApp enterprise application” on page 107, we only focus on the differences for the OrderEntryApp application.

Creating the EJB module for OrderEntryBeans.jar

As in “Creating the EJB module for MyEJBs.jar” on page 109, start AAT and then follow these steps:

1. Open the OrderEntryBeans.jar file.
2. Select Session Beans-> OrderEntryClerk. You see the details of the bean as shown in Figure 3-66.
3. Click the **Bindings** tab and make sure the JNDI name is `OrderEntryClerk`.

4. Repeat this for the `OrderPlacement` session bean and make sure the JNDI name is `OrderPlacement`.

5. Use the same technique to select the `Customer`, `District`, `Item`, `Order`, and `Stock` entity beans under **Entity Beans**, and make sure their JNDI names are `Customer`, `District`, `Item`, `Order`, and `Stock` respectively.

6. Add EJB references. Because the `OrderEntryClerk` session bean invokes the `OrderPlacement` session bean for placing orders in our application, add this reference for `OrderEntryClerk`. Select **Session Beans**-> `OrderEntryClerk`-> **EJB References**-> New. As shown in Figure 3-67, enter `OrderPlacement` for Name and click the link pulldown to select the `OrderPlacement` bean. Then click **OK**.

---

**Figure 3-66 OrderEntryClerk Session Bean**

```plaintext
3. Click the **Bindings** tab and make sure the JNDI name is `OrderEntryClerk`.

4. Repeat this for the `OrderPlacement` session bean and make sure the JNDI name is `OrderPlacement`.

5. Use the same technique to select the `Customer`, `District`, `Item`, `Order`, and `Stock` entity beans under **Entity Beans**, and make sure their JNDI names are `Customer`, `District`, `Item`, `Order`, and `Stock` respectively.

6. Add EJB references. Because the `OrderEntryClerk` session bean invokes the `OrderPlacement` session bean for placing orders in our application, add this reference for `OrderEntryClerk`. Select **Session Beans**-> `OrderEntryClerk`-> **EJB References**-> New. As shown in Figure 3-67, enter `OrderPlacement` for Name and click the link pulldown to select the `OrderPlacement` bean. Then click **OK**.
```
7. As shown in Figure 3-68, select **EJB References** -> **OrderPlacement** -> **Bindings**. Enter `OrderPlacement` for the JNDI Name and click **Apply**.
8. The OrderPlacement session bean invokes entity beans such as Customer, District, Order, and Stock in the application. We need to add EJB references for all these entity beans to the OrderPlacement session bean. Select Session Beans -> OrderPlacement -> EJB References -> New. Use the same technique to add EJB references to the OrderPlacement bean for the following EJBs:

- Customer
- District
- Order
- Stock

Figure 3-69 shows the detailed EJB references for the OrderPlacement bean. Don’t forget to click the Bindings tab to set the JNDI name for each EJB reference.

![Application Assembly Tool]

Figure 3-69 OrderPlacement EJB references

9. Add supporting classes into the module for the EJBs. Right-click Files and select Add Files. In the Add Files window, shown as Figure 3-70, click the Browse button to locate the nonejb.jar file in the wsws directory. Select all the files in the interfaces package and click Add. Then click OK to return to the main AAT window.
10. Generate the deployed code. Select File-> Generate Code for Deployment. In the Generate code for deployment window (Figure 3-71), leave Working Directory as the default. The EJBs use classes from other packages like:

- IBM Toolbox for Java (jt400.jar)
- Application supporting classes(nonejb.jar)
- WebSphere Command package (ace.jar)

We need to add them into the classpath with the following entry so they can be found:

```
x:\jt400\jt400.jar;x:\wsws\nonejb.jar;x:\WebSphere\AppServer\lib\ace.jar
```

Here x is the workstation drive on which these files are stored.

We click the Generate Now button to generate the deployed code. Wait for the code generation to finish. When it is done, scroll down the messages and make sure that you have no errors.

11. Close the Generate code for deployment window. Check the wsws directory. There is a new file named Deployed_OrderEntryBeans.jar, which contains all the code necessary to actually run the EJBs.
Creating the client module for OrderEntryClient.jar

In AAT, follow these steps:

1. Select **File-> Wizards-> Create Application client Wizard**. On the following page (Figure 3-72), enter **OrderEntryClient** for Display Name and **x:\wsws\MyOrderEntryClient.jar** for File name (replace x with the drive letter where the wsws directory is stored). Click **Next**.
2. On the Create Client Application Add Files window, click the **Add** button. On the Add Files window, click **Browse** to locate the **OrderEntryClient.jar** file in the ws ws directory. As shown in Figure 3-73, select all the files from the EJBApplications package and click **Add**. Use the same technique to add all the files from the Support package. Click **OK** to return to the Add Files window.

![Figure 3-73   Adding class files into client module](image)

3. Click **Next**. In the Specifying Additional Application Client Module Properties window, click **Browse**. Select the **OrderEntryWdwJ.class** from the EJBApplications package as the Main class as shown in Figure 3-74. Click **OK**. You return to the previous page.

![Figure 3-74   Specifying the main class for the client module](image)

4. Keep clicking **Next** until you reach the Adding EJB References window. Click **Add** and on the following page, enter **OrderEntryClerk** in the Name field. Click **Browse** next to **Home**.
to locate the **Deployed_OrderEntryBeans.jar** file in the ws ws directory. As shown in Figure 3-75, select the **OrderEntryClerkHome.class** file and click **OK**.

![Select file for Home](image)

**Figure 3-75   Adding the Home interface for EJB reference**

Use the same technique to set the OrderEntryClerk class file for the Remote interface.

5. Now we have set all the values for EJB reference. As shown in Figure 3-76, make sure that Type is set to **Session**.

![Add EJB References](image)

**Figure 3-76   Setting the EJB reference**

6. Keep clicking **Next** until you see a **Finish** button. Click it to finish the settings.
7. In the AAT main window, click **EJB References** on the left pane and click the **Bindings** tab for the OrderEntryClerk bean. As shown in Figure 3-77, enter OrderEntryClerk for JNDI name and click **Apply** to save the settings.

![Application Assembly Tool](image)

**Figure 3-77** Setting the bindings for EJB references

8. The Application Client is now complete. Select **File-> Save** to save the work in the wsws directory as MyOrderEntryClient.jar.

**Creating a Web module for OrderEntryWar.jar**

In AAT, follow these steps:

1. Select **File-> New-> Web Module**. In the AAT main window, right-click **Web Components** and select **New** to create a new component. In the New Web Component window, enter ItemSessionServlet for the Component name. Click **Browse** next to Class name to add the servlet class.

2. In the Select file class name window, click the **Browse** button to locate the **OrderEntryWar.jar** file in the wsws directory. Select **ItemSessionServlet.class** in the tservlets package as the servlet, as shown in Figure 3-78. Click **OK** to finish the selection.
3. You return to the New Web Component window. Click OK to save the settings.

4. Create a new servlet mapping. We use ItemSessionServlet as the URL pattern and select ItemSessionServlet as the servlet.

5. Since this servlet invokes the OrderEntryClerk Session Bean for the transactions, we need to add EJB References for it. Using the same steps in “Creating the client module for OrderEntryClient.jar” on page 129, we create an EJB reference to the OrderEntryClerk session bean and also make sure the Bindings value is set to OrderEntryClerk.

6. ItemSessionServlet uses initialization parameters for userid, password, system name, and bootstrap port. We need to configure them.

   a. Select Web Components-> ItemSessionServlet-> Initialization Parameters-> New. Add the parameter Name/Value pairs as listed in Table 3-2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>A valid user ID</td>
</tr>
<tr>
<td>password</td>
<td>A valid password</td>
</tr>
<tr>
<td>system</td>
<td>iSeries server name</td>
</tr>
<tr>
<td>port</td>
<td>WebSphere Application Server bootstrap port</td>
</tr>
</tbody>
</table>

   b. Select File-> Save As to save the work in the wsws directory as OrderEntryServlet.war.

   c. Since we use another servlet named CartServlet for shopping cart processing, follow the same procedure to add the CartServlet.
i. Add a new Web Component CartServlet to point to CartServlet in tservlets package (same as with defining ItemSessionServlet).

ii. Create a new Servlet Mapping called CartServlet to specify the CartServlet component.

iii. Because CartServlet invokes the OrderPlacement session bean for transactions, we need to add a new EJB reference for CartServlet. Using the same technique as for creating an EJB reference to the OrderEntryClerk session bean, we create an EJB reference called OrderPlacement to point to OrderPlacement session bean. We set the binding as OrderPlacement.

iv. CartServlet also uses initialization parameters when calling the init() method. The parameters are the same as those for ItemSessionServlet. Following the same steps, we define the initial parameters of userid, password, system and port for CartServlet.

7. We need to add the supporting classes for the Web module. In AAT, expand Files in the left pane, right-click Class Files, and then select Add Files, as shown in Figure 3-79.

![Figure 3-79 Adding additional class files for the Web module](image)

8. In the Add files window, click the Browse button to locate the OrderEntryWar.jar file in the wsws directory. Select all the files from the ShopData package and click Add. Use the same technique to add all the files from the Support and tservlets package. Click OK to complete the settings.

9. Select File-> Save to save the work. Now we have finished the tasks for the Web module. A new file called OrderEntryServlet.war is created in the wsws directory.
Creating an enterprise application for the OrderEntryApp

Now we have the following three modules:

- Deployed_OrderEntryBeans.jar for EJBs
- MyOrderEntryClient.jar for application client
- OrderEntryServlet.war for servlets

Based on these files, we create the enterprise application using AAT:

1. In AAT, select File-> Wizards-> Create Application Wizard. In the Create Application Wizard initial page (Figure 3-80), enter OrderEntryApp for Display name and x:\wsws\OrderEntryApp.ear for File Name. Replace x with the workstation drive letter.

   ![Figure 3-80 Creating OrderEntryApp enterprise application](image)

2. Keep clicking Next until the Adding EJB Modules window appears. Click Add to add the Deployed_OrderEntryBeans.jar file from the wsws directory. When the Confirm values dialog (Figure 3-81) appears, leave Alternate DD blank and click OK.

   ![Figure 3-81 Confirming the values for adding EJB modules](image)

3. Click Next and in the Adding Web Modules window. Click Add to add the OrderEntryServlet.war file. In the Confirm values window, as shown in Figure 3-82, leave Alternative DD blank, enter /OrderEntry2 for Context root, and click OK.

   ![Figure 3-82 Confirming the values for adding Web modules](image)
4. Click **Next**. In the Adding Application Clients Modules window, click **Add** to add the **MyOrderEntryClient.jar** file. In the confirm value window, leave Alternate DD blank and click **OK**.

5. Click **Next** and then click **Finish** to complete the settings.

6. In the AAT main window, click the **OrderEntryApp** application in left pane and click the **Bindings** tab in the right pane. Fill in the Enterprise application name with **OrderEntryApp**, as shown in Figure 3-83. Click **Apply** to save the changes.

7. Select **File-> Save** to save the file. A new file named **OrderEntryApp.ear** is created in the **wsws** directory. This is the file where we can install on WebSphere Application Server 4.0.

### 3.3.15 Installing the OrderEntryApp application on Advanced Edition

In this section, we install the OrderEntryApp enterprise application on WebSphere Application Server Version 4.0 Advanced Edition with the Administrative Console.

1. Copy the **OrderEntryApp.ear** file from our local drive to the iSeries IFS **wsws** directory.
Now, the Administrative Console can access it when creating enterprise applications.
2. In the Console, select the wizard icon and select **Install Enterprise Application**. The Specifying the Application or Module window (Figure 3-84) appears. Make sure that the **Install Application** radio button is selected. Click the **Browse** button next to Path to locate the **OrderEntryApp.ear** in the wsws directory. Enter **OrderEntryApp** for the Application name.

![Figure 3-84   Installing the application](image)

3. Keep clicking **Next** until you reach the Specifying the Default Datasource for EJB Modules window. Click the **Select Datasource** button and select the **NativeDS** Datasource. Enter a user ID and password (twice) and click **OK**.

4. Click **Next**. In the Specifying Data Sources for individual CMP beans window, select all three of the CMP beans. Click the **Select Datasource** button. Then select the **NativeDS** Datasource. Enter a user ID and password (twice) and click **OK**. Now we have a Data Source for each CMP bean as shown in Figure 3-85.

**Note:** We use the NativeDS DataSource. See Chapter 3 in the redbook *WebSphere 4.0 Installation and Configuration on the IBM @server iSeries Server*, SG24-6815, for information on how to create a DataSource.
5. Keep clicking **Next** until you reach the Completing the Application Installation window. Click **Finish** to install the application. When the Regenerate the application dialog appears, click **No**.

6. Now we have successfully installed the OrderEntryApp enterprise application on WebSphere Application Server Version 4.0 Advanced Edition. We need to stop the server. Before we restart it, we set up the classpath for this application. The application server needs to find the classes in the IBM Toolbox for Java. As shown in Figure 3-86, select the application server and click the **JVM Settings** tab in the right pane. Add the following files to the Classpaths settings and click **Apply**.

   - `/QIBM/ProdData/Java400/jt400ntv.jar`
   - `/QIBM/ProdData/http/public/jt400/lib/jt400.jar`
3.3.16 Testing the OrderEntryApp application

Now we have successfully installed the OrderEntryApp application on WebSphere Application Server Advanced Edition. The next step is to test it.

Since we have two kinds of EJB clients – application clients and servlets – we test both of them.

Testing the application client
To test the application client, follow these steps:

1. Open a workstation command prompt session and change the directory to the 
   \WebSphere\install\AppServer\bin directory. Enter the following command:

   \launchclient x:\wsws\orderentryapp.ear port -CCBootstrapHost=sysname
   -CCBootstrapPort=port

   Here x is the drive where the wsws directory is stored, sysname is the name of iSeries server, and port is the administrative server bootstrap port.

2. Once the client application window (Figure 3-87) appears, select Connect-> Connect.
   Enter the iSeries server name, user ID, and password in the signon dialog, and click OK.
3. Once you successfully sign on (a status message indicates this), click the **List Customers** button, select one customer, and click **OK**.

4. Click the **List Items** button, select an item, and click **OK**. Dismiss the List Items window. Return to the main window, enter a quantity for the selected item, and click the **Add Item** button. As shown in Figure 3-88, this item is added to the list. You can keep adding new items.
5. Click the **Submit** button to place an order. As shown in Figure 3-89, if everything works correctly, an order number is returned to the Status field.
Testing the servlet
To test the servlet, follow these steps:

1. Open a browser to access the servlet by entering the URL:

   http://sysname:port/OrderEntry2/ItemSessionServlet

   Here sysname is the iSeries server name, and port is the external (or internal) HTTP server port.

   If everything works properly, the item list shown in Figure 3-90 appears.

   ![Figure 3-90 Testing the servlet](image)

   2. Select some items and click the **Add to Cart** button to add them to the shopping cart.

   3. After viewing the shopping cart, click the **Check Out** button. As shown in Figure 3-91, enter a customer number (0001 to 0010) and click the **Place Order** button.
If everything works successfully, we see a confirmation message as shown in Figure 3-92.

We have successfully run the OrderEntryApp transaction application in WebSphere Application Server Version 4.0 Advanced Edition.
Introduction to WebSphere Studio Application Developer

This chapter provides an introduction to WebSphere Studio Application Developer. It is a follow-on product to VisualAge for Java and WebSphere Studio. It provides many of the key VisualAge for Java and WebSphere Studio capabilities, plus it adds many new capabilities. It helps you build and test J2EE compliant applications. We also refer to it as Application Developer.

The purpose of this chapter is to get you started with WebSphere Studio Application Developer. We walk you through some simple examples to show you how to navigate in Application Developer and how to customize your workspace. We show you how to do more complex examples – such as building, testing, and deploying servlets, JavaServer Pages, and Enterprise JavaBeans – in other chapters of this redbook.
4.1 WebSphere Studio Application Developer overview

WebSphere Studio Application Developer is the follow-on technology for WebSphere Studio, Professional and Advanced Editions and VisualAge for Java Enterprise Edition. These new tools support the end-to-end development, testing, and deployment of e-business applications.

The new WebSphere Studio products are designed from the ground up to meet the requirements for all new types of applications. These requirements include:

- Open standards
- Java
- XML
- Web Services
- Testing
- Varying levels of integration with other components and ISV products
- Pluggability
- Expandability
- Role-based development
- Increased usability for all users
- Enhanced team support
- Increased speed to market

Application Developer provides integrated development tools for all e-business development roles. These roles include Web developers, Java developers, business analysts, application architects, and enterprise application programmers.

Application Developer brings together the most popular features of WebSphere Studio "Classic", and VisualAge for Java and combines them with the advantages of the latest technology, providing open standards, tool integration, more flexibility, and the ability to tie in existing applications.

Figure 4-1 shows all the tasks you can manage with Application Developer.
Java features
► Ships with JDK 1.3
► Pluggable runtime JDK support per project
  – Specify JRE and rt.jar for running/building/debugging
► Incremental saves
► Java code snippet support (scrapbook)
► Task Sheet (All Problems Page)
► Code Assist
► JDI-based debugger
  – One debugger for local/remote debugging
► Run code with errors
► Refactoring support
  – Rename/move support for method/class/packages
  – Fixes all dependencies for renamed element
  – Method extraction

J2EE features
► Full J2EE 1.2 Support
► WAR/EAR deployment support
► All metadata exposed as XMI
► Enhanced Unit Test Environment for J2EE
  – IBM WebSphere Application Server or Apache Tomcat
  – Create multiple projects with different Unit Test configurations/instances
    • Version Unit Test Environment
    • Share Unit Test Environment configuration across developers
- Object-oriented mapping for EJBs
  - Top-down/bottom-up/meet-in-the-middle
- Updated EJB Test Client
  - HTML-based
  - J2EE Programming model
  - Built-in JNDI Registry Browser
- Enterprise Connectors (separate plug-in offering)
  - JCA Connector based
- Built-in J2EE perspective provides the useful views for the EJB/J2EE developer

**Web features**
- HTML/JSP editing
  - WYSIWYG page design, source editing, page preview
- WAR import/export
- Links View (Relations)
  - View HTML/JSP and all links referenced in HTML/JSP
- Parsing/link management
  - Automatically fix links when resources are moved/renamed
- Built-in servlet, database, JavaBean wizards
  - Quick generation of HTML/Servlet/JSP
- Built-in JSP debugger
- Site-style and template support
- Built-in Web perspective provides the useful views for HTML/JSP developer

**XML features**
- Built-in XML tooling provides integrated tools/perspectives to create XML-based components:
  - DTD Editor
    - Visual tool for working with DTDs
    - Create DTDs from existing documents
    - Generate an XML Schema from a DTD
    - Generate JavaBeans for creating/manipulating XML documents
    - Generate an HTML form from a DTD
  - XML Schema Editor
    - Visual tool (Design/Source) for working with an XML Schema
    - Generate DTDs from an XML Schema
    - Generate JavaBeans for creating/manipulating XML documents
  - XML Source Editor
    - DTD/Schema validation
    - Code Assist for building XML documents
- Additional XML Tools:
  - XML Mapping Editor
    - Generate XSL to map XML between DTDs/schemas
Chapter 4. Introduction to WebSphere Studio Application Developer

4.2 Getting started with Application Developer

Next we help you get started with Application Developer. We provide step-by-step instructions to show how to perform some general tasks with WebSphere Studio Application Developer. These tasks are:

- XSL Trace Editor
  - Trace XSL transformation
  - Examine relationships between the result node, the template rule, and the source node
- XML to/from relational databases
  - Generate XML, XSL, XSD from an SQL query
- RDB/XML Mapping Editor
  - Map columns in a table to elements and attributes in an XML document
  - Generate a Database Access Definition (DAD) script to compose/decompose XML documents to/from a database
  - DAD is used with DB2 XML Extender
  - Built-in XML perspective provides the useful views for the XML developer

Web Services features

- Easily consume/construct Web Services:
  - Discover
    - Browse UDDI registry to locate existing Web Services
    - Generate JavaBean Proxy for existing Web Services
  - Create/transform new Web Services from JavaBeans, databases, and so on
  - Deploy Web Service to WebSphere or Tomcat for testing
  - Test (via a built-in test client) immediately local/remote Web Services
  - Publish Web Services to UDDI registry

Relational Database features

- Relational Schema Center
  - Provides views geared for database administrators (DBAs) to:
    - Create Databases
    - Create Tables/Views/Indexes/Keys
    - Generate DDL
  - Online and offline support for working with databases
    - Metadata generated as XMI
  - SQL Query Builder
    - Visually construct SQL statements
      - Insert, update, delete, select supported
    - Metadata generated as XMI
    - SQL/XML mapping
  - Built-in Data perspective provides the useful views for DBAs
4.2.1 Navigating in Application Developer

In Application Developer, there are several ways to see your project. You must choose one method or the other, depending of the role you are playing at the time. These views are called perspectives. To work with perspectives, follow these steps:

1. Start Application Developer by clicking **Start-> Programs-> IBM WebSphere Studio Application Developer-> IBM WebSphere Studio Application Developer.**

   Application Developer starts in the J2EE perspective (this is the default perspective) as shown in Figure 4-2.

   ![Figure 4-2 J2EE perspective](image)

2. Click the **Open perspective** button.

3. Look at the predefined selection of perspectives in the pop-up menu and select **Other**.

4. Select **Help** and click **OK** as shown in Figure 4-3.

   ![Figure 4-3 Open perspective](image)
Chapter 4. Introduction to WebSphere Studio Application Developer

The Help perspective contains the help information for the tool; you can explore any topic. Notice that the perspective has three tabs:

- By type...
- By feature...
- Search

The first two tabs have information ordered by those two references. The last one is for searching any topic that you want in the tool help information.

5. Click the Search tab. Enter Testing EJB in the search text field (you have to replace Enter search string in the search text field) as shown in Figure 4-4.

6. After some time, you see all the topics that contain the string Testing EJB. You can browse through the search results. Notice that the search facility highlights the search string in the text. Click the Advanced button located under the search text field.

7. Select the Search headings only check box as shown in Figure 4-5.
Click OK to set this option and close the window.

8. Click GO again.

9. Now you see only a subset of the references you had before. Observe the differences in
the search results.

Next, we open the Java perspective by following these steps:

1. Select Perspective-> Open-> Other from the menu (Figure 4-6). Select Java from the list
and click OK. This is an alternative method of opening a perspective.

2. In the Java perspective, you can develop Java code for your application, as we do later in
this chapter.

   You can return to the help perspective by clicking the help icon in the perspective. You see
that Application Developer remembers the last search you did.

3. Right-click the Java perspective button on the perspective toolbar. Select Close, as
shown in Figure 4-7. This closes the Java perspective.
4.2.2 Importing resources

When working with Application Developer, you can import code from previous Java developments. You can import Java code or JAR files to an existing project. We explain how to do this in the following steps:

1. Switch to the J2EE perspective by clicking the J2EE perspective button in the perspective toolbar.

2. Create a Java project by selecting File-> New-> Other from the menu, as shown in Figure 4-8.

3. Click Java in the left pane of the pop-up window and select Java Project in the right pane. There is also a button on the toolbar in the Java perspective to perform this task. Click Next as shown in Figure 4-9.

4. Enter JUnit as the project name. Click Finish as shown in Figure 4-10.
Application Developer switches to the Java perspective. Now we are going to import a JAR file:

1. Select **File -> Import** from the menu.
2. Select **Zip file** and click **Next** as shown in Figure 4-11.
3. Click **Browse** next to the Zip file field, and navigate to the zip file. We use the JAR file `junit37src.jar`.

4. Click **Open**.

5. Click **Browse** next to the Folder field. The JAR file is found in the `<Application Developer install directory>`\plugins\org.eclipse.jdt.ui.examples.projects\archive\junit\ directory.

6. Select **JUnit** and click **OK** as shown in Figure 4-12.

![Figure 4-12 Selecting the folder for importing the JAR file](image)

7. Click **Finish** as shown in Figure 4-13.

![Figure 4-13 Clicking Finish to import the JAR file](image)

### 4.2.3 Customizing Application Developer

Now we show you how to customize some of the options of Application Developer to make it more suitable to your personal preferences. Follow these steps:
1. Click **Window-> Preferences** to open the Preferences window as shown in Figure 4-14.

![Figure 4-14 The Preferences window](image)

2. One of the most interesting options is the automatic build. It is enabled by default. You can deselect **Perform build automatically on resource modification** to disable it. If you do this, you need to invoke the build/rebuild project option from the Application Developer menu every time you need to compile the code in your project.

3. Expand the **Workbench** tree. Select **Perspectives**. J2EE is the default perspective. You can select the perspective you want to be your default one. For example, select the Web perspective and click **Make Default** as shown in Figure 4-15.
4. Select **Fonts** from the workbench tree. You can change the default font as shown in Figure 4-16.

5. Click **Server**. On the Server Preferences panel, you can configure the behavior of Application Developer when you run your application on a server, as shown in Figure 4-17.
6. Select **Web Browser**. Here you can choose the settings for the browser. You can choose the internal or another browser for testing as shown in Figure 4-18.

![Server preferences](image1)

**Figure 4-17 Server preferences**

![Web Browser preferences](image2)

**Figure 4-18 Web Browser preferences**

Click **OK**. Application Developer saves and applies your changes to the environment.

There are many more preferences that you can set.
Now we work some with the editors. Follow these steps:

1. Switch to the Java perspective if you are in a different one. You should see a window similar to the one shown in Figure 4-19.

![Figure 4-19 Java perspective](image)

2. To create more space for your Java file editor, you can move the Outline view to a different location:
   a. Click the Outline title bar and drag it to the bottom of the Packages view. The cursor changes its shape as you drag the view through the different areas of the window.
   b. When the cursor changes its shape to an arrow, release the mouse button. You see a window similar to the one shown in Figure 4-20.
Figure 4-20  Moving the Outline view

When you move the views, you can obtain three different cursors as shown in Table 4-1.

Table 4-1  Different cursors

<table>
<thead>
<tr>
<th>Cursor shape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Cursor 1]</td>
<td>The view is added to the border that points the arrow.</td>
</tr>
<tr>
<td>![Cursor 2]</td>
<td>The view is set where you release the cursor; you can resize it however you want. The view will be as a different window.</td>
</tr>
<tr>
<td>![Cursor 3]</td>
<td>This is the stacked folders icon. The view is added to a pane with other views. Then, you can select any view with a tab.</td>
</tr>
</tbody>
</table>

Now we continue working with the Java perspective. Follow these steps:

1. In the Java perspective, select the Packages view, and then expand the JUnit tree.

   By default, Application Developer shows, in the tree, the library and JAR files that are part of the project build path.

2. To change this behavior, open the Packages view menu by clicking the down arrow on the title bar as shown in Figure 4-21.
3. There is a check mark next to Show Referenced Libraries. If you click **Show Referenced Libraries**, the referenced libraries are removed.

When you are programming, there are some places that you frequently go to. You can bookmark these places, so you can access them very quickly. Follow these steps to create a bookmark:

1. Open the Bookmarks view by clicking **Perspective-> Show View-> Other**.
2. Expand the **Basic** tree and select **Bookmarks**. Click **OK** as shown in Figure 4-22.

3. Now you see the bookmark view in the Java perspective. You can add this new view to the same set of views as Navigator by dragging it into the Navigator view and having the correct cursor as shown in Table 4-1. See Figure 4-23.
4. Now we convert the bookmarks view to a fast view. Click the **Bookmarks** title bar and drag it to the perspective toolbar. As you move your cursor over the perspective toolbar, the cursor changes its shape to the stacked folders icon.

Release the mouse button. You see a new icon on the toolbar as shown in Figure 4-24.

Now we can add some bookmarks:

1. Expand the **junit.samples** package in the package view.

2. Open **VectorTest.java** by double-clicking the file in the tree. Application Developer opens the file in the Java editor view.

3. Place the cursor on any line in the file.

4. Right-click the marker bar, which is the gray left band before the line. Select **Add Bookmark** as shown in Figure 4-25.
5. Enter a name in the pop-up window. As an alternative, you can highlight any text in the file and then right-click and select **Add Bookmark**. In this case, the highlighted text is the bookmark name.

6. Click **OK** as shown in Figure 4-26.

A bookmark icon is added to the marker bar as shown in Figure 4-27.

7. Open any other file from the JUnit project.

8. Click the **Bookmark view** icon on the perspective toolbar. This opens the Bookmark view.
9. Right-click the bookmark and select **Go to File** from the pop-up menu, as shown in Figure 4-28. The tool switches to the location of your bookmark.

![Figure 4-28 Going to a bookmark](image)

10. To close the bookmark view, click outside of the view or click the underscore icon on the Bookmark title bar.

If you are used to developing Java applications with VisualAge for Java, you can get a VisualAge for Java behavior for displaying Java code. Follow these steps:

1. Open any Java file and click anywhere in the Java editor view.
2. Click the **Show Source of Selected Element Only** button as shown in Figure 4-29.
3. In the Outline view, select the element (method or variable) that you want to display.

   You see only the selected method or variable in the Java editor view, as with VisualAge for Java.
Now we add the Navigator view to the perspective. In this view, you see all the folders and objects of your application. This view is common for all perspectives. Follow these steps:

1. Click **Perspective-> Show View-> Other** in the main menu.
2. Expand **Basic**.
3. Select **Navigator**.
4. Click **OK** as shown in Figure 4-30.

You see the Navigator view added to the bottom right part of the Application Developer window (Figure 4-31).
You can browse all the folders from this view.

Now we save the Java perspective by following these steps:

1. From the main menu, select Perspective-> Save As.
2. Enter the new name or leave Java. Click OK as shown in Figure 4-32.
3. If the perspective already exists, you are prompted with a confirmation window as shown in Figure 4-33. Click Yes if you want to overwrite it.
4.3 Working with Java code

With Application Developer, you can work with Java code as we explain in the previous topic. In this section, we learn more about some other basic actions on working with Java code. Of course, for working with the Java code, the best perspective is the Java perspective. We show you how you can:

- Add new methods, classes, packages
- Compile Java code
- Run code
- Export Java code

4.3.1 Adding new methods

To add a new method, follow these steps:

1. Open the Java perspective.
2. Expand the JUnit tree in the Packages view.
3. Expand the junit.samples package.
4. Open VectorTest.java by double-clicking the file name in the tree.
5. Scroll to the very bottom of the file in the editor view.
6. Add the following text just before the closing bracket:
   ```
   public void test() {
   ```
7. On the next line, start typing System., and press Ctrl-spacebar to invoke the content assist tool, as shown in Figure 4-34.

8. Select out and press Enter.
9. You can perform the same step to select the `println(String)` method in the `PrintStream` class.

10. Enter `Java with Application Developer` as the parameter for the `println()` method.

11. Enter `;` at the end of the line. The text you enter should look like Example 4-1.

   **Example 4-1 The test method**
   ```java
   public void test()
   {    
       System.out.print("Java with Application Developer");
   }
   ```

12. Save the file by pressing Ctrl-S. You see one error in the Task view, as shown in Figure 4-35.

![Figure 4-35 The unmatched error task](image)

13. Double-click the error in the Task view. Application Developer displays the line in error as shown in Figure 4-35.

14. Click `test()` in the Outline view (lower-left corner of the window).

15. Add a closing bracket for the `test()` method.

16. Save the file. The error is gone.

### 4.3.2 Compiling Java code

As you have noticed, Application Developer automatically performs the build as you save the file. This feature is controlled by the Perform build automatically on resource modification option in the Preferences window.
You can build the project at any time. Follow these steps:

1. Right-click **JUnit** in the Packages view.
2. Select **Build Project** as shown in Figure 4-36. The Rebuild Project option re-compiles all classes in the project.

![Figure 4-36 Building the Java project](image)

### 4.3.3 Running the Java code

Next we run the Java program. Follow these steps:

1. Change the project’s properties:
   a. Right-click **JUnit**. Select **Properties** as shown in Figure 4-37.

![Figure 4-37 Opening the project properties](image)

   b. Select **Launcher**.
   c. From the Run/Debug pull-down menu, select **Java Application** as shown in Figure 4-38.
   d. Click **OK**.
2. Expand the `junit.textui` package.

3. Double-click `TestRunner.java`.

   Notice the running icon next to the class name in the Outline view. This icon indicates that the class contains a `main()` method and can be run as shown in Figure 4-39.
4. Click the **Run** button on the toolbar.

   Application Developer switches to the Debug perspective and reports an error (look for the error message in the Console view).

![Figure 4-40   The debug perspective](image)

5. Switch to the Java perspective.

6. Right-click **TestRunner.java** in the Packages view.

7. Select **Properties**.

8. In the Program Arguments field on the Execution Arguments properties page, type `junit.samples.VectorTest` as shown in Figure 4-41. This argument indicates which class should be tested by TestRunner.java.
9. Click **OK**.

10. Click the **Run** button again.

    Application Developer switches to the Debug view. This time the test runs with no errors.

11. Go back to the Java perspective and change the Execution Arguments property to `junit.samples.AllTests`.

12. Run TestRunner.java again.

Under Console, you see the line you added to the VectorTest class as shown in Figure 4-42.

![Figure 4-41 The Program Arguments window](image)

![Figure 4-42 The TestRunner output](image)
4.3.4 Exporting the Java code

The last step is to export the JUnit project. Follow these steps:

1. Click File-> Export on the menu.
2. Select JAR file and click Next as shown in Figure 4-43.

![Figure 4-43 Exporting a JAR file](image)

3. Select the check box next to the JUnit project.
4. Make sure that the Export generated class files and resources and Export java source files and resources check boxes are selected.
5. Enter c:\temp\junit.jar in the Select the export destination field.
6. Click Finish as shown in Figure 4-44.
4.4 Conclusion

WebSphere Studio Application Developer combines the functionality that was found in VisualAge for Java and the earlier WebSphere Studio product. However, many new features were added. It supports perspectives that allow you to work with your applications from different views.

For example, the J2EE perspective allows you work in an environment customized for building J2EE compliant applications. Application Developer allows you to export your applications directly into J2EE compliant formats, such as enterprise archive (EAR) files and Web archive (WAR) files. You can install these files as enterprise applications in WebSphere Application Server 4.0 without using the WebSphere Application Assembly Tool.

This chapter introduced you to Application Developer. We walked you through some simple examples to show you how to navigate in Application Developer and customize your workspace.
Building Java servlets and JSPs with WebSphere Studio Application Developer

WebSphere Studio Application Developer is an optimized J2EE application development platform. It is evolved from VisualAge for Java Enterprise Edition and WebSphere Studio Advanced Edition. It inherits some of the best features of VisualAge for Java and WebSphere Studio and adds many new features.

Many developers are still working on VisualAge for Java and WebSphere Studio. When they move to Application Developer, an important task is to smoothly migrate their code to this new environment.

In this chapter, we migrate the code developed with VisualAge for Java in Chapter 2, “Servlet and JSP development using VisualAge for Java” on page 21, to Application Developer. We also explain how to build new applications with Application Developer.
5.1 Migrating code from VisualAge for Java

In Chapter 2, “Servlet and JSP development using VisualAge for Java” on page 21, we develop the Java code with VisualAge for Java. We export the code into JAR files that are used by the Application Assembly Tool.

In Chapter 3, “WebSphere V4.0 assembly and deployment tools” on page 75, we use the WebSphere Application Assembly Tool (AAT) to assemble the Java code, JSPs, and images into a WAR file. A WAR file can be deployed on WebSphere Application Server 4.0.

Application Developer can migrate code from either JAR files or WAR files, as shown in Figure 5-1. Since these two file formats have some differences, Application Developer treats them differently.

Since a WAR file has gone through AAT, all the configuration tasks like servlet mappings, initial parameters, adding supporting classes, adding JSP pages, and so on have already been done. In fact, a WAR file is a ready-to-go Web application. When migrating WAR files into Application Developer, we simply import them. All the settings are already there, and the IBM extended configuration files for WebSphere (XML files) are ready. When you develop Web applications for WebSphere Application Server 4.0 with VisualAge for Java and want to migrate code into Application Developer, we recommend that you to generate a WAR file with AAT first. Then, import the WAR file into Application Developer.

If you develop Java applications for non-J2EE compliant servers, like Apache Tomcat or WebSphere 3.5, you may not use AAT. For this situation, just export the Java code into JAR files. Then import it into Application Developer. You have to configure the servlet mappings, parameters, JSPs, and so on in Application Developer.

Tip: The completed applications developed for this chapter are available for download from the ITSO Web site as explained in Appendix A, “Additional material” on page 425.

In this chapter, we import a WAR file named Wsws_Appweb.war into Application Developer. It contains the Java code developed in Chapter 2, “Servlet and JSP development using VisualAge for Java” on page 21.
5.2 Migrating the OrderEntry WAR file

Now we start to work on the OrderEntry application in Application Developer. Since we already wrote the code with VisualAge for Java and assembled it into WAR file with AAT, we can reuse this code in our new development environment.

5.2.1 Importing a WAR file

If you want to keep your application development environments separated in Application Developer, you can create a new workspace to store the application artifacts for each environment. To do this, follow these steps:

1. Create a new directory in the file system. Here we call the workspace directory itsoad. From a command line window, type:

   ```
   md x:\itsoad
   x:\<Application Developer install directory>\wsappdev -data x: \itsoad
   ```

   Here x is the drive letter and <Application Developer install directory> is the path where you installed Application Developer. By default, it is x:\Program Files\IBM\Application Developer.

   It may be worth creating a shortcut on your Windows desktop for launching the command in the future. The default parameters to launch Application Developer used by the start menu icon revert your settings back to the default workspace in the installation directory.

2. Once Application Developer is started, click the down arrow next to the Open The New Wizard button, as shown in Figure 5-2.

   ![Figure 5-2 Creating an enterprise application project](image)

   We select Enterprise Application Project. Another way to do this is to click Other and select J2EE in the left pane and Enterprise Application Project in the right pane of the pop-up window.
3. In the Enterprise Application Project Creation window, as shown in Figure 5-3, we enter Wsws_App as the name of the project. We deselect Application Client project name and EJB project name because we don’t work on EJBs in this chapter. But we select Web project name and then click Finish.

![Figure 5-3 Enterprise application project creation wizard](image)

Once the project is created, Application Developer automatically switches to the J2EE perspective and shows the new project under Enterprise Applications.

4. We import the WAR file into the Wsws_AppWeb project. Click the Navigator tab and select the Wsws_AppWeb project. Click File-> Import, and the Import window appears as shown in Figure 5-4. Select WAR file from the available options and click Next.

![Figure 5-4 Importing a WAR file](image)

5. In the Import Resources from a WAR File window (Figure 5-5), we click the Browse button next to the WAR file field and navigate to the WAR file. We select the Overwrite existing resources without warning check box; otherwise, Application Developer asks if you want
to overwrite one of the XML files it has created for the Wws_App project. Finally, we click Finish.

![Image of Importing the wsws.war file](image)

**Figure 5-5 Importing the wsws.war file**

6. In the Application Developer main window, click the Navigator view and expand Wws_AppWeb-> webApplication-> WEB-INF-> lib.

7. We select Wws_AppWeb_classes.jar, if it exists, right-click it, and select Delete. This JAR file is deleted.

**Note:** Application Developer places all the original class files from the Web application in this JAR file. If we update the Web application and try to test the changes, we don’t see the changes because Application Developer uses the original JAR file for testing.

### 5.2.2 Building the project and modifying the project’s properties

This section explains how to build the project and modify its properties. We follow these steps:

1. We set the application context root. The context root defines the logical root for all of the application’s files. To invoke ItemServlet.html, in the browser, we enter:
   
   http://sysname:port/<context root>/ItemServlet.html

2. We right-click the Wws_AppWeb project in the Navigator view and select Properties.

3. In the Properties window, shown as Figure 5-6, we click Web in the left pane and replace Wws_AppWeb with OrderEntry as the context root. We click OK and then click Yes in the pop-up window asking whether we want to change links based on the new value for the context root.
4. We delete the **Wsws_AppWeb_classes.jar** file and add the context root.

5. We compile the whole project. This may be an optional step if you left the **Perform build automatically on resource modification** option selected. We select **Wsws_AppWeb** in the Navigator view, right-click, and select **Rebuild Project**.

   Application Developer compiles and validates all files within the project. As a result of the build, we see many errors and warnings in the task view (bottom portion of the Application Developer window). If we double-click any error, Application Developer opens the file with the error. All errors are visible in the marker area of the editor window.

6. We can filter some unnecessary warnings. Click the **Filter** button in the task view's toolbar (circled in Figure 5-7).
7. In the Filter Tasks windows, select the options that are shown in Figure 5-8 and click **OK**. The number of items shown in the Task view decreases.

8. Most of the errors indicate that the Java compiler cannot resolve some packages and classes. All of them are part of the IBM Toolbox for Java. We need to add the IBM Toolbox for Java JAR file to the classpath.
We select the **Wsws_AppWeb** project in the Navigator view, right-click, and select **Properties**. In the Properties window, shown as Figure 5-9, we select **Java Build Path** and the tabbed window appears in the right pane. We click the **Libraries** tab and then click the **Add External JARs** button.

![Figure 5-9 Adding external JAR files into the classpath](image)

In the JAR selection window, we navigate to the directory where **jt400.jar** is stored. In our example, we place it into the **c:\jt400** directory. We select the **jt400.jar** file. Click **Open** to select it and then click **OK** in the Properties window.

9. Now we can rebuild the project. If you left the **Perform build automatically on resource modification** option selected, it automatically rebuilds the project. All errors should disappear.

### 5.2.3 Exploring the enterprise application structure in Application Developer

Application Developer follows the J2EE specification in the way it handles enterprise projects. Any EJB jar or WAR file you import into Application Developer has to be part of an enterprise application project. Next, we explore the structure of an enterprise application in Application Developer:

1. We open the J2EE perspective and switch to the J2EE view. This view explores the Application Developer resources from the J2EE specification point of view.

2. We expand the **Enterprise Applications** tree. We see the **Wsws_App** application. Expand this application and all the modules that are part of the application (Web, EJB, or application client) are displayed. In this case, there is only one module – **Wsws_AppWeb.war**.
3. We right-click **Wsws_App** and select **Open With-> Application Editor**. Application Developer opens the application.xml file, which is the deployment descriptor (DD) for this application. This is shown in Figure 5-10.

![Figure 5-10 Deployment descriptor for Wsws_App](image)

4. In the left pane, we expand **Web Modules** and see the **Order Entry Application Module**. This name is defined in AAT. We select this module, right-click and select **Open with-> Web.xml.editor**. Application Developer opens the Deployment Descriptor (DD) for the Web module (WAR file).

5. As shown in Figure 5-11, click the **Servlets** tab on the bottom portion of the Web.xml editor and select **ItemServlet** from the list. The editor displays the Java class that implements this servlet and the servlet's URL mapping.

**Note:** URL mapping is what you enter in the Web browser window as the servlet's name under the context root of the application. It is the same concept as servlet mappings in AAT. Consider this example:

```
hhttp://sysname:port/OrderEntry/ItemServlet
```
6. We click the **Initialization** button. The new pop-up window (Figure 5-12) lets us add or remove the initialization parameters for the servlet. In this case, these parameters include user name, password, URL of the database, and the JDBC driver class name. You can change the parameters to values appropriate for your system and click **OK**.

7. We repeat the steps for ItemPoolServlet to change the initial parameters. We can click the **Source** tab and view the XML source for the parameters we modified with the web.xml editor. If all the changes are correct, we press the Ctrl-S key combination to save the web.xml file. Now the servlets are ready for testing.
5.2.4 Testing the servlets

Now we can test the servlets. We close all the open files in Application Developer and save them if necessary.

**ItemServlet testing**

The ItemServlet application has an HTML page for data input. We need to run it instead of running the servlet itself:

1. In Application Developer, we open the J2EE perspective and select the Navigator view.
2. We expand `Wsws_AppWeb-> webApplication`, right-click `ItemServlet.html`, and select **Run on Server**. A new pop-up Publishing window appears showing the progress of the publishing task. Once it is finished, the WebSphere Test Environment Server automatically starts, and the perspective is switched to the Server perspective, as shown in Figure 5-13.

![Figure 5-13 The server view](image)

3. The embedded browser starts with the input HTML page displayed. We click the **Get Items Information** button in the browser. We see an error screen in the Web browser and the exception stack (Figure 5-14) in the Console view. The important line is highlighted.

```
java.lang.ClassNotFoundException: com.ibm.as400.access.AS400JDBCDriver
    at java.net.URLClassLoader.findClass(URLClassLoader.java:205)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:325)
```

![Figure 5-14 Exception stack in the Console view](image)
4. The WebSphere Test Environment cannot find the JDBC driver class which is part of the
jt400.jar file. We need to add it to the server classpath.

a. We select the Servers view from one of the tabs in the lower portion of the Application
Developer window. We right-click the server instance and select Stop. You can also
click the Stop button on the Server view toolbar. This is shown in Figure 5-15.

b. After this server stops, we expand the Server Instances tree in the Server
Configuration view (lower-left portion of the Application Developer window). We
double-click WebSphere v4.0 Test Environment and select the Paths tab in the editor
window.

c. As shown in Figure 5-16, we click the Add External JARs button and navigate to the
directory where the jt400.jar file is stored. We select it and click Open. We use the
same technique to add the j2ee.jar file as an external JAR file. It is found in the
\Program Files\IBM\Application Developer\plugins\com.ibm.etools.websphere.runtime\lib directory. We close the file in the editor window
and click Yes on the pop-up window to save the changes.
5. Now we can restart the server by clicking the **Start the Server** button on the title bar of the **Server** view. When the server is ready, we see the **Server Default Server open for e-business line in the Console view**.

6. We expand **Wswe-AppWeb-> webApplication**, right-click **ItemServlet.html**, and select **Run on Server**. When the input HTML page appears, we click the **Get Items Information** button. We see the list of items from the iSeries database table as shown in Figure 5-17.

![Figure 5-16  Adding external JAR files into the classpath](image)

![Figure 5-17  ItemServlet result page](image)
**ItemPoolServlet testing**

To test the ItemPoolServlet class, we need to create a DataSource in the WebSphere Test Environment:

1. We open the Server perspective, expand the **Server Configuration** tree in the Server Configuration view, and double-click **WebSphere Administrative Domain**. Application Developer opens the server configuration file in the editor view as shown in Figure 5-18.

![Figure 5-18 WebSphere Administration Domain configuration](image)

2. We select the **Data source** tab in the editor view. We click **Add** next to the JDBC driver list box and enter the parameters as shown in Figure 5-19, where the full name for the Implementation class name field is `com.ibm.as400.access.AS400JDBCConnectionPoolDataSource`. Then we click **OK**.

![Figure 5-19 JDBC driver parameters](image)
3. The IBM Toolbox for Java JDBC driver has been added and highlighted. We click the Add button next to the Data Source defined in the JDBC driver selected above box. We fill in the fields as shown in Figure 5-20 and then click OK.

![Figure 5-20 Data Source parameters](image)

4. Finally, we specify the target system where the database tables are located. We click Add next to the Resource properties defined in the data source selected above box. We fill in the parameters as shown in Figure 5-21. We make sure the system name, in the Value field, is correct and click OK.

![Figure 5-21 Data source property dialog box](image)

5. We use the same technique to add a resource property named libraries. We set the value to the name of the iSeries library that holds the tables, apilib.

6. We save the file by pressing the Ctrl-S key combination. The ItemPoolServlet is ready for testing. We select the Server view by clicking the Server tab in the lower right part of the...
Application Developer window. It informs you that you have made some changes and you have to republish your project (see Figure 5-22).

![Published to server](image)

**Figure 5-22** The Server view

7. We click the **Publish to the server** button and then click **Finish**. We click **OK** in the Publishing pop-up window.

8. Now we can start the server by clicking the **Start the server** button.

9. When the server is ready, we expand `Wsws_AppWeb -> webApplication`, right-click `ItemPool.html`, and select **Run on Server**.

10. We see the input HTML page. Then we click the **Get Items Information** button and retrieve the data.

### Working with HTML files in Application Developer

Since Application Developer is evolved from VisualAge for Java Enterprise Edition and WebSphere Studio Advanced Edition, it inherits the page designing capabilities, which makes it an integrated Web application development environment. Next we work with Page Designer in Application Developer:

1. In Application Developer, we switch to the Web perspective and expand `Wsws_AppWeb -> webApplication`. We double-click `ItemServlet.html`. Then Page Designer is invoked with the HTML page shown in Figure 5-23.
2. In the Page Designer view, we place the cursor just below the line that reads: Press the Button to retrieve the items. We select Insert-> Form and Input Fields-> Submit Button from the menu.

3. In the pop-up window, as shown in Figure 5-24, we enter Submit in the Name box and Get All Items in the Label box.
4. Now we have a Submit button that invokes the ItemServlet servlet and retrieves all the data. We have to pass the part number as a parameter. We use JavaScript to do this.

   a. We click the **Event** button. In the pop-up Edit Event window, we select **OnClick** from the Event list. We click the **Script** button.

   b. The new pop-up Script window allows us to create a custom Java script. We enter the body of the setPart function as shown in Figure 5-25 and click **OK**.

![Figure 5-24 Adding a Submit button](image)
c. In the Script box of the Edit Event window, we enter the following code that calls the `setPart` JavaScript function when we click the **Submit** button:

```
javascript:setPart('**ALL**')
```

d. We click the **Add** button. The Edit Event window looks like the example in Figure 5-26. We click **OK**.

![Edit Event window](image)

**Figure 5-26** The Edit Event window

e. We click **OK** in the Attribute window to add the new button to the page.
f. Since this new button retrieves all items from the database table, we can remove the default value *ALL for the Part Number field. We double-click the Part Number text field (where you see *ALL) in the Page Designer view. Then we delete *ALL from the initial value box and click OK. We press Ctrl-S to save the file. Now the page looks like the example in Figure 5-27.

![Figure 5-27 Modified ItemPool.html page](image)

5. We can test the new HTML page using the same steps as those outlined in “ItemServlet testing” on page 185. To retrieve all the item data, we click the new Get All Items button. We can also retrieve information for a single item using the Get Items Information button. Valid values are in the range 000001 - 000100.

**Exporting an EAR file**

After the modification of the OrderEntry application, we can deploy it on WebSphere Application Server 4.0. Application Developer is capable of generating J2EE standard enterprise applications, so we don’t need to go through the Application Assembly Tool.

1. In Application Developer, we select **File->Export** from the menu.
2. In the pop-up Export window, we select **EAR file** and click **Next**.
3. In the EAR Export window, as shown in Figure 5-28, we select **Wsws_App** from the What resources do you want to export? pull-down list. We enter \x:\temp\Wsws_App.ear in the Where do you want to export resources to? input field. We select the **Export source file** check box and click **Finish**.
Chapter 5. Building Java servlets and JSPs with WebSphere Studio Application Developer

5.3 Developing a new application with Application Developer

In 5.2, “Migrating the OrderEntry WAR file” on page 177, we imported the OrderEntry WAR file into Application Developer. The application was originally developed using VisualAge for Java. This is a convenient way to migrate from VisualAge for Java to Application Developer.

In this section, we use Application Developer to develop a new application. The new application is a Web-based version of the RPG OrderEntry application described in 7.1, “The ABC Company” on page 232.

You may have already noticed that, in the ItemSessionServlet example shown in 2.7.3, “ItemSessionServlet example” on page 64, we hard code all the output pages in the servlet. If you want to change anything about the presentation, such as font size, images and so on, you have to change the code of the servlet. The reason for this is that VisualAge for Java does not have the capability of building JSP pages with What You See Is What You Get (WYSIWYG) tools. We either have to hard code the page in the servlet, or integrate VisualAge for Java with WebSphere Studio to build the JSP pages. This makes the example more complicated.

A popular Web application model is to use a logical three-tier model. The first tier is a presentation tier that typically consists of a graphical user interface shown in a browser. The middle tier is business tier where we build the business logic. The third tier is the data tier that contains the application data.
This architecture takes advantage of each Web component. We use JSPs for the presentation layer. We use servlets and EJBs to control the application flow and to implement the business logic. This is known as *Model-View-Controller architecture*. Figure 5-29 shows the application design.

We build the new OrderEntry application using JSPs, servlets, and JavaBeans. We do not use Enterprise JavaBeans. For an example that uses JSPs, servlets, and EJBs, see Chapter 10, “Building Java applications with Enterprise JavaBeans” on page 315. Since Application Developer has an integrated environment for all these Web components, all the tasks like designing, testing, and exporting can be done within it.

### 5.3.1 New OrderEntry application logic

The application logic is similar with the ItemSessionServlet example in 2.7.3, “ItemSessionServlet example” on page 64, but we use JSPs for presentation of the information. We also add the ability to actually place an order.

When we call the servlet, it uses a JavaBean to retrieve all the available items from the Item table. It passes the result to a JSP page through a session object. The JSP page is responsible for showing the items in a browser. This is shown in Figure 5-30.
Once we select some items, enter numbers in the quantity field, and enter the customer number, the Add to Cart button and Show Cart button are enabled. Clicking these buttons, invokes a controller servlet. It sends the shopping data and customer data to another JSP page for display as shown in Figure 5-31.
We can either click the **Continue Shopping** button to go back to the item JSP page or update the customer information by clicking the related button. Once we are ready to check out, we click the **Check Out** button. This invokes the controller servlet again. The controller servlet calls the JavaBeans to perform order processing and shows the transaction result on a JSP page as shown in Figure 5-32.

![Figure 5-32](image)

**Figure 5-32** Transaction result page

### 5.3.2 Building the application

We build three new JSPs for presentation:

- *StartOrderEntry.jsp* to show all the items you can add to the shopping cart
- *OutputCart.jsp* to show the shopping cart content and customer information
- *Checkout.jsp* to show the transaction result

We build two new servlets for logic control:

- *StartOrderEntryServlet* to create a new session object and retrieve the available items from the Items table
- *OrderEntryControllerServlet* to control all the actions from the JSP pages

We build four new JavaBeans for the business process:

- *Customer* to contain the customer information
- *JDBCPoolCustomer* to retrieve customer information from the Customer table and return the Customer bean
- *OrderDetail* to contain the order detail information
- *OrderProcess* to handle the order placement process

We reuse some existing beans in the access, shopData, and support packages:

- *Items*, which contains the item detail
- *CartItem*, which contains the item detail in the shopping cart
ShoppingCart, which represents shopping cart data
> JDBCPoolCatalog, which retrieves all items from the database

**Note:** The Customer and OrderDetail beans already exist in the Support package. Since our new Customer and OrderDetail beans are different, we create new beans instead of using the old ones. We create them in the access package.

The architecture for our new application is shown in Figure 5-33. The StartOrderEntryServlet is used to retrieve item data and pass it to StartOrderEntry.jsp. All the action control is done by the OrderEntryControllerServlet, which passes different data beans back and forth through a session object.

![Figure 5-33  New shopping cart application architecture](image.png)

The new project is based on the project we created in 5.2, “Migrating the OrderEntry WAR file” on page 177. We add the JavaBeans, JSPs, and servlets to the project.

### 5.3.3 Building the OrderEntry application with Application Developer

We switch to the Java perspective. By default, Application Developer shows the library and JAR files that are part of the project build path. To change this behavior, we open the **Packages** view menu by clicking the down arrow on the title bar. As shown in Figure 5-34, we deselect **Show Referenced Libraries**.
Customer bean

We follow these steps:

1. We add the JavaBeans to the project. We expand the source folder in the left pane, right-click the access package, and select New-> Class.

2. In the Java Class window, shown in Figure 5-35, we enter Customer for Name, remove java.lang.Object from Superclass, and click the Add button next to the Extended interfaces field.

3. In the Extended Interfaces Selection window, shown in Figure 5-36, we select Serializable as the interface name and java.io as the package name. We click OK and the interface is added into the Extended Interface input field. We click Finish.
4. The Customer object is created in the access package and the source is opened automatically. In the source window, we add the following line of code just below package access:

```java
import java.math.*;
```

5. We define the private variables that represent customer information. Example 5-1 shows the private variables that we add.

**Example 5-1 Variable definition for Customer class**

```java
private String id;
private String lastName;
private String firstName;
private String init;
private String address1;
private String address2;
private String city;
private String state;
private String postCode;
private String phonenumber;
private String wid;
private String did;
```

6. For all the variables defined, we want to define setter and getter methods. Application Developer provides a function to do this. In the Outline window shown in Figure 5-37, we right-click in each field and select **Generate Getter and Setter**.
7. Setter and getter methods are created for each private variable. We need to modify one getter method, `getDid()`. The `Did` variable is defined as numeric in the database. When we call `getDid()` from other classes, we want an `int` returned. The method is changed as shown in Example 5-2.

**Example 5-2  getDid() method**

```java
public int getDid() {
    return Integer.parseInt(did);
}
```

8. We also need to build a constructor for this class. We pass the constructor a String array. It interprets the array and moves the data to the corresponding private variables. Example 5-3 shows the constructor code.

**Example 5-3  Constructor for Customer class**

```java
public Customer(String[] custInfo) {
    id = custInfo[0];
    lastName = custInfo[1];
    firstName = custInfo[2];
    init = custInfo[3];
    address1 = custInfo[4];
    address2 = custInfo[5];
    city = custInfo[6];
    state = custInfo[7];
    postCode = custInfo[8];
    phonenumber = custInfo[9];
    wid = custInfo[10];
    did = custInfo[11];
}
```
OrderDetail bean
We use the same technique to build the OrderDetail class. Since the steps are very similar, we do not go through it in detail. Example 5-4 shows the code for the OrderDetail class.

Example 5-4  OrderDetail class

```java
package access;
import java.math.*;
import java.io.*;

public class OrderDetail implements Serializable {
    String itemID = null;
    float itemAmount = 0;
    int itemQty = 0;
    int lineNumber = 0;
    public OrderDetail () {
        super();
    }
    public OrderDetail (String itemID, float itemAmount, int itemQty) {
        this.itemID = itemID;
        this.itemAmount = itemAmount;
        this.itemQty = itemQty;
    }
    public float getItemAmount() { return itemAmount; }
    public String getItemID() { return itemID; }
    public int getItemQty() { return itemQty; }
    public int getLineNumber() { return lineNumber; }
    public void setLineNumber(int ln) { lineNumber = ln; }
}
```

The Customer and OrderDetail beans represent customer and order detail information. They are passed back and forth by the servlets and JSPs. Once you select items and click the Check Out button, the OrderDetail bean is filled in by the controller servlet. For the customer bean, customer information is retrieved from the Customer table. We build an access bean, JDBCPoolCustomer, which connects to the database table and retrieves the data for the Customer bean.

JDBCPoolCustomer class
For the JDBCPoolCustomer class, we use JDBCatalgSupport as the super class because we want to reuse some of its functions. We add three methods to the class:

- `connectToDB(datasource, userid, password)` looks up the data source name using JNDI APIs.
- `getCustomerByID(id)` retrieves Customer data from the Customer table using a Customer ID key and returns a Customer bean.
- `setCustomer(String[])` updates the Customer table with new customer information.

The `connectToDB` method functions are similar to the functions we described in 2.5.1, "DataSource version" on page 41. The `getCustomerByID(id)` and `setCustomer(String[])` methods use the data source retrieved in the `connectToDB()` method to connect to the database table. Example 5-5 shows part of the code of the `getCustomerByID(id)` method.
Example 5-5  getCustomerbyID() method

```java
try {
    dataConn1 = getConnection(v3DS, connUserId, connUserPassword);
} catch (Exception e) {
    System.out.println("JDBCPoolCustomer: "+ e.getMessage());
    e.printStackTrace();
}
try {
    psSingleRecord = dataConn1.prepareStatement("SELECT CLAST, CFIRST, CINIT,
    CADDR1, CADDR2, CCITY, CSTATE, CZIP, CPHONE, CWID, CDID  FROM CSTMR WHERE CID = ?");
    psSingleRecord.setString(1, customerID);
    aResultSet = psSingleRecord.executeQuery();
    if (aResultSet.next()){
        String[] acustomerData = new String[12];
        acustomerData[0]=customerID;
        for (int i=1; i<=11;i++)
            acustomerData[i]=aResultSet.getString(i);
        customer=new access.Customer(acustomerData);
    }
    dataConn1.close();
} catch (SQLException ex) {
    ex.printStackTrace();
}

return customer;
```

Now, the Customer, OrderDetail, Item beans, JDBCPoolCustomer, and JDBCPoolCatalog access classes are complete. We are ready to test them. Since we haven't created the servlets and JSPs to invoke these classes yet, we use the Application Developer Scrapbook to test the Customer bean.

**Testing the Customer bean in the scrapbook**

We click the **Create a Scrapbook Page** icon as shown in Figure 5-38. In the pop-up window, we select the access package, enter mytest as the File name, and click **Finish**.
A new scrapbook page, named *mytest.jspage*, is created in the access package. We double-click this file. In the source code area, we enter the code shown in Figure 5-39.

```java
String[] custinfo=new String[12];
custinfo[0]="0001";
custinfo[1]="Liang";
custinfo[2]="Cliff";
custinfo[3]=" ";
custinfo[4]="15010 Yonge Street";
custinfo[5]=" ";
custinfo[6]="Aurora";
custinfo[7]="ON";
custinfo[8]="L4G 1M6";
custinfo[9]="905-727-2384";
custinfo[10]="0001";
access.Customer customer=new access.Customer(custinfo);

System.out.println(customer.getId() + " " + customer.getFirstName() + " " + customer.getInit() + " " + customer.getLastName());
```

We select all the code in the scrapbook and click the **Run the Selected Code** button as shown in Figure 5-40. The result is displayed on the console.
OrderProcess class
The OrderProcess class is very similar to the JDBCPOOLCustomer class. This class encapsulates the order placement process. The logic to place an order is to:

1. Access the District table to retrieve the next order number.
2. Increase the next order number by one and write it back to the District table.
3. Retrieve the shopping cart from the session object. For each item in the cart, retrieve the item ID, item quantity, and item amount. Deduct the quantity from the quantity in stock field of the Stock table.
4. Insert a new record into the Order Line table with the order details.
5. Insert a order summary into the Orders table.
6. Update the customer balance in the Customer table.

This logic is placed in the placeOrder() method. We do not go through this in detail.

StartOrderEntryServlet servlet
The StartOrderEntryServlet servlet is the entry point of this application. It accepts a request from the browser, retrieves all the items from the Item table, creates a new session object, and puts the list of items in it. Finally, it forwards the request to the StartOrderEntry JSP. After this, all the flow control is done by another servlet, OrderEntryControllerServlet.

1. In Application Developer, we switch to the Web perspective.
2. We right-click the nservlets package and select New-> Servlet.
3. In the Create the Servlet Class window, shown in Figure 5-41, we enter StartOrderEntryServlet as the servlet Name. We replace javax.servlet.http.HttpServlet with SuperServlet in the Superclass field because we want to use some functions defined in nservlets.SuperClass. We select the init(), doPost(), and doGet() methods in the Which method stubs would you like to create? option and click Next.
4. The next window, Define the Servlet in the Deployment Descriptor File, allows you to define the servlet initial parameter settings (see Figure 5-42). We select the Add to web.xml option and leave the values in Display name and Mappings fields untouched. We click the Add button next to Init Parameters and add the name/value pairs listed in Table 5-1.

Table 5-1  StartOrderEntryServlet initialization parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>A valid iSeries user ID</td>
</tr>
<tr>
<td>password</td>
<td>A valid iSeries password</td>
</tr>
<tr>
<td>datasource</td>
<td>jdbc/NativeDS2</td>
</tr>
</tbody>
</table>

The StartOrderEntryServlet servlet is created in the nservlets package and the source code window is automatically opened.
5. We add the following import statements to the servlet:

```java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
```

6. In the `init()` method of this servlet, we instantiate a JDBCPoolCatalog object that is used to connect to the database. We retrieve the initial parameters and invoke the `connectToDB()` method, which looks up the JDBC data source. Example 5-6 shows part of the code of the `init()` method.

### Example 5-6  init() method of StartOrderEntryServlet

```java
super.init(config, SuperServlet.SYSTEM);
aJDBCPoolCatalog = new access.JDBCPoolCatalog();
String datasource = getInitParameter("datasource");
String userid = getInitParameter("userid");
String password = getInitParameter("password");
flexLog("DataSource: " + datasource);
flexLog("UserID: " + userid);
flexLog("Password: " + password);
if ((datasource == null) || (userid == null) || (password == null)) {
    flexLog("StartOrderEntryServlet: ......");
    throw new ServletException("Could not retrieve ....");
}
String returnValue = aJDBCPoolCatalog.connectToDB(datasource, userid, password);
if (!returnValue.equals("connection successful")) {
    throw (new ServletException("Unable to connect to DB"));
}
```
7. In the `doGet()` method, we pass control to the `doPost` method. This way we only need to add logic to the `doPost()` method.

8. In the `doPost()` method, we create the session object and invoke the `getAllV()` method of `JDBCPoolCatalog` to retrieve the items from database. We put the data into the session object and forward the request to the StartOrderEntry JSP. Example 5-7 shows part of the code of the `doPost()` method.

---

**Example 5-7  doPost() method of StartOrderEntryServlet**

```java
HttpSession session = request.getSession(false);
if (session != null)
    session.invalidate();
session = request.getSession(true);
response.setContentType("text/html");
response.setHeader("Pragma", "no-cache");
PrintWriter out = response.getWriter();
if (null == session) {... }
try {
    Vector items = null;
    try {
        items = aJDBCPoolCatalog.getAllV();
    } catch (Exception e) {
        System.out.println("...... ");
    }
    session.setAttribute("sessionlist.items", items);
    response.sendRedirect(response.encodeRedirectURL("StartOrderEntry.jsp"));
}
```

---

**StartOrderEntry.jsp**

Since the `StartOrderEntryServlet` forwards the request to the StartOrderEntry JSP, we need to build the JSP page:

1. Still in the Web perspective, we right-click `webApplication` in the left pane and select New-> JSP File. In the pop-up window, we enter `StartOrderEntry` for File Name and click Finish. The `StartOrderEntry.jsp` file is created under the `webApplication` folder.

2. We do not go through how to design JSP pages in this section. We just simply click the Source tab for the JSP and input the code there. The JSP code checks the session object and picks up the item data from the session. It then creates a table to show all the data in the browser. It displays an input field for customer ID and three buttons for the tasks of adding items to the shopping cart, showing the shopping cart and resetting the values. Each action we invoke in the JSP page is routed to the `OrderEntryControllerServlet` servlet. Example 5-8 shows some of the code of the StartOrderEntry JSP.

---

**Example 5-8  Code of StartOrderEntry.jsp**

```html
<%@ page import="java.util.* " %>
<%
    HttpSession mySession = request.getSession(false);
    Vector itemVector = (Vector)(mySession.getAttribute("sessionlist.items"));
```
String custID = (String)(mySession.getAttribute("customerID"));

<% if (itemVector != null) { %> 
<% for (int i = 0; i < itemVector.size(); i ++) { %>
<% String[] aPart = ((String[]) itemVector.elementAt(i)); %>
<tr>
<td width="8%" valign="middle" align="center"><CENTER><INPUT TYPE=checkbox name=index value="<%= i %>"></CENTER></td>
</tr>
<% } %>
</table><br>
...</form method="POST" name="orderEntryForm" action="/OrderEntry/OrderEntryControllerServlet">
<INPUT TYPE="hidden" NAME="action" VALUE="default">
</CENTER>
</table><br>
...

3. Now we can run StartOrderEntryServlet in Application Developer. We right-click the
servlet and select Run On Server. Application Developer automatically starts the
WebSphere 4.0 Test server and shows the result in the embedded browser. Figure 5-43
shows the output of the JSP.

Figure 5-43  Running StartOrderEntryServlet in the WebSphere Test Environment
OrderEntryControllerServlet servlet

The OrderEntryControllerServlet is a very important part of our application. It acts as the gateway and control center. All the requests from the JSPs come into this servlet. It decides which function should be called. All the data coming from the access classes is sent to this servlet, which forwards the data to the JSPs through the session object.

Creating OrderEntryControllerServlet is similar with creating StartOrderEntryServlet. It also needs initial parameters such as userid, password, and data source. It extends SuperServlet instead of extending HttpServlet directly.

The action control part is in the doPost() method. It calls different functions based on the action value returned from the JSP page. Example 5-9 shows part of the code of the doPost() method.

Example 5-9  doPost() method for OrderEntryControllerServlet

```java
if (action.equals("AddToCart")) {
    addToCart(session, request, response, customerID);
} else
    if (action.equals("ShowCart")) {
       displayCart(customerID, response, session);
    } else
        if (action.equals("CheckOut")) {
            placeOrder(session, response);
            response.sendRedirect(response.encodeRedirectURL("Checkout.jsp"));
        } else
            if (action.equals("UpdateCustomerInformation")) {
                session.setAttribute("custUpdateFlag", "false");
                updateCustomer(request, response, session);
            } else
                if (action.equals("ContinueShopping")) {
                    response.sendRedirect(response.encodeRedirectURL("StartOrderEntry.jsp"));
                } else
                    if (action.equals("ShopSomeMore")) {
                        Vector itemVector = (Vector) (session.getAttribute("sessionlist.items"));
                        String custID = (String) (session.getAttribute("customerID"));
                        if (session != null)
                            session.invalidate();
                        session = request.getSession(true);
                        session.setAttribute("sessionlist.items", itemVector);
                        session.setAttribute("customerID", custID);
                        response.sendRedirect(response.encodeRedirectURL("StartOrderEntry.jsp"));
                    }
```  

If we click the Add to Cart button or the Show Cart button in the StartOrderEntry JSP, the controller servlet calls the addToCart() method and the displayCart() methods respectively. We add these methods.

The addToCart() method is used to add the items you select into the shopping cart and to put the shopping cart into the session object. It first retrieves the selected items from the session object. It moves them into the CartItem bean and puts it into the session object. Example 5-10 shows the key part of the addToCart() method.
Example 5-10  Code of addToCart() method

Vector parts = (Vector) session.getAttribute("sessionlist.items");
String[] value = request.getParameterValues("index");
String[] quantity = request.getParameterValues("quantity");
if (value != null) {
    int j = 0;
    for (int i = 0; i < value.length; i++) {
        j = Integer.parseInt(value[i]);
        String[] data = (String[]) parts.elementAt(j);
        Integer qty = new Integer (quantity[j]);
        CartItem aCartItem = new CartItem(data[0], data[1], data[2], data[3], qty);
        cart.getItems().addElement(aCartItem);
    }
    session.setAttribute("shopcart.selected", cart);
    displayCart(customerID, response, session);
}

The displayCart() method is used to assemble the shopping cart data and customer information together and forward them to a JSP page. Recall that in the StartOrderEntry page, you need to input customer ID. The customer detail is retrieved based on this ID. Since the shopping cart is already in the session object, we only need to retrieve customer data from the database. The method then puts the customer information into the session object and forwards the request to OutputCart JSP. This JSP page is responsible for retrieving data from the session object and presenting it in the browser. Example 5-11 shows some of the code in the displayCart() method.

Example 5-11  Code in displayCart() method

aJDBCPoolCustomer = new access.JDBCPoolCustomer();
access.Customer customer=null;
String returnValue = aJDBCPoolCustomer.connectToDB(..);
.....
customer = aJDBCPoolCustomer.getCustomerByID(newCustID);
session.setAttribute("customerObj", customer);
session.setAttribute("cFirstName", customer.getFirstName());
session.setAttribute("cMiddleInitials", customer.getInit());
session.setAttribute("cLastName", customer.getLastName());
session.setAttribute("cAddressLine1", customer.getAddress1());
session.setAttribute("cAddressLine2", customer.getAddress2());
session.setAttribute("cCity", customer.getCity());
session.setAttribute("cState", customer.getState());
session.setAttribute("cZip", customer.getPostCode());
session.setAttribute("cPhone", customer.getPhonenumber());
    wid=customer.getWid();
    did=customer.getDid();
try {
    response.sendRedirect(response.encodeRedirectURL("OutputCart.jsp"));
}
catch (Exception ex){
    System.out.println(ex.getMessage());
}

There are two other key methods – placeOrder() and updateCustomer(). The placeOrder() method invokes the function to create an order transaction. The updateCustomer() method calls the setCustomer() method of JDBCPoolCustomer to update the customer information.
Now the OrderEntryControllerServlet servlet is complete. We add OutputCart.jsp and Checkout.jsp into the Web application using the same technique as in “StartOrderEntry.jsp” on page 209. We do not go through these JSP pages in detail. They are basically used to present the results of the shopping cart, customer information, and transactions.

**Exporting to an EAR file**

After completing the OrderEntry application, we can deploy it on WebSphere Application Server 4.0. Application Developer is capable of generating J2EE standard enterprise applications, so we don’t need to go through the Application Assembly Tool. The steps are exactly same as “Exporting an EAR file” on page 194.

### 5.4 Conclusion

In this chapter, we used Application Developer to build J2EE compliant applications. We migrated the examples created in Chapter 2, “Servlet and JSP development using VisualAge for Java” on page 21, to Application Developer. We also created a new application using Application Developer. Application Developer allows you to export directly to war and ear formats. You do not need to use the Application Assembly Tool.
Chapter 6. Introduction to Enterprise JavaBeans

JavaBeans have become widely accepted in the Java programming community as the way to create the client side for applications. JavaBeans allow application programmers to create reusable components and build applications with those components. The flexibility and ease-of-use associated with the component model, and the tools that use JavaBeans, have helped application developers greatly. Extending this concept to the server side, business components add considerable complexity. Developers must address the following items:

- Persisting the component data in a data source
- Distributing the components across a network
- Managing transactions
- Building the necessary security required by business applications

The Enterprise JavaBeans (EJB) technology provides a logical extension to the JavaBeans concept. It is targeted at server-tier business logic development. It provides interfaces that insulate the programmer from the complexities and dependencies that are unique to a platform.

EJB technology is critical for the development of robust, Java-based business applications on the iSeries server. This chapter discusses the basics of EJB technology, what it offers application developers, its key features, and how EJB technology fits into the iSeries environment.
6.1 The Enterprise JavaBeans specification

Business applications that support a wide variety of users within or across business domains are usually very complex. The complexity of the domain requires application developers to understand the underlying domain technology and the business domain sufficiently to design and implement applications. Advancements in technology have increased the capabilities that can be integrated into business applications. At the same time, the new technology has increased the complexity of application development.

The EJB specification was defined by Sun Microsystems Inc. to address the issues concerned with developing applications. The mission of the EJB specification is to define an architecture for application development, which accomplishes the following goals:

- **Make it easier to write business applications**: By separating the business application from system services, the application developer can focus on the needs of the business application. This includes the business logic and the representation of business data.

- **Increase manageability of the systems environment**: By keeping the business logic and business data in a server environment, the application code is centrally managed. Administration of the application environment is simplified, which allows easier distribution of enhancements, fixes, and extensions to the business application.

- **Promote reuse**: By defining business objects as components, an application developer can create new components, reuse existing components, or purchase components. By using existing components or purchasing components, the application developer assembles the application rather than developing and testing new code.

Version 1.1 of the EJB specification defines the fundamental mission and concepts for creating enterprise server applications. Subsequent versions of the EJB specification will expand and fine tune the specification to define all aspects of the EJB environment.

6.2 Enterprise JavaBeans architecture definition

The Enterprise JavaBeans architecture defines the fundamental concepts for Java server-based business applications. The architecture defined in the EJB specification, version 1.1, is available from Sun Microsystems on the Internet at:

http://www.java.sun.com/products/ejb/

The EJB architecture is a component-based architecture for the development and deployment of server-based business applications. It allows the separation of business applications from the underlying system services. The following list breaks these statements down to help unravel the definition of the EJB architecture:

- **Component based**: The EJB architecture uses the concept of components to represent business objects, business logic, or business tasks within the application.

- **Architecture**: The EJB architecture provides the structure for developing Java-based, object-oriented business applications. It is not a tool for building these business applications. However, it is an architecture for defining the application components.

- **Development and deployment**: The EJB architecture is Java-based, which allows many existing Integrated Development Environments (IDE) to support creating and testing of the components. The application developer selects from a variety of tools that support the EJB specification and provides the services needed by the application. The details of deployment are separated from the business components. Therefore, deployment considerations are customized to the needs of the business environment without affecting the business components.
Server-based business application: In recent years, there has been a movement toward client/server computing-based environments, with increasing portions of the application residing on the client. The server is given the role of a data repository. By returning the application to the server, the ability to manage the application is simplified. The EJB specification allows you to distribute the application components across multiple servers to meet scalability and business structure needs.

Separation of the business application from the underlying system services: With the EJB architecture, the application server environment manages and supports the system services. This removes the requirement for the business application to deal with low-level application programming interfaces (APIs) to perform its functions. However, these low-level APIs are still available, and the application can use them.

Since the beginning of Java, EJB architecture has mainly been used for client-side processing and Internet-based applications. It is being accepted as the standard for building object-oriented business applications in Java. Many server vendors are using this specification to define server environments, which is making the EJB specification a reality. Many businesses are now using Java and the EJB technology for building and deploying business applications on servers.

6.3 How EJB technology fits into the Java architecture

EJB technology complements and extends the existing Java architecture. This technology extends the promise of Java by providing an application server environment that handles system services that would otherwise be handled by the application code. This allows EJB technology to extend the ease-of-use of Java by allowing the application developer to focus on business applications and let the application server focus on system services.

The JavaBeans and Enterprise JavaBeans technology may seem to be on a collision course, but they actually complement each other. JavaBeans technology defines a component model with conventions that enable development tools to examine a JavaBean’s components to determine its capabilities. The development tools expose the properties and capabilities of a JavaBean to application developers. The application developer sets bean properties or connects the beans methods to other JavaBeans. Many JavaBeans components that exist today are used in graphical interface development.

EJB technology does not implement the same interfaces as JavaBean technology, nor does it follow the JavaBean event model. This technology has its own component model with distinct interfaces and conventions. It supports representing business logic and business data within an application environment. EJB components are intended for the server side of business applications, while JavaBeans components fit nicely with the client-side of the application. To be complete, the EJB component model includes an application server environment that provides the necessary services to secure the application, provide transactions, and persist business data.

EJB technology allows business applications to take advantage of the Internet, intranet, and extranet by making it easier to extend existing business applications. EJB technology provides extensions to the Java language that are needed before business application developers take Java seriously.
6.4 Why EJB is important

EJB technology is important to both application providers and the businesses that use their applications. The Java language allows application providers to develop applications that are platform independent. This is valuable to application users because it allows them to select the software they need, independent of the hardware platform, that fits their business environment. If a user has a mixture of hardware, they can still run the same application in these different environments. At the same time, the application developer does not need multiple versions of the same application, with each version tuned to the specific platform.

This allows the application developers to focus on the business problem they are trying to solve and let the EJB server handle system services. EJB technology allows the developer to define the deployment environment at installation time, rather than when the application is designed. This allows the developer to customize the application to the business domain, instead of to both the business domain and the specific hardware platform system.

Keeping the business logic and data on the server ensures that every user is working with the same master data and the same business logic. This also provides the application user better protection of intellectual assets that have been incorporated within the application. Servers, especially the iSeries server, have better security packages than many client systems. When a business is connected to the Internet, ensuring that corporate data and intelligence is safe from unauthorized access is important.

The Java language, as the application development language of choice, is the stated direction of IBM. The IBM Server division is moving forward to make Java its language of choice. Many computer industry leaders have committed to making Java the language of electronic business, the Internet, and business applications. By moving to Java and EJB technology, application providers position themselves for the future.

6.5 Leveraging Java and EJB technology

Java increases the productivity of developers who write applications and simplifies maintenance. The very nature of object-oriented technologies provides the basis for this statement. As a language, Java has taken the best of other object-oriented languages and applied various safeguards to minimize the problems programmers have experienced in the past. EJB technology makes developing applications with Java measurably easier. It hides all the system services details that are present when dealing with Java directly.

The EJB component model defines the infrastructure for creating an environment that separates system services from application business logic. The EJB server and EJB container deal with all the system service infrastructure included in the following list:

- Transactions
- Persistence
- Resources
- Security

This allows applications access to all of these services without dealing with their complexity. The application development activity is made easier, so the developer can focus on business logic application development.

The EJB component model represents business logic and business data as components. Business data is reused within the same domain in various aspects of the business. For example, assume the customer information is used in the following ways:
In an OrderEntry application when a customer places or updates an order
In accounting for creating invoices
In marketing for target advertising campaigns

By creating the customer as a component, the applications that support these different organizations use the same customer component, which minimizes development and testing. As the development team identifies and creates a repository of business object components, they begin to assemble applications to satisfy different end-user requirements.

EJB technology also leverages existing applications currently running business environments. It allows the extension of existing applications to provide new and additional functions for the business. The EJB specification identifies Common Object Request Broker Architecture (CORBA) as the means to provide inter-operability between different programming languages. This allows applications written in different languages, possibly running on different platforms, to interact with each other.

Even without CORBA, EJB objects can be used to wrapper existing application logic. For example, you can wrapper Component Object Model (COM) objects as EJB components and use them as part of a Java application.

### 6.6 EJB Architectural Overview

The EJB specification consists of these two major units:

- **Components**
  - Entity beans
  - Session beans

- **Services**
  - EJB container
  - EJB server

Figure 6-1 shows the relationship between entity beans and session beans.

![EJB Architecture Overview](image)

**Figure 6-1 EJB Architecture Overview**
Entity and session beans are explained here:

- **Entity beans**: These components represent business objects and contain business data. Because an entity bean contains business data, its contents are persisted for later use. Entity beans often reflect a row within an application database. An entity bean has methods to manage its data (get and set methods) and can support business logic pertaining to its business data.

- **Session beans**: These objects perform business processes or tasks within a business process. A client uses a session bean to complete a particular task. Session beans are transient (their data is not persisted) and only exist for the life of the transaction. Session beans usually perform such activities as obtaining or storing business data by using the entity beans or performing business logic that is maintained separately from the business data.

6.6.1 The EJB server

In the EJB architecture, system services can be broken down into two distinct areas, which are often linked: the enterprise bean server and the enterprise bean container.

Figure 6-2 shows the relationship between the EJB container and the EJB server.

![EJB system services providers](image)

**EJB container provider**
- Moves data between EJB and database (or application)
- Protects EJB provider from having to know EJB server interfaces
- Uses EJB server system services

**EJB server provider**
- System services such as transactions, security
- May provide a container to persist data to a specific data source
- May publish interfaces, so others can provide containers

Figure 6-2  EJB system services providers

**EJB container**

EJB containers serve as the means to insulate the enterprise bean developer from the specifics of the EJB server services such as transaction management, security, and object distribution. It provides a simple interface for the enterprise bean and accesses the system services for it. This interface is referred to as the *component contract* for the Enterprise JavaBean.

The container is defined as a separate mechanism from the EJB server. The clarity of this separation is at the discretion of the tool vendor. The container manages Enterprise JavaBean objects. It manages the life cycle of the object (creation, maintenance, and deletion), implements the security for the object, and coordinates distributed transactions involving the object. By performing these activities, the container eliminates the need for bean developers to concern themselves with these issues.
**EJB server**

The EJB server allows the application developer to obtain the system services required by the application without directly dealing with lower level APIs. Figure 6-3 shows the architecture of the EJB server.

![EJB server diagram](image)

**Figure 6-3  The EJB server**

The EJB server carries the majority of the burden of dealing with the system environment by managing and coordinating the allocation of resources to the application. The following list describes the key system services (APIs) that the application server supports:

- **Security**: In the EJB component model, security is both granular and flexible. It is granular because it is configured at the component level or at the method level. It is flexible because it is configured outside the application code by using utilities provided by the application server provider.

- **Transactions**: In the EJB component model, transactions take two basic forms, which are explained in the following list. It is important to notice that the EJB specification does not specifically distinguish between these two forms. This breakdown is to simplify the explanation.
  - **Database transactions**: Database transactions are granular and are configured by settings in the component. Database transactions reflect state changes in the component that are reflected in the database.
  - **Business transactions**: Business transactions represent business processes or tasks that can involve a variety of components and business logic. Business transactions often encapsulate changes to many of the components involved in the transaction.

- **Persistence**: The EJB architecture eliminates the need for the components to deal with the persistence mechanism. Using JDBC as the database management technology, the application server interacts with a wide variety of data stores on the market today.

The EJB server supports these capabilities and manages many of the resources that are common within a Java application. These resources include thread pools and the caching of objects. It is important to notice that many of the low-level APIs associated with Java programming are hidden from the application developer.
The EJB server provider is typically a company that produces middleware or a company that produces operating systems, such as IBM. Currently, there is no standard for the interface between a container and an EJB server. This may be introduced in a later release of the EJB specification. The EJB server provider often provides a container as well. EJB server support is currently available on the iSeries server with WebSphere Application Server Advanced Edition, BEA WebLogic Application Server, BlueStone Software’s Sapphire/Web, and Novera’s jBusiness.

6.6.2 Types of components

The components (entity beans and session beans) break down into more discrete groupings:

**Note:** Entity beans are either container managed or bean managed. Session beans are stateful or stateless.

- **Container-managed entity beans:** Delegate the reading and writing of bean attributes to the persistent datastore to the container that holds them. This allows the bean provider to set up a mapping schema from the bean attributes to the database columns outside of the entity bean itself. It also allows the entity bean developer to keep the persistence details separate from the business object (entity bean). Greater flexibility is achieved because the persistence information may be modified without affecting the entity bean. This makes the bean more reusable and portable.

- **Bean-managed entity beans:** Support the situation where the bean developer needs to have more control over persisting a bean. Bean-managed entity beans allow the bean provider to control the reading and writing of the bean attributes to the database. This gives the bean provider greater flexibility in providing their own persistence strategy. This can include persisting to a variety of databases or file types or providing nested database transactions. All the code necessary to map the beans attributes to the database are part of the bean itself.

Session beans are transient objects and perform operations on behalf of the client, such as accessing the database through the entity bean or performing business logic. Session beans can be involved in a transaction. However, they may not be recoverable in the case of a system crash. A session bean is stateless or stateful:

- **Stateless session beans:** Perform activities for the client, but do not maintain any data. They can perform business logic and calculations. However, no instance variables are defined within the bean.

- **Stateful session beans:** Have data and maintain that data for the life of a transaction. If this data must be persisted, it must be forwarded to entity beans. Stateful session beans minimize the amount of interaction between the client and the server, making the application more efficient. However, it is imperative that any data that needs preserving is captured and forwarded.

Entity beans and session beans provide the heart of the business application by representing both the business objects (data) and the business logic (processes and tasks).

6.6.3 Component content

A component consists of interfaces, the class, any required utility classes, and a deployment descriptor. Previous discussion of the components focused on the class itself. However, in a distributed environment, a component is accessed through its interfaces. As defined in the following list, each component has two interfaces:
- **Home interface**: Contains the methods for creating, deleting, and locating (finding) a particular instance of a bean.

- **Remote interface**: Contains the business methods that may be performed on a bean. The client interacts with the remote interface for the entity or session bean.

One of the key advantages of using EJBs is how easy it is to customize a bean component. The deployment descriptor provides the means to make this possible. Figure 6-4 shows how deployment descriptors are used.

![Diagram](image)

**Figure 6-4  EJB deployment descriptors**

Each Enterprise JavaBean class requires a *deployment descriptor*. The deployment descriptors are used to establish the runtime service settings for an enterprise bean. These settings tell the EJB container how to manage and control the enterprise bean. The settings can be set at application assembly or application deployment time.

The deployment descriptor specifies how to create and maintain an Enterprise Bean object. It defines, among other things, the enterprise bean class name, the JNDI name space that represents the container, the Home interface name, the Remote interface name, and the Environment Properties object name. It also specifies transaction semantics and security rules that should be applied to the Enterprise Bean. These settings are described in the following list:

- Bean JNDI Name
- Enterprise Bean Class Name
- Home Interface Class Name
- Remote Interface Class Name
- Environment Properties
- Reentrancy
- Control Descriptor (per bean or per method)
  - Run As Mode (Client, System, Specified Identity)
  - Isolation Level (Read committed, Read uncommitted, Repeatable read, Serializable)
  - Transaction Attribute (Bean Managed, Mandatory, Not Supported, Required, Requires New, Supports)
Access Control Entry (per bean or per method)
- Method name
- Identities

(Entity Only)
- Container Managed Fields
- Primary Key Class Name

(Session Only)
- Session Time-out
- State Management Type (Stateful or Stateless)

These are the key settings found within the deployment descriptor. They provide flexible use of a component by customizing the component at runtime.

6.7 EJB roles

The EJB specification identifies various application development and deployment roles. Both tool vendors and application providers have a variety of opportunities to take advantage of EJB technology and play an active part in providing applications using the technology. The EJB specification in the following list identifies these major roles:

- Enterprise JavaBean provider
- Application assembler
- Application deployer
- Server provider
- Container provider

6.7.1 Enterprise JavaBean provider

As shown in Figure 6-5, the Enterprise JavaBean provider provides the components for building business applications. Domain expertise is a critical characteristic of these providers. The objective is to create business components that are usable in a variety of business applications. The components implement a business process or a business object. EJB technology allows the provider to focus on business needs so they can develop components, without needing extensive knowledge of the system services.
6.7.2 Application assembler

As shown in Figure 6-6, the application assembler, who is also an expert in the business domain, constructs the application. They are typically responsible for building the user interfaces for the application and providing the additional classes needed to complete the application. The assembler can customize Enterprise JavaBeans by changing the deployment information contained in the deployment descriptor.

---

**Application Assembler**
- Creates applications using enterprise Beans with the contracts defined by the interfaces
- May customize Beans by changing the descriptor

**Deployer**
- Deploys enterprise Beans into containers for a specific server
- May customize Beans by changing the descriptor

---

*Figure 6-6 EJB assembler and deployer*
6.7.3 Application deployer

The application deployer is responsible for deploying the application in a specific system environment. The deployer is usually an expert on the function and features of the platform, as well as the supporting technologies. The deployer maps the application to the platform environment and can make adjustments for items, such as security and the data store. The deployer can customize the Enterprise JavaBeans by changing the deployment information.

6.7.4 Server provider

The EJB specification divides the server functions into two components: the server and the container. The EJB server provider produces the middleware (server) that communicates between the platforms operating system environment and the container or bean managed beans. The middleware, created by the server provider, provides the management system services shown in the following list:

- Distributed transactions
- Object distribution
- Security
- Data persistence

The server provider can elect to include the container as part of their product.

6.7.5 Container provider

The container provider produces the EJB container that insulates the Enterprise JavaBean developer from the specifics of the EJB server services, such as transaction management, security, and persistence. The container provider ensures that the container provides a simple interface to the Enterprise JavaBean. The container accesses system services for the EJB.

6.8 Using EJB technology on the iSeries server

The Java programming language provides numerous advantages as an application development language. For the iSeries server, the main focus of Java application development has been client/server oriented. Because the iSeries server does not have a native graphical user interface, it is not optimized for the display of client windows and graphics. In the client/server environment, the iSeries server normally plays the role of a server. The Java programs run on the client and access the iSeries database using the iSeries host server programs.

Starting with OS/400 V4R2, Java programs can run on the iSeries server. The main disadvantage in using Java for developing business applications is the extra programming required to manage the services required for the runtime environment.

EJB technology provides the means to make Java a viable language for iSeries business application development. It focuses on server-side Java. EJB technology returns the focus of the application developer to the business application. The developer does not deal with system-level services. The iSeries server provides a secure and scalable environment for running Java applications, while EJB application servers provide server and container support.
6.8.1 Overview of Java for the iSeries server

The Java environment for the iSeries server is Java-compatible, which means it conforms to the Sun standard and can run 100% pure Java code without modification or re-compilation. The Java Virtual Machine (JVM) is implemented in the iSeries machine interface (MI), providing a high degree of integration with the underlying system. The integrated JVM includes an advanced garbage collection algorithm that improves Java performance and scalability. In addition to the integrated, compatible runtime, the iSeries Developer Kit for Java provides the additional commands, tools, and classes needed for Java development on the iSeries server.

Like other Java environments, Java source code files on the iSeries server are ASCII text files stored in the integrated file system. These are compiled into platform-independent class files (also stored in the IFS) that are interpreted by the JVM at runtime. The Java environment for iSeries also includes a Java transformer that further optimizes Java class files and is used to create permanent, optimized 64-bit iSeries program objects.

The Qshell Interpreter is also required for Java development on the iSeries server. Qshell is a command interpreter, based on POSIX and X/Open standards. It provides the Java development tools that are typically required for program development. With a few exceptions, the Java tools support the syntax and options available on most Java platforms.

Figure 6-7 shows the different pieces that comprise the Java environment on the iSeries server.

![Figure 6-7 Java environment on the iSeries server](image-url)
6.8.2 EJB technology on the iSeries server

An EJB server runs as an application above OS/400 on the iSeries server. For the most part, current EJB servers are written in Java and require the iSeries Developer Kit for Java on the system. The iSeries strategy is to support application servers that deliver back-end systems integration and business-to-business e-commerce opportunities.

6.9 EJB interoperability with other distributed architectures

The EJB architecture supports distribution using the standard Java remote method invocation (RMI). Every enterprise bean has a home interface and a remote interface that declare the methods a client can indirectly invoke on the enterprise bean. The objects that implement these interfaces are Java RMI remote objects provided by the container. At runtime, the client receives a reference to the home or remote object, and invokes methods on that reference. The EJB container intercepts the method request and provides services before passing the request on to the enterprise bean. The communication between the references (or stubs) on the client and the objects on the server is defined by the Java RMI specification.

6.9.1 CORBA

In addition to distributed object support using RMI, the EJB architecture also provides for EJB servers based on the Common Object Request Broker Architecture. Through standard EJB to CORBA mapping, an Interface Definition Language (IDL) interface can be generated from the remote interface. This mapping can be implicit when RMI over IIOP is used.

In addition to on-the-wire interoperability, CORBA-based EJB servers are required to use the OMG COS Naming Service for publishing and resolving the EJB home interface objects. The OMG Object Transaction Service is also required for transaction support in a CORBA-based EJB server.

These requirements allow a non-Java CORBA client to access enterprise beans using a CORBA-based EJB server. The client can use both CORBA and EJB objects within a transaction, and the transaction can span multiple CORBA-based EJB servers. WebSphere Application Server Version 4.0 Advanced Edition does not support CORBA clients.

6.9.2 Component Object Model (COM)

The EJB architecture provides no specific support for Microsoft’s Component Object Model. However, it is possible to wrap the client interface to an enterprise bean with a COM object to make it accessible from Visual Basic or Visual C++. Conversely, it is possible to wrap a COM object to make it accessible from Java.

There are a number of products available that provide COM to Java interface support. WebSphere Enterprise Edition 4.0 and Linar's J-Integra such products. They provide COM-Java bridging tools. You can access ActiveX components as if they were Java objects. You can also access pure Java objects as if they were ActiveX components.

For more information about J-Integra, see: http://www.linar.com/
6.9.3 IBM WebSphere Business Components

IBM WebSphere Business Components are easy-to-use software implementation packages. These components support these features:

- Provide a coherent set of functions
- Can be independently developed and delivered
- Can be composed from other components
- Have explicit and well-specified interfaces for the component services they provide and require

IBM has developed WebSphere Business Components to help you reduce development time, create applications that operate across diverse infrastructures, and deliver value to your company faster with a whole project solution.

6.10 Conclusion

The Enterprise JavaBean architecture provides a component model for server applications. With EJBs, you have rapid application development and the rich graphical interfaces of a client, without sacrificing the thin client manageability and the security of a server. EJBs accomplish this by making it easy to partition an application into a user interface and the business logic. The user interface can be specified in a client application (written in Java, Visual Basic, PowerBuilder, and so on) or in HTML using a Java servlet. The server-side business logic is packaged as Enterprise JavaBean components. Enterprise JavaBeans are easily deployed anywhere on the network, reused within other business applications running on disparate platforms, and easily managed from a remote console.

Enterprise JavaBean technology is not a tool to build applications. Rather, it is the architecture for defining components that can be used with a variety of tools. A stated goal of the EJB specification is to be the standard component architecture for building distributed object-oriented business applications in the Java programming language. Enterprise JavaBeans make it possible to build distributed applications by combining components developed using tools from different vendors.
Overview of the OrderEntry application

This chapter explains the RPG OrderEntry application example. This application is representative of a commercial application. However, it does not include all of the necessary error handling that a business application requires.

This chapter also introduces the application and specifies the database layout. In Chapter 10, “Building Java applications with Enterprise JavaBeans” on page 315, the RPG OrderEntry application is converted to a Web-enabled application. The goal is to use the existing RPG application to service both the Web application and the host 5250 application.
7.1 The ABC Company

The ABC Company is a wholesale supplier with one warehouse and 10 sales districts. Each district serves 3,000 customers (30,000 total customers for the company). The warehouse maintains stock for the 100,000 items sold by the Company.

Figure 7-1 illustrates the company structure (warehouse, district, and customer).

7.2 The ABC Company database

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

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

7.3 A customer transaction

A customer transaction occurs based on the following series of events:

1. Customers telephone one of the 10 district centers to place an order.
2. The district customer service representative answers the telephone, obtains the following information, and enters it into the application:
   - Customer number
   - Item numbers of the items the customer wants to order
   - The quantity required for each item
3. The customer service representative may prompt for a list of customers or a list of parts.
4. The application then performs the following actions:
a. Reads the customer last name, customer discount rate, and customer credit status from the Customer Table (CSTMR).

b. Reads the District Table for the next available district order number. The next available district order number increases by one and is updated.

c. Reads the item names, item prices, and item data for each item ordered by the customer from the Item Table (ITEM).

d. Checks if the quantity of ordered items is in stock by reading the quantity in the Stock Table (STOCK).

5. When the order is accepted, the following actions occur:

a. A new row is inserted into the Order Table to reflect the creation of the new order (ORDERS).

b. A new row is inserted into the Order Line Table to reflect each item in the order.

c. The quantity is reduced by the quantity ordered.

d. A message is written to a data queue to initiate order printing.

7.4 Application flow

Note: To download the sample code used in this redbook, please refer to Appendix A, “Additional material” on page 425, for more information.

The RPG OrderEntry application consists of the following components:

- **ORDENTD (Parts Order Entry):** Display File
- **ORDENTR (Parts Order Entry):** Main RPG processing program
- **PRTORDERP (Parts Order Entry):** Print File
- **PRTORDERR (Print Orders):** RPG server job
- **SLTCUSTD (Select Customer):** Display file
- **SLTCUSTR (Select Customer):** RPG SQL stored procedure
- **SLTPARTD (Select Part):** Display file
- **SLTPARTR (Select Part):** RPG stored procedure

Figure 7-2 show the RPG application flow. ORDENTR is the main RPG program. It is responsible for the main line processing. It calls two supporting RPG programs that are used to prompt for and select end-user input. They are SLTCUSTR, which handles selecting a customer, and SLTPARTR, which handles selecting part numbers. PRTORDERR is an RPG program that handles printing customer orders. It reads order records that were placed in a data queue and prints them in a background job.
7.5 Customer transaction flow

The following scenario walks through a customer transaction and shows the application flow. By understanding the flow of the iSeries application, you can understand the changes made to this application to support a graphical client.

Starting the application
To start the application, the customer follows this series of steps:

1. The customer calls the main program from an OS/400 command line:
   
   ```
   CALL ORDENTR
   ```
   
   When the OrderEntry application is started, the Parts Order Entry display (Figure 7-3) appears.
2. When the Parts Order Entry display appears, the user has two options:
   - Type a customer number and press the Enter key
   - End the program by pressing either F3 or F12.

   If they do not know their customer number, the user can press F4 to view a window containing a list of available customers.

3. The user presses F12 to remove the window and return to the initial panel, or scrolls through the items in the list until they find the customer they want. They indicate their choice by typing a 1 in the option field and pressing the Enter key. The selected customer is then returned to the initial panel (Figure 7-5).
4. After selecting a customer from the list, or typing a valid customer number and pressing the Enter key, the customer details are shown and an order number is assigned. An additional prompt is displayed, which allows the user to type a part number and quantity.

If the user does not know the part number, they can press F4 to view a window containing a list of available parts (Figure 7-6).

5. The user presses F12 to remove the window and return to the initial panel, or scrolling through the items in the list until they find the part they want. They type a 1 in the option field and press the Enter key to indicate their choice. The selected part is returned to the initial panel (Figure 7-7).
6. They select a customer from the list, or type a valid customer number and press the Enter key. Then the part and quantity ordered are added to the list section below the part entry fields (Figure 7-8).

7. The user may type 2 beside an entry in the list to change the order. When the user presses the Enter key, a window appears that allows the order line to be changed (Figure 7-9).
8. The user can choose to press F12 to cancel the change, press F4 to list the parts, or type a new part identifier or different quantity. Pressing the Enter key validates the part identifier and quantity. If they are valid, the order line is changed in the list, and the window is closed. The completed order is shown Figure 7-10.

9. Figure 7-11 shows you the quantity for Cross Country Ski Set is changed to 1. When the order is complete, the user presses F6 to update the database. Then, an order is placed in the data queue for printing.
Figure 7-11  Printed order

The printed order (Figure 7-11) is created by a batch process. It shows the customer details and the items, quantities, and cost of the order.

### 7.6 Database table structure

The ABC Company database has eight tables:

- District
- Customer
- Order
- Order Line
- Item
- Stock
- Warehouse
- History

The relationship among these tables is shown in Figure 7-12.
7.7 OrderEntry application database layout

The sample application uses the following tables of the database:

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

The following tables describe, in detail, the layout of the database.
Table 7-1  District Table Layout (Dstrct)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>DID</td>
<td>District ID</td>
<td>Decimal</td>
<td>3</td>
</tr>
<tr>
<td>DWID</td>
<td>Warehouse ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>DNAME</td>
<td>District Name</td>
<td>Character</td>
<td>10</td>
</tr>
<tr>
<td>DADDR1</td>
<td>Address Line 1</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>DADDR2</td>
<td>Address Line 2</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>DCITY</td>
<td>City</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>DSTATE</td>
<td>State</td>
<td>Character</td>
<td>2</td>
</tr>
<tr>
<td>DZIP</td>
<td>Zip Code</td>
<td>Character</td>
<td>10</td>
</tr>
<tr>
<td>DTAX</td>
<td>Tax</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>DYTD</td>
<td>Year to Date Balance</td>
<td>Decimal</td>
<td>13</td>
</tr>
<tr>
<td>DNXTOR</td>
<td>Next Order Number</td>
<td>Decimal</td>
<td>9</td>
</tr>
</tbody>
</table>

Primary Key: DID and DWID

Table 7-2  Customer Table Layout (CSTMR)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CID</td>
<td>Customer ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>CDID</td>
<td>District ID</td>
<td>Decimal</td>
<td>3</td>
</tr>
<tr>
<td>CWID</td>
<td>Warehouse ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>CFIRST</td>
<td>First Name</td>
<td>Character</td>
<td>16</td>
</tr>
<tr>
<td>CINIT</td>
<td>Middle Initials</td>
<td>Character</td>
<td>2</td>
</tr>
<tr>
<td>CLAST</td>
<td>Last Name</td>
<td>Character</td>
<td>16</td>
</tr>
<tr>
<td>CADDR1</td>
<td>Address Line 1</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>CCREDT</td>
<td>Credit Status</td>
<td>Character</td>
<td>2</td>
</tr>
<tr>
<td>CADDR2</td>
<td>Address Line 2</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>CDCT</td>
<td>Discount</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>CCITY</td>
<td>City</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>CSTATE</td>
<td>State</td>
<td>Character</td>
<td>2</td>
</tr>
<tr>
<td>CZIP</td>
<td>Zip Code</td>
<td>Character</td>
<td>10</td>
</tr>
<tr>
<td>CPHONE</td>
<td>Phone Number</td>
<td>Character</td>
<td>16</td>
</tr>
<tr>
<td>CBAL</td>
<td>Balance</td>
<td>Decimal</td>
<td>7</td>
</tr>
<tr>
<td>CCRDLM</td>
<td>Credit Limit</td>
<td>Decimal</td>
<td>7</td>
</tr>
<tr>
<td>CYTD</td>
<td>Year to Date</td>
<td>Decimal</td>
<td>13</td>
</tr>
</tbody>
</table>
### Table 7-3  Order Table Layout (ORDERS)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPAYCNT</td>
<td>Payment</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>CDELCNT</td>
<td>Delivery Qty</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>CLTIME</td>
<td>Time of Last Order</td>
<td>Numeric</td>
<td>6</td>
</tr>
<tr>
<td>CDATA</td>
<td>Customer Information</td>
<td>Character</td>
<td>500</td>
</tr>
</tbody>
</table>

Primary Key: CID, CDID, and CWID

### Table 7-4  Order Line Table Layout (ORDLIN)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWID</td>
<td>Warehouse ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>ODID</td>
<td>District ID</td>
<td>Decimal</td>
<td>3</td>
</tr>
<tr>
<td>OCID</td>
<td>Customer ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>OID</td>
<td>Order ID</td>
<td>Decimal</td>
<td>9</td>
</tr>
<tr>
<td>OENTDT</td>
<td>Order Date</td>
<td>Numeric</td>
<td>8</td>
</tr>
<tr>
<td>OENTTM</td>
<td>Order Time</td>
<td>Numeric</td>
<td>6</td>
</tr>
<tr>
<td>OCARID</td>
<td>Carrier Number</td>
<td>Character</td>
<td>2</td>
</tr>
<tr>
<td>OLNBR</td>
<td>Number of Order Lines</td>
<td>Decimal</td>
<td>3</td>
</tr>
<tr>
<td>OLOCAL</td>
<td>Local</td>
<td>Decimal</td>
<td>1</td>
</tr>
</tbody>
</table>

Primary Key: OWID, ODID, and OID

### Table 7-5  Order Line Table Layout (ORDLIN)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID</td>
<td>Order ID</td>
<td>Decimal</td>
<td>9</td>
</tr>
<tr>
<td>ODID</td>
<td>District ID</td>
<td>Decimal</td>
<td>3</td>
</tr>
<tr>
<td>OWID</td>
<td>Warehouse ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>OLNBR</td>
<td>Order Line Number</td>
<td>Decimal</td>
<td>3</td>
</tr>
<tr>
<td>OLSPWH</td>
<td>Supply Warehouse</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>OLIID</td>
<td>Item ID</td>
<td>Character</td>
<td>6</td>
</tr>
<tr>
<td>OLQTY</td>
<td>Quantity Ordered</td>
<td>Numeric</td>
<td>3</td>
</tr>
<tr>
<td>OLAMNT</td>
<td>Amount</td>
<td>Numeric</td>
<td>7</td>
</tr>
<tr>
<td>ODLV</td>
<td>Delivery Date</td>
<td>Numeric</td>
<td>6</td>
</tr>
<tr>
<td>OLSTI</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
</tbody>
</table>

Primary Key: OLWID, OLDID, OLOID, and OLNBR
This redbook concentrates on the use of the iSeries server as a database server in a client/server environment. In some cases, we use SQL to access the iSeries database. In other cases, we use native database access.

### Table 7-5  Item Table Layout (ITEM)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>IID</td>
<td>Item ID</td>
<td>Character</td>
<td>6</td>
</tr>
<tr>
<td>INAME</td>
<td>Item Name</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>IPRICE</td>
<td>Price</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>IDATA</td>
<td>Item Information</td>
<td>Character</td>
<td>50</td>
</tr>
</tbody>
</table>

Primary Key: IID

### Table 7-6  Stock Table Layout (Stock)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Real name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>STWID</td>
<td>Warehouse ID</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>STIID</td>
<td>Item ID</td>
<td>Character</td>
<td>6</td>
</tr>
<tr>
<td>STQTY</td>
<td>Quantity in Stock</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>STDI01</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI02</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI03</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI04</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI05</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI06</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI07</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI08</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI09</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STDI10</td>
<td>District Information</td>
<td>Character</td>
<td>24</td>
</tr>
<tr>
<td>STYTD</td>
<td>Year to Date</td>
<td>Decimal</td>
<td>9</td>
</tr>
<tr>
<td>STORDERS</td>
<td>Number of orders</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>STREMORD</td>
<td>Number of remote orders</td>
<td>Decimal</td>
<td>5</td>
</tr>
<tr>
<td>STDATA</td>
<td>Item Information</td>
<td>Character</td>
<td>50</td>
</tr>
</tbody>
</table>

Primary Key: STWID and STIID

### 7.8 Database terminology

This redbook concentrates on the use of the iSeries server as a database server in a client/server environment. In some cases, we use SQL to access the iSeries database. In other cases, we use native database access.
The terminology used for the database access is different in both cases. In Table 7-7, you find the correspondence between the different terms.

Table 7-7  Database terminology

<table>
<thead>
<tr>
<th>iSeries native</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>Collection, schema</td>
</tr>
<tr>
<td>Physical file</td>
<td>Table</td>
</tr>
<tr>
<td>Field</td>
<td>Column</td>
</tr>
<tr>
<td>Record</td>
<td>Row</td>
</tr>
<tr>
<td>Logical file</td>
<td>View or index</td>
</tr>
</tbody>
</table>
iSeries EJB application development scenario

This chapter discusses how the example RPG application discussed in Chapter 7, “Overview of the OrderEntry application” on page 231, can be implemented using EJB components. The application architecture is detailed by identifying the objects, using Unified Modeling Language (UML) diagrams. These objects are identified as entity and session beans. This chapter discusses the reasons behind these selections, the implementation details of these choices, and how these EJB components are used to create various types of applications.
8.1 OrderEntry application architecture with objects

This architecture example assumes that the database tables already exist and that an object architecture is built surrounding them. Initially, it is important to name the identifiable objects in the application. A variety of methods exist for creating an object-oriented application. One common way is to describe the process and highlight the nouns, as a starting point for identifying the objects.

In our example, a customer resides in a particular district. The customer contacts an order entry clerk to place an order for one or more items. Each item appears as an order detail line in the order. Before the order can be placed, the order entry clerk checks to see if there is enough of the item in stock.

This list of objects corresponds to the tables identified in Chapter 7, "Overview of the OrderEntry application" on page 231, which also identify the fields contained in the tables. An object diagram corresponding to these tables is shown in Figure 8-1.

![Figure 8-1 Application object diagram](image)

In Figure 8-1, we make one further refinement. We separated out an address object to hold the common address information that is required by the customer and the district.
This list of objects represents all of the tables. However, the description includes another noun—the order entry clerk (OrderEntryClerk). This object becomes important in the next step of identifying objects, which is to identify the relationships between the objects. The OrderEntryClerk is an object with the primary purpose of acting on other objects. The relationships between the identified objects are either containment or use relationships. Figure 8-2 shows all of the objects that were previously identified and the relationships between them.

Figure 8-2  Object relationships

Both the customer and the district class contain an address. This provides reuse, one of the key benefits of object-oriented programming. As shown later in this chapter, reuse, as a goal, may need to be overlooked to gain other advantages, such as using some of the programming facilities. The order contains a collection of order details. Because the order entry clerk initially takes down the order information, it also needs to contain a collection of order details (OrderDetails). This collection of order details is used as a parameter when calling the method necessary to create a new order.

The use relationships primarily involve the OrderEntryClerk, as well. The OrderEntryClerk acts on several different objects. It retrieves a list of customers when the order is initiated. Later, when an order is placed, it verifies the existence of a specific customer. It also retrieves a list of items to be ordered. As requests for creating order details are received by the OrderEntryClerk, it checks the stock availability of the item being ordered. After all of the order details are created, the order is placed through the OrderEntryClerk.
To this point, all of the actions that have been described reflect only as reads of the database tables. Placing the order is the activity that involves writing to the tables represented by the objects. Because this is a complex task and represents a business transaction, OrderPlacement is represented as a separate object. The OrderPlacement object also acts on several different objects. It creates a new order object. Because it receives a collection of order details as a parameter, it merely passes them on to the order for them to be created. The OrderPlacement object also retrieves the next order number from the district, updates the customer balances, and reduces the stock quantity. All of these updates are completed as a single unit of activity and, therefore, are encapsulated in a transaction, as shown later in this chapter.

The objects described in this section serve one of two purposes. They either represent data that is maintained in a table, or they represent actions on the data or business tasks. These two types of objects correspond to entity and session beans, respectively.

### 8.2 Business data: Entity Enterprise JavaBeans

This section discusses the entity enterprise beans we developed for our example application. It also highlights some of the key topics we address in writing them.

Entity beans are persistent objects that represent data stored in some persistent fashion, such as a database or an existing legacy application. When you modify the variables for the bean, you modify the data in the database. In our case, the persistent store is the relational database on the iSeries server.

Our analysis identified the following five entity beans:

- Stock
- Item
- Customer
- Order
- District

### 8.2.1 Database access: Using a connection pool

In our example application, we use connection pools to optimize performance. Connection pools are commonly used because creating a new JDBC connection is time consuming and uses system resources. With a pool, WebSphere Application Server creates connection objects when the server starts. When a bean needs a connection to the database, it gets one from the pool. When the bean is done with the connection, it returns it to the pool.

The way you return a WebSphere Application Server connection to the pool is to call the connection close method, for example, `con.close()`. This works because a connection is actually a class that wraps a JDBC connection.

**Using WebSphere connection pools**

To optimize performance in the enterprise bean data access methods, the developer should not use the `java.sql DriverManager getConnection(String url)` method to get a JDBC connection to the database. The developer should take advantage of the WebSphere Application Server connection pools defined in the DataSource Configuration object. For information about creating data sources, see Chapter 3 in the redbook *WebSphere 4.0 Installation and Configuration on the IBM iSeries Server*, SG24-6815. The code in Example 8-1 shows this technique.
Example 8-1   Using a DataSource

```java
import javax.naming.*;
import javax.sql.*;
import java.sql.*;
........
DataSource ds;
try {
    InitialContext initCtx = new InitialContext();
    String dataSourceName = (String)initCtx.lookup("java:comp/env/dataSourceName");
    ds = (DataSource) initCtx.lookup("jdbc/" + dataSourceName);
}
........
Connection con = ds.getConnection();
Statement stmt = con.createStatement();
........
ds.close()
```

DataSources can be found by looking them up in the WebSphere Application Server JNDI
naming space. To use a DataSource, you must:

1. Get an instance of the local iSeries javax.naming.InitialContext object.
2. Use its `lookup()` method, specifying `jdbc/<data source name>` as a parameter, to get an
   instance of DataSourceBean in a variable of type javax.sql.DataSource.

   **Note:** If you are working on WebSphere Application Server Advanced Edition 3.5, the
   DataSource is a variable of type com.ibm.ejs.dbm.jdbcext.javasql.DataSource.

3. Use the DataSourceBean `getConnection()` method to get a JDBC connection out of the
   DataSource connection pool.
4. When you are done with the connection, use the `java.sql.Connection close()` method to
   put the connection back in the pool.

The name of the DataSource should be externalized, because it is not likely that you will know
it during bean development. A good technique is to place the DataSource name in an
environment variable of the bean deployment descriptor. It can be changed at deployment
time.

Figure 8-3 shows how to set the environment variable in Application Developer:

1. Right-click the project and select **Open with-> EJB Editor**.
2. Select the **Environment** tab.
3. Select the bean and click **Add** to add the environment variables.
8.2.2 Persistence: Container or bean managed

After you identify your entity beans, you must decide whether to use container-managed persistence (CMP) or bean-managed persistence (BMP). Use CMP, unless you want to do something it does not support, because CMP is much simpler to implement.

The common reasons for using BMP are:

- Performance
- Complex entity/database mapping

In our example, we use BMP in the Order bean to achieve maximum performance. The underlying data is stored in two tables: order (ORDERS) and order line (ORDLIN). It may also be used many times during the placement of an order. For a complete description of the applications tables, see 7.6, “Database table structure” on page 239.

Another reason to reject CMP is that the bean data is stored in a legacy application rather than in a database.

In our example, we also use BMP for the Customer bean because the Customer bean contains an Address object, which contains fields that need persisting.

Note: As a bean developer, you must choose which form of persistence to use. It is not possible to write an entity bean that can be switched from BMP to CMP at deployment time. Changing from one form of persistence to the other requires changes in your source code for the bean.

8.2.3 Container-managed persistence

In our example, the Stock, Item, and District beans use container-managed persistence (CMP). This means that we rely on WebSphere Application Server to update the Stock, Item, and District database tables any time we change the data in our entity bean. In Chapter 9, “Developing EJBs with Application Developer” on page 265, we use the Stock bean to describe how to implement CMP.
8.2.4 Bean-managed persistence

When you create an entity bean that uses bean-managed persistence, you take responsibility for the code that inserts, selects, updates, and deletes data from the underlying database or other persistent storage mechanism by implementing four life-cycle methods (listed below).

In the simplest case where a bean instance represents one row in a database table, the responsibilities of the methods are summarized in the following list:

- **ejbCreate**: Uses the client-supplied arguments to perform the following tasks:
  - Insert a new record in the database.
  - Set the instance variables for the bean.
  - Return an instance of the primary key for the bean.

- **ejbLoad**: Uses the primary key associated with the instance to perform the following tasks:
  - Retrieve the record from the database.
  - Update the bean instance variables from the database.

- **ejbStore**: Updates the database record using the values in the instance variables of the bean.

- **ejbRemove**: Deletes the database record.

The container is responsible for calling these methods at the appropriate time to maintain the synchronization between the bean and the underlying database.

In addition to the life-cycle methods, you must implement the **ejbFindByPrimaryKey** method and any other finder methods that you want to define. The container calls this method when the client calls **findByPrimaryKey** on the home interface.

When you use bean-managed persistence, you are not limited to storing a bean as a single row in a table. For example, the Order bean in our example application gets its data from two tables and several rows. You are not even limited to storing your data in a database. Regardless of how the data is stored, the basic function of the four methods remains the same.

8.3 Business processes: Session Enterprise JavaBeans

This section discusses the session enterprise beans that we develop for our example application. It also highlights some of the key topics addressed in writing them. Session beans are non-persistent objects that run on the server and implement business logic.

Our example uses two session beans:

- OrderEntryClerk
- OrderPlacement

8.3.1 Three-tier versus two-tier architecture

Session beans allow you to easily implement a three-tier architecture. The traditional client/server architecture is a two-tiered architecture in which there are many clients and a single database. The clients often implement the business logic as well as the user interface. In a three-tiered architecture, business logic is moved out of the many clients and into a session bean running in an EJB server. The clients talk to the session bean and the bean talks to the databases.
This architecture makes the client code smaller and easier to maintain. You can change the business logic without redistributing any client code. More importantly, this architecture lets you share business logic across different types of clients. If you implement your business logic in a bean, you can use that same bean in a Java application, an applet, a servlet, or a non-Java application as shown in Figure 8-4.

8.3.2 Stateless or stateful beans

A stateless session bean allows you to write code that runs on the server and can be reused by many clients. However, the stateless session bean does not remember anything about the client between method invocations. In fact, if your client calls two methods of a stateless session bean, there is no guarantee that both methods are called on the same object in the server.

Conversely, a stateful session bean remembers information between method calls. To use a trivial example, a stateful session bean can have two methods – setName and getName. A client can pass a name to the bean using setName and retrieve that name in another call using getName.

8.3.3 Order Entry example

In our example application, the OrderPlacement bean is a stateless session bean, and the OrderEntryClerk bean is a stateful session bean. The OrderPlacement bean encapsulates the business logic to place an order. The OrderEntryClerk encapsulates the ordering process. The OrderEntryClerk bean is used by GUI programs such as Java applications or applets. The OrderPlacement bean is used by the OrderEntryClerk bean to actually place the order.

8.3.4 Stateless session bean: OrderPlacement

The OrderPlacement bean encapsulates the business logic to place an order in a method named placeOrder().

The source code for the OrderPlacement bean consists of the following four components:

- **Deployment descriptor**: The ejb-jar.xml file that contains attribute and environment settings that define how the container invokes enterprise bean functions
OrderPlacementHome: The home interface for the bean
OrderPlacement: The remote interface for the bean
OrderPlacementBean: The implementation of the bean

Deployment descriptor

Every enterprise bean must have a deployment descriptor that contains settings for the bean attributes. There are two different types of attributes – bean-level and method-level.

A bean-level attribute is an attribute that applies to the entire bean, and it has a single value for the enterprise bean. A method-level attribute can be set on a single bean method. If a method-level value is specified, that value is used. If there is no method-level value, the bean-level value is used.

To see the deployment descriptor for the OrderPlacement bean, select the project ToWebSphere in the J2EE perspective. Right-click and select Open With-> EJB Editor. Select the Beans tab and click OrderPlacement. Figure 8-5 shows the settings used in our example beans.

![Figure 8-5  OrderPlacementBean properties](image)

We define the OrderPlacement bean as a stateless session bean using the type option value in the deployment descriptor as shown in Figure 8-5. This tells our container that it does not need to assign one instance of the OrderPlacement bean to each client. Instead, it can create a pool. For example, there may be one hundred clients connected to the server, but only five call placeOrder at any one time. In this case, the EJB container only creates five instances of the OrderPlacement bean.
The Environment tab of the deployment descriptor, shown in Figure 8-6, allows you to enter information that is specific to the bean. This is useful to externalize values that depend on the specific deployment environment. The `placeOrder` method of the `OrderPlacementBean` class uses a data queue to pass order information to other applications. We use the deployment descriptor to externalize the name of the data queue.

**Figure 8-6  OrderPlacement bean properties environment tab**

**Home interface (OrderPlacementHome)**
As shown in Figure 8-7, the home interface for a session bean does not have any finder methods. They don’t exist because a session bean does not represent an entity in a persistent datastore that you can locate.

**Figure 8-7  OrderPlacement home interface**
Remote interface (OrderPlacement)

We define the business methods of the bean in the remote interface. In the case of the OrderPlacement bean, there is only the `placeOrder` method. As shown in Figure 8-8, three versions of the `placeOrder` method are available, depending on the parameters passed by the client.

![Figure 8-8 OrderPlacement remote interface](image)

Bean implementation (OrderPlacementBean)

We use the OrderPlacementBean class, shown in Figure 8-9, to implement the EJB SessionBean interface, the create method from the home interface, and the business logic.

![Figure 8-9 OrderPlacementBean](image)

The SessionBean interface, together with the create method from the home interface, defines the bean instance life cycle. As you can see in Example 8-2, very little happens in these methods. In fact, the only work that is done is in the `setSessionContext` method.

Example 8-2  Session bean life-cycle methods

```java
// private variables
private transient SessionContext ctx;
private transient Properties props;
// Implementation of methods required by the SessionBean interface.
```
public void ejbActivate() {
}

public void ejbRemove() {
}

public void ejbPassivate() {
}

// Implementation of create methods defined in home interface OrderPlacementHome
public void ejbCreate() {
}

The business logic is implemented in the placeOrder method. As shown in Example 8-3, placeOrder uses the District, Stock, Order, and Customer entity beans to perform much of its processing. It also uses the private method, writeDataQueue, to write the order to the data queue. We use an iSeries batch program to read the data queue entries and print order information.

Example 8-3  The placeOrder method processing

```java
public float placeOrder(String wID, int dID, String cID, Vector orderLines) throws javax.ejb.EJBException {
    float oID = 0;
    try {
        //The InitialContext will let us retrieve references to the entity beans we need.
        InitialContext initCtx = new InitialContext();
        //Get the Order Number from the District entity bean.
        DistrictHome dHome = (DistrictHome) initCtx.lookup("District");
        DistrictKey districtID = new DistrictKey(dID, wID);
        District district = (District) dHome.findByPrimaryKey(districtID);
        int oIDint;
        oIDint = district.getNextOrderId(true);
        oID = oIDint;
        // true' tells the District to increment the order id.
        oID = oIDint;
        //Update the Stock level for each item in an order line using the Stock entity bean
        StockHome sHome = (StockHome) initCtx.lookup("Stock");
        Enumeration lines = orderLines.elements();
        float orderTotal = 0;
        while (lines.hasMoreElements()) {
            OrderDetail od = (OrderDetail) lines.nextElement();
            String itemID = od.getItemID();
            int itemQty = od.getItemQty();
            orderTotal += od.getItemAmount();
            //Calculate the order total while we are going through the orders.
            StockKey stockID = new StockKey(wID, itemID);
            Stock stock = (Stock) sHome.findByPrimaryKey(stockID);
            stock.decreaseStockQuantity(itemQty);
        }
        //Save the Order to the database by creating an Order entity bean
        OrderHome oHome = (OrderHome) initCtx.lookup("Order");
        oHome.create(wID, dID, cID, oID, orderLines);
        //Update the Customer records using the Customer entity bean
        CustomerHome cHome = (CustomerHome) initCtx.lookup("Customer");
        CustomerKey customerID = new CustomerKey(cID);
        Customer customer = (Customer) cHome.findByPrimaryKey(customerID);
        ...
8.3.5 Stateful session bean: OrderEntryClerk

The OrderEntryClerk encapsulates the ordering process. We define the OrderEntryClerk bean as a stateful session bean to use it as an order clerk. This bean keeps a record of who the customer is and what items they want to order.

The source code for the OrderClerk bean consists of the following four components:

- **Deployment descriptor**: The ejb-jar.xml file that contains attribute and environment settings that define how the container invokes enterprise bean functions
- **OrderEntryClerkHome**: The interface for the bean
- **OrderEntryClerk**: The remote interface for the bean
- **OrderEntryClerkBean**: The implementation of the bean

**Deployment descriptor**
To see the deployment descriptor for the OrderEntryClerk bean, select the project ToWebSphere in the J2EE perspective. Right-click and select Open With-> EJB Editor. Select the Beans tab and click OrderEntryClerk. Figure 8-10 shows the settings used in our bean.
We define the OrderEntryClerk bean as a stateful session bean using the *type option* value attribute value in the deployment descriptor as shown in Figure 8-10. This tells our container that it needs to assign an instance to each client and that it needs to store the non-transient variables of the bean across method calls.

The Environment tab of the deployment descriptor, shown in Figure 8-11, allows you to enter information specific to the bean. This is useful to externalize values that depend on the specific deployment environment. We use the deployment descriptor to externalize the name of the DataSource.
Figure 8-11  OrderEntryClerkBean properties environment tab

OrderEntryClerkBean
We use the OrderEntryClerkBean class, shown in Figure 8-12, to implement the EJB SessionBean interface, the create method from the home interface, and the business logic. The SessionBean interface, together with the create method from the home interface, defines the bean instance life cycle.
Figure 8-12 OrderEntryClerkBean

Implementing a state
As a bean developer, use the following steps to implement a state:

1. Define non-transient variables in your bean.
2. Initialize these variables in the ejbCreate methods.
3. Use these variables in the business logic.

The bean state consists of any variables we define that are not transient. As shown in Example 8-4, the OrderEntryClerk bean has two variables that maintain the bean’s state.

Example 8-4 Defining state variables

```java
// Following is the state in this stateful session bean
public String custID;
public Vector items;
```

As shown in Example 8-5, we initialize these variables in the ejbCreate method. In our example, ejbCreate does not take any arguments. If we write the example differently, where the client passes the customer ID in as an argument when creating the bean, we set custID to the provided argument.

Example 8-5 Initializing values

```java
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.RemoteException {
    try {
        InitialContext initCtx = new InitialContext();
        String dataSourceName = (String)initCtx.lookup("java:comp/env/dataSourceName");
```

```java
```
ds = (DataSource) initCtx.lookup("jdbc/" + dataSourceName);
}
catch (Exception e) {
    throw new RemoteException("Error in ejbCreate: " + e.getMessage());
}
items = new Vector();
custID = null;
isDirty = true;

We use these variables in the business methods. The three methods related to the bean state are:

- setCustomer (Example 8-6)
- addOrderLine (Example 8-7)
- placeOrder (Example 8-8)

In the setCustomer method, we set the value of custID.

Example 8-6  The setCustomer method

```java
public void setCustomer(String cid) throws RemoteException {
    if (verifyCustomer(cid)) {
        custID = cid;
    } else {
        throw new CpwejbException("Customer id " + cid + " not valid");
    }
}
```

In the addOrderLine method as shown in Example 8-7, we add another OrderDetail object to the items vector. These are the items that we want to order.

Example 8-7  The addOrderLine method

```java
public void addOrderLine(String iid, int quantity) throws RemoteException {
    if (custID == null) {
        throw new RemoteException("OrderEntryClerkBean:  Customer ID must be set first");
    }
    Item item = null;
    try {
        InitialContext initCtx = new InitialContext();
        Object homeObject = initCtx.lookup("Item");
        ItemHome home = (ItemHome)javax.rmi.PortableRemoteObject.narrow((org.omg.CORBA.Object) homeObject,
        com.ibm.itso.roch.wasaejb.ItemHome.class);
        ItemKey itemK = new ItemKey(iid);
        item = home.findByPrimaryKey(itemK);
    } catch (Exception e) {
        throw new RemoteException(e.getMessage());
    }
```
OrderDetail ol = new OrderDetail(iid, item.getItemPrice() * quantity, quantity);
items.addElement(ol);
}

The placeOrder method is responsible for placing an order. As shown in Example 8-8, it uses the OrderPlacement EJB’s placeOrder method to actually place the order.

Example 8-8  The placeOrder method

```
public String placeOrder() throws RemoteException {
    String orderNumber = null;
    if (custID == null) {
        throw new CpwejbException("OrderEntryClerkBean: Hey buddy! Customer ID must be set first");
    }
    if (items.size() == 0) {
        throw new CpwejbException("OrderEntryClerkBean: What are you thinking! No order lines!");
    }
    try {
        InitialContext initCtx = new InitialContext();
        OrderPlacementHome home = (OrderPlacementHome) initCtx.lookup("OrderPlacement");
        OrderPlacement placement = home.create();
        float number = placement.placeOrder(whid, did, custID, items);
        orderNumber = Float.toString(number);
        orderNumber = orderNumber.substring(0, orderNumber.length() - 2);
        // Clear out the state of the session bean at this point
        items = new Vector();
        custID = null;
    }
    catch (Exception e) {
        throw new RemoteException(e.getMessage());
    }
    return orderNumber;
}
```

The OrderEntryClerk bean also implements several methods that pertain to activities that a clerk performs, but does not use its state:

- The findAllCustomers method retrieves a list of customers from the server.
- The findAllItems method retrieves a list of items from the server.
- The findRangeOfItems method retrieves a subset of items from the server.
- The verifyCustomer method verifies that a customer number is valid.
- The getCustomerForOrder method retrieves the customer number for an order.
- The getItemsForOrder method retrieves the order lines for an order.

These methods are used by client applications. See Chapter 10, “Building Java applications with Enterprise JavaBeans” on page 315, for details about how we use these methods.

### 8.4 Conclusion

This chapter covered the architecture and design of the Enterprise JavaBeans that are used in this redbook.
Our analysis identified the following five entity beans:

- Stock
- Item
- Customer
- Order
- District

In this chapter, we designed the following two session beans:

- OrderEntryClerk
- OrderPlacement

We also covered the following design decisions:

- Whether to use container managed or bean-managed persistence for the entity beans
- Whether to use stateful or stateless session beans

We covered key code snippets and control file settings that we used in implementing our beans. The entire source of all the beans is available for download from our Web site as explained in Appendix A, “Additional material” on page 425.

Now that our Enterprise JavaBeans are designed and written, we are ready to use them in applications. These applications are demonstrated in Chapter 10, “Building Java applications with Enterprise JavaBeans” on page 315. This chapter demonstrates the power of using Enterprise JavaBeans. After they are written and deployed on a server, such as the iSeries server, client programmers can use them with very little knowledge of how the beans actually work.

The client programmer only has to know what methods the Enterprise JavaBeans support and how to call them. All the difficult work is done by the bean writer or the Java server. Another key advantage is that whether you are writing a Java application, a Java applet, a Java servlet, or even a Visual Basic program, it always works the same way. You only need to call the methods provided by the Enterprise JavaBeans to handle the application processing.
Developing EJBs with Application Developer

The objective of this chapter is to show you how to develop Enterprise JavaBeans using Application Developer. This chapter focuses on entity bean creation and testing.

Entity beans with container-managed persistence (CMP) fields must be mapped to database tables in order to persist the data that the bean represents. You map entity beans to database tables by creating a map. Maps are used to generate the SQL and other supporting code needed to make enterprise beans persistent. The EJB to RDB mapping wizard supports the following techniques for mapping enterprise beans to database tables:

- **Bottom-up mapping**: In this approach, the CMP enterprise bean and the mapping between the table and CMP fields are automatically generated.
- **Top-down mapping**: This approach assumes that you already have existing enterprise beans. The enterprise bean design determines the database design.
- **Meet-in-the-middle**: In this approach, it is assumed that you have existing enterprise beans and a database schema. You do a meet-in-the-middle mapping by matching fields in enterprise beans to fields in the database table.

For an introduction to Enterprise JavaBeans, refer to Chapter 6, “Introduction to Enterprise JavaBeans” on page 215.
9.1 Bottom-up mapping

When application developers move to the Enterprise JavaBeans technology, it is most likely that they want to map any new entity beans to legacy data.

This section shows how Application Developer can automatically generate a Stock entity bean from the existing Stock database table. We use the Stock table from the RPG OrderEntry application discussed in Chapter 7, “Overview of the OrderEntry application” on page 231.

9.1.1 Creating an EJB project

An EJB project is a logical module that allows you to manage your enterprise beans. To create an EJB project, follow these steps:

1. In the J2EE view of the J2EE perspective, select File-> New-> EJB Project.
2. Enter a project name.
3. Type or select the Enterprise Application project name to which this EJB project will be added. If the Enterprise Application does not exist, it is created.
4. Click Next. You can also click Finish at this point to create a default EJB project if you have no module dependencies or build settings to specify.

9.1.2 Importing a schema

The first step is to create a database schema that is used to create the Enterprise JavaBean. A schema is a representation of a database. On the iSeries server, we call it a library or collection. A schema map is a set of definitions for all the attributes that match all the columns for a database table, view, or SQL statement. You can create a schema based on an existing iSeries table. Follow these steps to create the Stock table schema:

1. Click the Open Perspective button on the perspective toolbar as shown in Figure 9-1.
2. Select the **Data** perspective. A window appears like the example in Figure 9-2.

3. Ensure that the **DB Explorer** view is selected. Then right-click anywhere on this view and select **New Connection** as shown in Figure 9-3.
4. Complete the Database connection dialog as shown in Figure 9-4.

Figure 9-4   Database connection
Enter the following parameters:
- **Connection name**: testcon or any other name
- **Database**: This is the datasource name that we use to connect to the iSeries server
- **userid**: A valid iSeries user ID
- **password**: A valid password
- **Database Vendor Type**: DB2 UDB for iSeries, V5R1 or V4R5 depending on your iSeries server
- **JDBC Driver**: Other DB2 UDB for iSeries driver
- **JDBC driver class**: com.ibm.as400.access.AS400JDBCDriver
- **Class location**: The location of the IBM Toolbox for Java jt400.jar file
- **Connection URL**: jdbc:as400://SystemName

5. Click the Finish button. The connection is established and the DB Explorer shows all the libraries on the server that you are authorized to.

Attention: The DB Explorer view represents the physical schema inside the database, and no modification can be made.

6. Locate the library that contains the tables. Select the **Stock** table. Right-click and select **Import to Folder** as shown in Figure 9-5.

![Figure 9-5 Importing the Stock table schema](image)

7. The Import to folder window appears. Click the **Browse** button next to the Folder as shown in Figure 9-6 and select your project. Then click **OK**.
8. Select the Database and Schema from the drop-down list and click **Finish** as shown Figure 9-7. Click **Yes to All** to any dialog messages.

9. Go to the Navigator view. It's interesting to spend some time looking at the files generated in the project for the imported schema. This allows us to view how Application Developer has stored the metadata that represents the schema as shown in Figure 9-8.
10. Application Developer creates a separate .xmi file for each table in the schema with the file extension .tblxmi. It also creates an XML file for the schema with a .schxmi extension. Each of these files has its own editor. Double-click the file to open the editor. These files are also copied into the bin directory of the project for inclusion in the EJB JAR file.

11. Switch to the J2EE perspective. The database schema appears under Databases as shown in Figure 9-9.
12. Double-click the Stock table schema and then select the **Primary Key** tab on the Stock table editor as shown in Figure 9-10.

![Figure 9-10  Stock table schema browser](image)

The schema browser could not find a primary key for the Stock table. The Application Developer wizard can generate a key descriptor only for files with primary key constraints. STOCK is a keyed physical file, but it does not have any primary key constraints.

13. To manually add the key descriptor for the Stock table, select the fields (**STWID** and **STIID**) and move them to the right pane using the >> button.

14. Close the schema browser and save the changes.

### 9.1.3 Creating the entity EJB from the imported schema

Now the schema is generated for the STOCK table and is imported to our working project. Follow these steps to create the Stock bean from the schema:

1. Under the EJB modules in the J2EE perspective, select the working project. In this case, we use **ItsoEjb** as our project. Right-click and select **Generate-> EJB to RDB Mapping** as shown in Figure 9-11.
2. The mapping wizard opens as shown in Figure 9-12. Since no EJBs are currently defined in the project, only Bottom Up is enabled. Select the **Bottom Up** radio button and select the **Open mapping editor after completion** check box. Click **Next**.
3. Leave the Prefix for generated EJB classes field blank as shown in Figure 9-13. In this case, we use `com.ibm.itso.roch.wasaejb` as the package name. By leaving the Prefix for generated EJB class field blank, the wizard uses the table names defined in the schema. Click the **Finish** button.

![Create new EJB/RDB Mapping](image1)

*Figure 9-13  Creating a new EJB/RDB mapping*

When Application Developer completes the Create EJB to RDB mapping, the mapping file `Map.mapxmi` is opened in the J2EE perspective as shown in Figure 9-14.

![EJB to RDB editor](image2)

*Figure 9-14  EJB to RDB editor*

The top half of the editor shows the EJB and table definitions, and the bottom half shows how the contained fields and columns are mapped to each other. If you need to open this file again in the future, you must switch to the Navigator view and double-click the `Map.mapxmi` file.
Investigating the generated files

Again, it’s worth spending some time navigating and reviewing the generated files from the Create EJB to RDB mapping wizard. The J2EE and Navigator views of the generated files in the project are shown in Figure 9-15.

Figure 9-15  J2EE and Navigator perspectives

EJB classes review

You see four Java classes for the Stock EJB generated in the J2EE perspective:

- **StockHome**: The home interface. This is a factory that is used to create and find the instances of the Stock EJB.
- **Stock**: The remote interface. This class determines which methods can be remotely invoked.
- **StockBean**: The implementation class. This is where the logic is defined for the methods in the Stock EJB.
- **StockKey**: The key class. This is used to represent a unique key for the entity bean.

Each of these classes appears in the Navigator view as .java files in the package folder defined in the mapping wizard.

Generated EJB metadata

The Navigator view as shown in Figure 9-15 also shows the xmi files generated by the mapping wizard in the EJB project:

- **ejb-jar.xml**: The deployment descriptor for this EJB module
- **ibm-ejb-jar-bnd.xmi**: WebSphere bindings for the EJB module
- **ibm-ejb-jar-ext.xmi**: WebSphere extensions for the EJB module
- **Map.mapxmi**: The mappings between the EJB and the database schema
This is different from VisualAge for Java, where the metadata is stored internally inside the repository.

9.1.4 Defining getter methods as read-only

Application developer does not, by default, mark all getter methods in the EJB as read-only. This is an important step for obtaining reasonable performance from entity EJBs because it prevents the application server from writing the contents of the EJB back to the datastore after each get method is invoked. To set the getter methods as read-only, follow these steps:

1. Select the ItsoEjb project in the J2EE view of the J2EE perspective. Right-click and select Open With-> EJB Extension Editor.

2. Click the Methods tab and select the Stock bean. Click the + sign next to Stock to expand it and then expand Remote methods to view all the methods in the remote interface as shown in Figure 9-16.

![Figure 9-16 Setting EJB get methods to read only](image)

3. Select the getter methods one by one and change the Access intent for each to READ from the drop-down list as shown in Figure 9-17.
4. After you are done with all the methods, save and close the editor.

9.1.5 Deploying the Stock bean in the WebSphere Test Environment

In this section, we take you through the steps to define a JDBC data source for the EJB. We then validate and test the EJB.

**Binding the Stock EJB to a JDBC data source**

Each CMP EJB module must be assigned a JDBC data source reference in order for its CMP entities to persist correctly. Use the following steps to bind the data source:

1. Select the ItsoEjb project under the EJB module in the J2EE view of the J2EE perspective. Right-click and select **Open With-> EJB Extension Editor**.

2. Select the **Bindings** tab. Then click your project and complete the data source information for the ItsoEjb module. Enter the JNDI name for the data source name used while creating the schema. Also enter a valid user ID and password as shown in Figure 9-18.
3. Save the changes by selecting **File-> Save EJB Extension Editor**.

**Generating the deployed code**

In this section, we show you the steps to generate the deployed code to support execution under WebSphere Application Server:

1. Switch to the J2EE view in the J2EE perspective and select the EJB module.
2. Right-click the EJB module and select **Generate-> Deploy and RMIC code** from the menu as shown in Figure 9-19.
3. Select the Stock check box and click Finish as shown in Figure 9-20.

4. Switch back to the Navigator view in the J2EE perspective and open your project. Under the ejbModule folder, you can see the deployed code for the Stock EJB.

Testing the Stock entity EJB

Now that we have generated the deployed code for the Stock bean, we can test it. We test it in the Application Developer WebSphere Test Environment.
1. Switch to the Server perspective. If you do not have the icon on the toolbar, select the **Open perspective** button and select the **Server** perspective.

Once the server perspective is open, the window looks like the example shown in Figure 9-21.

![Server perspective](image)

**Figure 9-21**  Server perspective

2. Right-click the **WebSphere Administrative Domain** under Server Configurations and select **Add Project**. Select the project as shown in Figure 9-22.

![Adding the project to the server configuration](image)

**Figure 9-22**  Adding the project to the server configuration

You can now see the project listed under the WebSphere Administrative Domain.

3. Double-click **WebSphere Administrative Domain**. Application Developer opens the server configuration file in the editor view.

4. Select the **Data source** tab as shown in Figure 9-23.
5. Click **Add** next to the **JDBC driver list** box.

6. Enter the parameters as shown in Figure 9-24, where the full name for the Implementation class name field is `com.ibm.as400.access.AS400JDBCConnectionPoolDataSource`.

7. Specify the folder where the IBM Toolbox for Java jt400.jar file is stored in the Class path field.

8. Click **OK**.

9. Click **Add** next to the Data Source defined in the JDBC driver selected above box.

10. Fill in the fields as shown in Figure 9-25. Make sure that the JNDI name is the same name as that defined in “Binding the Stock EJB to a JDBC data source” on page 277.
11. Click **OK**.

12. We need to specify the target system where the database tables are located. Click **Add** next to the Resource properties defined in the data source selected above box.

13. Fill in the parameters as shown in Figure 9-26. Replace the value field with your iSeries server name.

14. Click **OK**. Save changes by pressing Ctrl-S.

**Note:** In this example, we use the iSeries library specified in the user ID value of the data source. If you want to use a different library, you can specify a second resource property. The name of the property is **libraries**, and the value contains the library to use.

15. Start the server by selecting **WebSphere v4.0 Test Environment** in the Server Control Panel View and right-click. Select **Start** from the menu, or you can click the run icon as shown in Figure 9-27.
It takes some time to start so please be patient. Look for the Server Default Server open for e-business message in the console view or a started status in the server control panel view.

16. Switch back to the J2EE perspective and select the project under EJB modules. Right-click and select Run on Server.

17. The EJB Test Client opens. The first step is to see if the Stock bean has been added to the JNDI namespace. Click the JNDI Explorer link on the EJB Test Client as shown Figure 9-28.

In the EJB Test Client, you see the JNDI names as shown in Figure 9-29.
18. You can see StockHome in the namespace. Click the **StockHome** link. This invokes the EJB in the test client as shown in Figure 9-30.

19. Expand the **EJB references** tree until you see the methods available on the home interface (**create** and **findByPrimaryKey**).

20. Click the **findByPrimaryKey** method. You see the **findByPrimaryKey** method open in the parameter pane in the right pane as shown in Figure 9-31.
21. On the parameter frame, click the **Constructor drop-down** list and select `com.ibm.itso.roch.wasaejb.StockKey(java.lang.String,java.lang.String)` as shown in Figure 9-32.

22. This link invokes a page that prompts for the key constructor parameter. Enter 0001 for `stwid` (stock warehouse ID) and 000001 for `stiid` (stock item ID). This forms the key of an existing row in the STOCK file.

23. Click the **Invoke and Return** button as shown in Figure 9-33.
24. In the resulting screen, click the **Invoke button** to complete the call on the `findByPrimaryKey` method.

25. In the results pane, you see an instance of the Stock remote interface returned from the server. Click the **Work with Object** button as shown in Figure 9-34.

26. The EJB remote interface matching the Stock Key value appears in the EJB reference tree in the left frame. Expand the **Stock remote interface** to display the methods available on the Stock EJB remote interface.

27. Click the **BigDecimal getStqty()** method as shown in Figure 9-35.
28. On the parameter pane, click the **Invoke** button.

The results pane displays the stock quantity for the stock key selected as shown in Figure 9-36.

29. We can repeat the same steps to test the other remote methods.

We have now tested the Stock entity EJB using the EJB container in the WebSphere Test Environment in Application Developer.
9.2 Top-down mapping

This approach assumes that you already have existing enterprise beans. In this approach, the enterprise bean design determines the database design.

After you finish defining your enterprise beans, you can generate a schema and map. They are used to create the database table used to persist the entity bean data.

A set of tables are generated to support the CMP entities inside the EJB project. In these tables, each column corresponds to a CMP field of the enterprise bean, and the generated mapping maps the field to the column. Relationship associations are mapped to foreign-key relationships.

In the previous section, you saw how easy it is to generate the CMP entity bean directly from the iSeries database table to which you want to map it. However, this technique has a couple of disadvantages:

- One property is generated in the entity bean for every field in the database file.
- The property names are exactly the same as the field names in the iSeries table.

Perform the following steps to perform top-down mapping:

1. Switch to the J2EE View of J2EE perspective, and create a new EJB project as explained in 9.1.1, “Creating an EJB project” on page 266. We name it ManualMap.
2. You can see the ManualMap module under the EJB Modules. Right-click EJB Module and select New-> Enterprise Bean from the menu as shown in Figure 9-37.

3. In the create EJB window complete these tasks. Use Figure 9-38 as a reference.
   a. Enter Stock in the Bean name field.
   b. Select ManualMap in the EJB project field.
   c. Select the Entity bean with container-managed persistence (CMP) fields radio button.
d. Type the default package name as com.ibm.itso.roch.wasaejb. The names for the home interface, remote interface, and key class are automatically generated.

Figure 9-38  Create Enterprise Bean wizard

e. Click Next to move to the Enterprise Bean Details window as shown in Figure 9-39.

f. Click the Add button next to CMP attributes.

Figure 9-39  Create Enterprise Bean wizard

4. The create CMP attribute window appears as shown Figure 9-40. The key to the Stock table is composed of warehouse ID and item ID. First, we create a field for the warehouse ID. Enter a meaningful name for the warehouse ID field in the Field Name field.
5. Because the warehouse ID field type is of type CHAR, select `java.lang.String` in the Type field drop-down list. Because it is part of the STOCK file key, select the **Key field** check box. Click the **Apply** button and continue adding the other fields.

![Figure 9-40 Creating persistent fields](image)

6. Repeat the same process for the other character key field, item ID.

7. As explained in 9.1, “Bottom-up mapping” on page 266, the automatic generation of an entity bean from an iSeries table takes care of the mapping between database types and Java types. In our new example, we control the mapping. Refer to the JDBC documentation for a correct mapping. Table 9-1 shows the mapping of the most common database types.

<table>
<thead>
<tr>
<th>iSeries type</th>
<th>SQL type</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>CHAR</td>
<td><code>java.sql.String</code></td>
</tr>
<tr>
<td>PACKED</td>
<td>DECIMAL</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>ZONED</td>
<td>NUMERIC</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td><code>java.sql.Date</code></td>
</tr>
<tr>
<td>BINARY</td>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>FLTDBL</td>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>FLTDBL</td>
<td>DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

8. Add the quantity and year-to-date fields. Refer to Figure 9-41 for an example.

![Figure 9-41 Creating a quantity CMP field](image)
a. For the quantity and yearToDate field, enter `java.math.BigDecimal` in the Type field drop-down list. You can use the **Browse**... button to help find the class you need. The mapping database fields are of type PACKED.

b. Deselect the **Key Field** check box.

c. Select the **Access with getter and setter methods** check box. The setter and getter methods are then automatically generated in the StockBean class.

d. Optionally, you can select the **Promote getter and setter methods to remote interface** check box to have them in the Stock remote interface as well.

9. When you finish adding fields to the Stock entity bean, click **Cancel** on the create CMP attributes window and then click **Finish** on the create EJB window as shown in Figure 9-42.

![Figure 9-42 Creating the Stock enterprise bean](image)

Once the Stock bean is created, you can see the implementation of its classes and methods as shown in Figure 9-43.
9.2.1 Adding methods to the Stock bean

To add a user-defined method, follow these steps:

1. Switch to the J2EE View in the J2EE perspective. Drill down the tree to see the Stock bean under the ManualMap module.

2. Expand the StockBean to display StockBean, StockHome, Stock, and StockKey as shown in Figure 9-44.

3. Right-click StockBean and select Open With-> Enterprise Bean Java Editor.

4. To add a user-defined method, enter the code in the Enterprise bean Java editor. To add an increaseStockQuantity method, add the code shown in Figure 9-45.

Note the meaningful names for the fields, unlike the bottom-up mapping example.
public void increaseStockQuantity(int qty) throws javax.ejb.EJBException{
    quantity= new java.math.BigDecimal(quantity.intValue() + qty);
}

Figure 9-45 The increaseStockQuantity() method code

5. Declare that the method throws a javax.ejb.EJBException exception. Keep in mind that
   BigDecimal is not the same type as int, so you must convert it before using it.

6. After adding a method, right-click anywhere in the editor and click Save. Add the methods
to the remote interface by right-clicking the method in outline view and selecting
   Enterprise Bean-> Promote to Remote Interface as shown in Figure 9-46.

   Figure 9-46 Promoting a method to the remote interface

7. Add another method to decrease the quantity, named decreaseStockQuantity, using the
   same technique.

8. Close the Stock bean and save your work.

### 9.2.2 Mapping the Stock bean

Before you deploy the Stock bean, you must create a mapping and schema. To do this follow
these steps:

1. Switch back to the J2EE view of the J2EE perspective. Right-click ManualMap under EJB
   modules and select Generate-> EJB to RDB mapping.

2. The Creating a new EJB/RDB mapping window appears. Select the Top-down radio
   button and select the Open mapping editor after completion check box as shown in
   Figure 9-47. Click Next.
3. In the EJB/RDB Mapping dialog, select **DB2 UDB for iSeries, V5R1** for Target database. Enter the Database name and Schema name that was imported earlier in 9.1.2, “Importing a schema” on page 266. This is shown in Figure 9-48.
4. Click **Finish** to generate the mapping. Spend some time investigating the Map.mapxmi files to see whether the mapping is correct. Also navigate to see the files generated as explained in 9.1.3, “Creating the entity EJB from the imported schema” on page 272.

5. After the schema is generated, in the Navigator view of the J2EE perspective, expand the tree structure until you find the *.tblxmi file for the schema in the ManualMap project. This is the schema generated by top-down mapping.

6. Double-click the *.tblxmi file to open it in the table editor as shown in Figure 9-49.
If you want to change the column name or types, you can do so in the table editor.

7. After any change, save your work by pressing Ctrl-S.

8. In the Navigator view, locate the **Table.ddl** file. This file contains the generated data definition language (DDL) statements that create the table on the iSeries server. The DDL source is shown in Figure 9-50.

![Figure 9-50](image.png)

9. Copy the script from the Table.ddl file and run it on the iSeries server to create the table and schema. You can do this with interactive SQL or with Operations Navigator.

10. You are now ready to test. Before testing, you need to point the data source to the new schema. If you are working on an existing schema, check to ensure that the data source in the WebSphere Test Environment points to your schema.

### 9.2.3 Deploying the stock bean and testing

To deploy and test the stock bean created by manual mapping follow the steps shown in 9.1.5, “Deploying the Stock bean in the WebSphere Test Environment” on page 277.

### 9.3 Meet-in-the-middle mapping

In the meet-in-the-middle approach of mapping enterprise beans to database tables, it is assumed that you have existing enterprise beans and a database schema. You can perform meet-in-the-middle mapping by matching by name, by type, or no matching.

In this approach, you map each field of the enterprise bean to the corresponding column of a table within the selected schema. This technique provides an alternative technique to map CMP entity beans to existing database tables. The objective is to create a Stock entity bean whose properties map to only *some* of the iSeries STOCK table columns and use meaningful property names.

Manual mapping is generally the preferred way to create CMP entity beans because it gives the developer the most control over how the object-relational mapping takes place.

#### 9.3.1 Creating the stock enterprise bean

In this approach, we first create the Stock Enterprise bean and to add the methods to the bean. To create the Stock Enterprise bean, follow these steps:

1. Switch to the J2EE View of the J2EE perspective, and create a new EJB project as explained in 9.1.1, “Creating an EJB project” on page 266. We name it *MiddleMeet*.
2. You can see the MiddleMeet module under the EJB Modules. Right-click EJB Module and select New-> Enterprise Bean from the menu as shown in Figure 9-51.

![Figure 9-51 Creating an enterprise bean](image)

3. In the create EJB window complete these tasks.
   a. Enter **Stock** in the Bean name field.
   b. Select **MiddleMeet** in the EJB project field.
   c. Select **Entity bean with container-managed persistence (CMP) fields** in the Beantype drop-down list.
   d. Enter the default package name as `com.ibm.itso.roch.wasaejb`. The names for the home interface, remote interface, and key class are automatically generated.
   e. Click **Next** to move to the next screen.
   f. Click the **Add** button next to the CMP attributes fields of the bean as shown in Figure 9-52.
4. The create CMP Attribute window appears as shown Figure 9-53. Enter a meaningful name for the Warehouse ID field in the Field Name field.

5. Because the warehouse ID field type in the Stock table is of type CHAR, select `java.lang.String` in the Type field drop-down list. Because it is part of the STOCK file key, select the Key field check box. Click the Apply button and continue adding the other fields.

6. Repeat the same process for the other character key field, item id. The item ID field in the Stock file is of type CHAR. Select `java.lang.String` in the Field type drop-down list. Select the Key field check box.

7. As explained in 9.1, “Bottom-up mapping” on page 266, the automatic generation of an entity bean from an iSeries table takes care of the mapping between database types and Java types. In our new example, we control the mapping. Refer to the JDBC documentation for a correct mapping. Table 9-1 on page 290 shows the mapping of the most common database types.

8. Add the rest of the fields to the Stock bean. They are not primary key fields, so deselect the Key field check box. The fields to add are shown in Table 9-2.
Table 9-2  Stock bean fields

<table>
<thead>
<tr>
<th>Description</th>
<th>Table column</th>
<th>Bean field name</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Quantity</td>
<td>STORDRS</td>
<td>orderQuantity</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>Remaining Order</td>
<td>STREMORD</td>
<td>remOrderQuantity</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>Quantity</td>
<td>STQTY</td>
<td>stockQuantity</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>Year to Date</td>
<td>STYTD</td>
<td>yearToDate</td>
<td>BigDecimal</td>
</tr>
</tbody>
</table>

Figure 9-54 shows an example of adding the stockQuantity field.

Figure 9-54  Creating the stockQuantity CMP field

a. For the all the fields, select `java.math.BigDecimal` in the Field type drop-down list. You can use the Browse... button to help find the class you need. The mapping database fields are of type PACKED.

b. Deselect the Key Field check box.

c. Select the Access with getter and setter methods check box. The setter and getter methods are then automatically generated in the StockBean class.

d. Optionally, you can select the Promote getter and setter methods to remote interface check box to have them in the Stock remote interface as well.

9. When you finish adding the fields to the Stock entity bean, click Cancel on the create persistent field window and then click the Finish button on the Create an Enterprise Bean window as shown in Figure 9-55.
Once the Stock bean is created, you can see the implementation of its classes and methods as shown in Figure 9-56.

Note the meaningful names for the fields unlike in bottom-up mapping. Once the bean is created, we are ready to map the bean to the schema.
9.3.2 Adding the methods

To add the user defined methods, follow the steps as explained in “Adding methods to the Stock bean” on page 292. In this section, we create more user-defined methods:

1. We add the following methods:
   - void decreaseRemOrderQuantity(int qty)
   - void decreaseStockQuantity(int qty)
   - void decreaseOrderQuantity(int qty)
   - void decreaseYearToDate(int qty)
   - void increaseOrderQuantity(int qty)
   - void increaseRemOrderQuantity(int qty)
   - void increaseStockQuantity(int qty)
   - void increaseYearToDate(int qty)

   Figure 9-57 shows a code example of how to write the logic for the methods.

   ```java
   public void decreaseOrderQuantity(int qty) throws javax.ejb.EJBException{
       orderQuantity = new java.math.BigDecimal(orderQuantity.intValue() - qty);
   }
   ```

   Figure 9-57  decreaseOrderQuantity() method logic

2. After you add a method, right-click anywhere on the editor and click Save. Add the methods to the remote interface by right-clicking the method in outline view and selecting Enterprise Bean-> Promote to Remote Interface.

   After the methods are added to the remote interface of the bean, the outline view looks like the example in Figure 9-58.

   ![Figure 9-58  Outline view of the Stock bean](image)
9.3.3 Mapping the stock bean

In the meet-in-the-middle approach of mapping enterprise beans to database tables, it is assumed that you have existing enterprise beans and a database schema. We now have the enterprise bean. To create the mapping, follow these steps:

1. The first action is to import the schema of the Stock table from the iSeries database table into your working project as explained in 9.1.2, “Importing a schema” on page 266. Once the schema is imported, the enterprise been and schema are ready.

2. Switch back to the J2EE view of the J2EE perspective. Select the project under EJB modules and select Generate-> EJB to RDB mapping.

3. The Create a new EJB/RDB Mapping window (Figure 9-59) appears. Select the Meet In The Middle radio button and select the Open mapping editor after completion check box. Click Next.

4. On the window (Figure 9-60), select the Match By Name, and Type radio button and click Finish.
Once the mapping is generated, you can see that no fields are mapped as shown in Figure 9-61.

5. We need to manually map the existing fields in the Stock enterprise bean to the columns in the Stock database schema. We follow these steps:
   a. Select **MiddleMeet (Project Name)** in the lower pane of the map editor and click the **Show both mapped and unmapped objects** icon (Figure 9-62).
b. A plus (+) sign appears next to the project name in the lower pane. Click it to expand the tree to view the enterprise bean and all the fields as shown in Figure 9-63.

c. Select the Stock bean in the lower left pane and select the schema in the lower right pane.

d. Select the warehouseId field of the bean on the left side. Click the drop-down list and select the STWID column from the schema as shown in Figure 9-64.
e. Repeat these steps to map the other fields listed in Table 9-3.

Table 9-3  Stock bean to Stock table mapping

<table>
<thead>
<tr>
<th>Stock bean field</th>
<th>Stock table field</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemId</td>
<td>STIID</td>
</tr>
<tr>
<td>stockQuantity</td>
<td>STQTY</td>
</tr>
<tr>
<td>yearToDate</td>
<td>STYTD</td>
</tr>
<tr>
<td>orderQuantity</td>
<td>STORDRS</td>
</tr>
<tr>
<td>remOrderQuantity</td>
<td>STREMORD</td>
</tr>
</tbody>
</table>

Once all the fields are mapped, the map editor looks like the example in Figure 9-65.

6. Be aware that iSeries Stock table fields that are not mapped to Stock bean properties must be null capable fields. Otherwise, inserts through the Stock bean create() method fail. Add the following code to the ejbCreate() method:
stockQuantity = new java.math.BigDecimal(0.0);
yearToDate = new java.math.BigDecimal(0.0);
orderQuantity = new java.math.BigDecimal(0.0);
remOrderQuantity = new java.math.BigDecimal(0.0);

7. Save the changes by selecting **File-> Save Map.mapxmi.**

9.3.4 Deploying and testing the enterprise bean

To deploy the Stock bean created by manual mapping, follow the steps shown in 9.1.5, “Deploying the Stock bean in the WebSphere Test Environment” on page 277. Beside testing the `getStockQuantity()` method as discussed earlier, invoke the other new methods added in meet-in-the-middle mapping:

1. Once the `getStockQuantity` method returns the stock quantity value, select the `increaseStockQuantity()` method to increase the stock quantity in the iSeries Stock table through the CMP Stock entity bean.

2. In the parameters pane, enter a value of the quantity to be increased and click the **Invoke** button as shown Figure 9-66.

3. Invoke the `getStockQuantity()` method to see that the increase to stock quantity has occurred as shown in Figure 9-67.
4. You can also query iSeries STOCK table using interactive SQL or Operations Navigator and see if there is an increase in the stock quantity.

5. Repeat the same procedure to try the other methods.

9.4 Developing a bean-managed persistence (BMP) entity bean

After discussing all the possible alternatives to develop and deploy a container-managed persistence entity bean, let's look at bean-managed persistence entity beans. You may want to develop BMP entity beans for two main reasons:

- Performance
- Complex entity/database mapping

BMP application development is more complicated. Application Developer helps you create BMP beans, but you must handle persistence in the code.

In this example, we use an existing project that we created earlier called ItsoEjb. Once the working project is ready, follow these steps:

1. Switch to the J2EE view in the J2EE perspective. Right-click the ItsoEjb project and select New-> Enterprise Bean as shown in Figure 9-68.
2. In the create EJB window, complete these tasks. Use Figure 9-69 as a reference.
   a. Enter Customer in the Bean name field.
   b. Select ItsoEjb for the EJB project field.
   c. Select Entity bean with bean-managed persistence (BMP) fields from the Beantype drop-down list.
   d. Enter the default package name as com.ibm.itso.roch.wasaejb. The names for the home interface, remote interface, and key class are automatically generated.
   e. Click the Next button.
3. The EJB attribute window appears. Add the packages as shown in Figure 9-70. Click the **Add Type** and **Add Package** buttons next to Add import statements to the bean class to find the package and type you want to add.
4. Click the Finish button to generate the bean. At this point, it's worth spending some time to see the files generated. In the J2EE perspective, you see that CustomerHome, Customer and CustomerBean are automatically generated with all the methods required by the Enterprise JavaBeans specifications. This is shown in Figure 9-71.

![Figure 9-71 Customer bean and methods](image)

You still have the ability to add methods to the remote and home interfaces. However, no field, no key fields, and no field setter and getter methods are generated. They must be created manually by the developer. Just as creating methods to enforce business rules is a developer responsibility, so is entity data persistence.

5. We add the following code:

   a. We update the ejbCreate() method of the Customer Bean class to insert a new record in the iSeries Customer table. The ejbCreate() method is shown in Example 9-1.

   ![Example 9-1 ejbCreate code for customer CMP bean](image)
// Set Field Variables
ivCustID = custID;
ivFirstName = fN;
ivLastName = lN;
ivPhone = phone;
ivAddress = new Address();
isDirty = true;

// Use a JDBC connection to insert a row into the customer data base
Connection con = null;
Statement stmt = null;
try {
// get a connection from the dataSource, generate SQL to insert this
// row into the database.
con = ds.getConnection();
stmt = con.createStatement();
String stmtString = "insert into CSTMR (cid, cfirst, clast,
cphone) " + "values
("
+ ivCustID + ", 
+ ivFirstName
+ ", 
+ ivLastName + ", 
+ ivPhone + ")";
if (stmt.executeUpdate( stmtString ) != 1) {
throw new CreateException ("JDBC did not create a customer row");
}
// Create the primary key object and return it.
CustomerKey cID = new CustomerKey();
cID.primaryKey = ivCustID;
return cID;
} catch (CreateException ce) {
throw ce;
}
catch (SQLException sqe) {
throw new CreateException (sqe.getMessage());
}
finally {
try {
// close the statement and return the connection to the pool
stmt.close();
releaseConnection(con);
} catch (Exception ignore) {}
}
"
}

b. We update the ejbLoad() method to retrieve data from the Customer table and place it
in the bean properties. Example 9-2 shows the ejbLoad() method.

---

b. We update the ejbLoad() method to retrieve data from the Customer table and place it
in the bean properties. Example 9-2 shows the ejbLoad() method.

---

b. We update the ejbLoad() method to retrieve data from the Customer table and place it
in the bean properties. Example 9-2 shows the ejbLoad() method.

---

Example 9-2   ejbLoad() code for customer CMP bean

```java
public void ejbLoad() throws javax.ejb.EJBException {
try {
    InitialContext initCtx = new InitialContext();
    String dataSourceName = (String)initCtx.lookup("java:comp/env/dataSourceName");
    ds = (DataSource) initCtx.lookup("jdbc/" + dataSourceName);
}
catch (Exception e) {
    throw new EJBException("Error in ejbCreate: " + e.getMessage());
}
try {
    // use the "refresh" method to populate the fields.
```
c. We update the `ejbStore()` method to update a row in the Customer table from the bean’s properties. Example 9-3 shows the `ejbStore` method.

**Example 9-3  `ejbStore()` code for customer bean**

```java
public void ejbStore() throws javax.ejb.EJBException {
    Connection con = null;
    Statement stmt = null;
    try {
        // Use JDBC to update a row in the data base
        con = ds.getConnection();
        stmt = con.createStatement();
        // modify the statement below for proper SQL
        String statement = "update CSTMR set";
        if (ivFirstName != null) {
            statement += " cfirst = '" + ivFirstName + "'";
        }
        if (ivLastName != null) {
            statement += " , clast = '" + ivLastName + "'";
        }
        if (ivInits != null) {
            statement += " , cinit = '" + ivInits + "'";
        }
        if (ivPhone != null) {
            statement += " , cphone = '" + ivPhone + "'");
        }
        if (ivAddress.ivAddressLine1 != null) {
            statement += " , caddr1 = '" + ivAddress.ivAddressLine1 + "'");
        }
        if (ivAddress.ivAddressLine2 != null) {
            statement += " , caddr2 = '" + ivAddress.ivAddressLine2 + "'");
        }
        if (ivAddress.ivCity != null) {
            statement += " , ccity = '" + ivAddress.ivCity + "'");
        }
        if (ivAddress.ivState != null) {
            statement += " , cstate = '" + ivAddress.ivState + "'");
        }
        if (ivAddress.ivZip != null) {
            statement += " , czip = '" + ivAddress.ivZip + "'");
        }
        if (ivCreditLimit != 0) {
            statement += " , ccreditLimit = " + ivCreditLimit;
        }
        if (ivBalance != 0) {
            statement += " , cbalance = " + ivBalance;
        }
        if (ivYearToDateBalance != 0) {
            statement += " , cyearToDateBalance = " + ivYearToDateBalance;
        }
        statement += " where cid = '" + ivCustID + "'");
        i = stmt.executeUpdate(statement);
```
if (i == 0) {
    throw new EJBException("ejbStore: CustomerBean (" + ivCustID + ") not updated");
}
} catch (EJBException re) {
    throw re;
} catch (SQLException sqe) {
    throw new EJBException(sqe.getMessage());
} finally {
    try {
        stmt.close();
        releaseConnection(con);
    } catch (Exception ignore) {
    }
}  

d. We update the **ejbRemove()** method to delete a row from the Customer table. Example 9-4 shows the **ejbRemove()** method.

---

**Example 9-4  ejbRemove() code for customer bean**

```java
public void ejbRemove() throws java.rmi.RemoteException, javax.ejb.RemoveException {
    Connection con = null;
    Statement stmt = null;
    try {
        // Use JDBC to delete the row from the data base. HINT: the primary key is in the EntityContext.
        con = ds.getConnection();
        CustomerKey pk = (CustomerKey) getEntityContext().getPrimaryKey();
        stmt = con.createStatement();
        int i = stmt.executeUpdate("delete from CSTMR where cid = " + pk.primaryKey + ");
        if (i == 0) {
            throw new EJBException("CustomerBean " + pk.primaryKey + " not found");
        }
    } catch (EJBException re) {
        throw re;
    } catch (SQLException sqe) {
        throw new RemoteException(sqe.getMessage());
    } finally {
        try {
            stmt.close();
            releaseConnection(con);
        } catch (Exception ignore) {
        }
    }
```
9.4.1 Testing the BMP bean

Testing the BMP is the same as testing the CMP beans. We follow these steps:

1. Generate the deployed code for the Customer Bean using Application Developer. Start the WebSphere Test Environment and test the Customer Bean as explained in 9.1.5, “Deploying the Stock bean in the WebSphere Test Environment” on page 277.

2. Invoke the `create(String, String, String, String)` method to create a record in Customer database table on the iSeries server.

3. Create a new customer record with the following value:
   
   2204 Dean Ascheman 5233726

4. Return to the home interface in the test client. Retrieve the row just added using the finder and getter methods.

5. Close the test client.

9.5 Conclusion

In this chapter, we discussed the different application development scenarios with respect to container-managed persistence and bean-managed persistence entity JavaBeans.

We covered the following application development scenarios:

- Bottom-up CMP entity bean mapping
- Top-down CMP entity bean mapping
- Meet-in-the-middle CMP entity bean mapping
- Developing a BMP entity bean
Building Java applications with Enterprise JavaBeans

This chapter discusses building iSeries Java applications using Enterprise JavaBeans. We use the Enterprise JavaBeans described in Chapter 8, “iSeries EJB application development scenario” on page 245, to provide access to iSeries resources. In this chapter, we go through the following types of application development scenarios:

- Java applications
- Java servlets

We begin by developing a simple HelloWorld EJB. Then we develop a servlet and a Java client application which use the HelloWorld bean. Next we develop the OrderEntry application, which uses the entity and session beans developed earlier.
10.1 Developing the HelloWorld EJB application

In this section, we start by developing the HelloWorld Enterprise JavaBean using Application Developer. We then test it using the WebSphere Test Environment under Application Developer. The objective of this section is to take you through the complete development of an EJB session bean and the development of a Java client application and servlet that use the HelloWorld EJB.

In this section, we:
- Use Application Developer to create the HelloWorld session bean.
- Test the HelloWorld session bean inside Application Developer.
- Develop a Java client application which uses the HelloWorld bean.
- Develop a servlet which uses the HelloWorld bean.
- Install the Enterprise application under WebSphere Application Server on the iSeries server.
- Test the enterprise application running on the iSeries server.

10.1.1 Creating the HelloWorld bean in Application Developer

We begin our discussion by creating a simple HelloWorld EJB session bean. To do this, we follow these steps:

1. In the J2EE view of the J2EE perspective, select File->New->EJB Project.
2. Enter HelloWorldTest as the project name.
3. Type or select the enterprise application project name to which this EJB project will be added as a module. If the enterprise application does not exist, it is created.
4. Click Next. You can also click Finish at this point to create a default EJB project if you have no module dependencies or build settings to specify.
5. Under the EJB modules in the J2EE perspective, select your working project. In this case we use HelloWorldTest as our project.
6. Right-click the project and select New-> Enterprise Bean.
7. In the Create an Enterprise bean window, enter the information as shown in Table 10-1.

Table 10-1 Creating an enterprise bean values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bean Name</td>
<td>HelloWorld</td>
</tr>
<tr>
<td>EJB Project</td>
<td>HelloWorldTest</td>
</tr>
<tr>
<td>Bean type</td>
<td>Session Bean</td>
</tr>
</tbody>
</table>

If you want to place the EJB in a unique package, you define the package as shown in Figure 10-1. Otherwise, the Default package of “root” is used.
8. Click **Next** to display the Enterprise Bean Details window as shown in Figure 10-2.

9. Click the **Finish** button to create the HelloWorld bean. It’s worth spending some time to navigate the files generated by Application Developer as shown in Figure 10-3.
Figure 10-3  J2EE/Navigator view of the HelloWorld bean

Notice what Application Developer has generated:

– HelloWorldHome is the home interface for the bean.
– HelloWorld is the remote interface.
– HelloWorldBean is the implementation of the bean.

Also in the Navigator view, you can see the .class files and .java files generated under the package defined while creating the bean.

10. We write a business method that returns a string that reads “Hello from ITSO”. To add a method, double-click HelloWorldBean.java to open it. Enter the code shown in Example 10-1 at the end of the file, before the final “}”.

Example 10-1  The printHello method

```java
public String printHello() throws javax.ejb.EJBException{
    return("Hello from ITSO");
}
```

11. After you add the method, save the file by selecting File-> Save-> HelloWorldBean.java from the menu.

12. Add the following import statement to both the HelloWorldBean.java and HelloWorld.java files:

```java
import javax.ejb.EJBException;
```

13. Add the printHello() method to the remote interface of the bean by right-clicking the method in the outline view. Then select Enterprise Bean-> Promote to Remote Interface as shown in Figure 10-4.
14. While in the Navigator view, expand the tree for the MyHelloWorld project to see the `ejb-jar.xml` file under the META-INF folder. Double-click the `ejb-jar.xml` file. This is the deployment descriptor.

15. In the deployment descriptor, click the Beans tab and select the HelloWorld bean. Ensure the bean is a Stateless session bean as shown in Figure 10-5.

16. Press Ctrl-S to save the settings, if necessary.

17. To register HelloWorld in the JNDI namespace, switch to the J2EE view. Select the EJB project. Then right-click and select **Open with-> EJB Extension Editor**.

18. In the EJB Extension Editor, click the Bindings tab and select the HelloWorld bean under HelloWorldTest. Enter HelloWorld in the JNDI name field on the right side of the page as shown in Figure 10-6.
19. Save the changes by selecting **File > Save EJB Extension Editor**. You can also press Ctrl-S.

20. Generate the deployed code by right-clicking the **HelloWorld** bean in the J2EE perspective and selecting **Generate Deployed Code**.

21. To test the EJB, right-click the **HelloWorldTest** EJB project and select **Run on Server**.

   If the test server is not started, it is now started. Look for the “Server Default Server open for e-business” message in the console view or a started status in the server control panel view.

22. The EJB Test Client opens. The first step is to see if the HelloWorld EJB has been added to the JNDI namespace. Click the **JNDI Explorer** link on the EJB Test Client as shown Figure 10-7.

23. You can see HelloWorldHome in the namespace. Click the **HelloWorld** link to look up the EJB home for the selected item. Once the instance of the EJB home is found, the window shown in Figure 10-8 appears in the test client.
24. Expand the EJB references tree for the bean until you see the create methods available in the home interface.

25. Click the create method. You see the create method open in the parameter pane, on the right-hand side, as shown in Figure 10-9.

26. In the parameters frame, click the Invoke button to create an instance of the HelloWorld EJB.

27. Click the Work with Object button in the results pane. You see the remote interface of the bean under the EJB reference.

28. Expand the remote interface of the bean and select the printHello() method as shown in Figure 10-10.
29. In the parameters pane, click the **Invoke** button. Once the method runs, you see the output “Hello from ITSO” in the results pane as shown in Figure 10-11.

### 10.1.2 Creating a servlet that uses the EJB

The objective in this section is to create a servlet that uses the HelloWorld bean created in the previous section. To create a servlet that accesses the HelloWorld EJB, we use the following steps:

1. Switch to the **Web perspective**. Right-click anywhere and select **New->Web Project** as shown in Figure 10-12.
2. In the create a Web project window, enter the information as shown in Table 10-2.

Table 10-2 Creating the Web project

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name</td>
<td>MyHelloWorldWeb</td>
</tr>
<tr>
<td>Enterprise Application project name</td>
<td>HelloWorldTestApp (select the application which has the HelloWorld bean)</td>
</tr>
<tr>
<td>Context root</td>
<td>Hello</td>
</tr>
</tbody>
</table>

3. Click the Next button as shown in Figure 10-13.
4. In the Module Dependencies window, select the HelloWorldTest.jar check box and click **Finish** as shown in Figure 10-14. This updates the runtime classpath and the Java project build.

![Module Dependencies](image1)

**Figure 10-14  Module dependencies**

Once the Web project is created, the navigator view looks like the example in Figure 10-15.

![Generated files after the Web project is created](image2)

**Figure 10-15  Generated files after the Web project is created**

It’s worth spending some time to review the files generated by Application Developer. For Web application development, the key files managed in a Web project are:

- **web.xml**: The deployment descriptor for the Web module. This file contains general information on servlet mapping, security information, and the Web page.
- **ibm-web-bnd.xml**: The WebSphere bindings for the Web application. This file contains bindings to references used at the Web module level.
- **ibm-web-ext.xml**: This file is used by WebSphere to support additional options beyond the J2EE specification such as reloading intervals, the default error page, if file serving is allowed, and if servlet invoking are enabled in the Web module.

- **Master.css**: This is the default cascading style sheet that the Application Developer page designer uses for any new HTML or JSP pages.

- **.classpath**: Application Developer uses this file to store metadata about the build path for the project when compiling Java classes and executing Java applications. Do not edit this file directly. To change the path information, simply click the properties context menu for the project and modify the Java Build Path item.

5. Select the MyHelloWorldWeb project. Right-click and select New-> Other-> Web-> Servlet from the menu. Click Next.

6. The Create the Servlet Class window (Figure 10-16) appears. Enter the package name as tservlets and the servlet name as HelloTestServlet. Deselect the doPost check box. Click the Finish button.

![Figure 10-16  Creating the servlet class](image)

7. Double-click and open the HelloTestServlet.java file. Enter the code as shown in Example 10-2 in the doGet method.

**Example 10-2  The doGet method**

```java
PrintWriter out = response.getWriter();
HelloWorldHome home = null;
try {
    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
```
InitialContext initCtx = new InitialContext(p);
    // Look up the EJB
Object o = initCtx.lookup("java:comp/env/HelloWorld");
System.out.println(" -- lookup was successful");
home = (HelloWorldHome) javax.rmi.PortableRemoteObject.narrow(o,
    HelloWorldHome.class);
System.out.println(" -- narrow was successful");
HelloWorld myEJB = home.create();
System.out.println(" -- create was successful");
// call methods
String msg = myEJB.printHello();
System.out.println("Message from Hello EJB: " + msg);
out.println("Message from Hello EJB: " + msg);
// destroy ejb
myEJB.remove();
System.out.println("HelloServlet ran successfully.");
}
} catch (EJBException e) {
    out.println("remoteException " + e.getMessage());
e.printStackTrace();
} catch (NullPointerException e) {
    if (home == null)
        out.println("noHome " + e.getMessage());
    else
        e.printStackTrace();
} catch (Exception e) {
    out.println("generalException " + e.getMessage());
e.printStackTrace();
} finally {
    out.close();
}

8. Add the import statements shown in Example 10-3.

Example 10-3   Adding the import statements
import com.ibm.itso.roch.ejbtest.HelloWorld.*;
import java.util.*;
import javax.naming.*;
import java.rmi.RemoteException;
import java.io.*;

9. Review the servlet code. Here is what it does:
   a. Creates a Properties object to hold the values required to create an InitialContext object. The InitialContext object is actually the naming service. When it is instantiated, it uses the name of the service provider and the URL where it is located. In this example, we use the IBM WebSphere JNDI naming service.
   b. Creates an InitialContext object using the properties object that we created.
   c. After the InitialContext object is created, the home interface for the bean must be located. We accomplish this through a method call to the lookup method of the InitialContext object. The home interface for the bean is located by name. The application programmer only needs to know the name of the interface and the class type it returns.
   d. Because the InitialContext object returns any type that is registered with it, the application programmer must cast to the correct type. We use the static method
narrow() to cast the java.lang.Object returned by the lookup method to the correct home interface type. The narrow() method takes the object to be narrowed and the class of the EJB home object be returned as parameters.

e. We use the home interface to create an instance of the HelloWorld EJB, which we name myEJB.

f. The myEJB object can be used to call the methods defined in the Remote interface. We use it to call the printHello method.

g. We display the value returned by the printHello method in the browser. We use the out.println method to do this.

h. We remove the EJB.

10. Save the file by selecting File -> Save HelloTestServlet.java file. Make sure that you have no errors.

11. The notation java:comp/env defines that the lookup is to a local reference of HelloWorld instead of an entity in the global JNDI namespace. We map this using the reference definition in the deployment descriptor.

Figure 10-17 Editing the web.xml file

a. Switch to the Navigator view and select the web.xml file (Figure 10-17).

b. Double-click to open it in the editor.

c. Select the References tab.

d. Select the EJB References radio button and click Add to add a reference to the HelloWorld EJB.

e. Use the values shown in Table 10-3.

\[
\begin{array}{|l|l|}
\hline
\text{Name} & \text{Value} \\
\hline
\text{EJB Reference} & \text{HelloWorld} \\
\text{Type} & \text{Session} \\
\text{Home} & \text{com.ibm.itso.roch.ejbtest.HelloWorld.HelloWorldHome} \\
\text{Remote} & \text{com.ibm.itso.roch.ejbtest.HelloWorld.HelloWorld} \\
\text{JNDI Name} & \text{HelloWorld} \\
\hline
\end{array}
\]
f. Press Ctrl-S to save your work.

12. Switch back to the Web perspective. Select the **MyHelloWorldWeb** project, right-click, and select **Properties**.

13. Make sure that the EJB project is added in the classpath. If it is not, add it now as shown in Figure 10-18.

![Figure 10-18 Java Build Path](image)

14. Expand the MyHelloWorldWeb project by selecting **MyHelloWorldWeb->webApplication->WEB-INF-classes->tservlets**.

15. Right-click **HelloTestServlet.class** and select **Run on Server**.

The Web browser opens and displays the message, “Message from Hello EJB: Hello from ITSO.”

### 10.1.3 Creating a Java client application that uses the EJB

The objective in this section is to create a client application that uses the HelloWorld bean created in 10.1.1, “Creating the HelloWorld bean in Application Developer” on page 316. To create a Java application that accesses the HelloWorld EJB, we follow these steps:

1. Switch to the Java perspective. Right-click and create a new Java project named **MyHelloWorldClient**.

2. Right-click the project and select **New->Class**.

3. In the Java Class window, enter the package name and the class name as shown in Figure 10-19.
4. Select the **public static void main(String[] args)** check box. Click **Finish** to create the Java program.

5. Double-click to open the **HelloWorldClient.java** file and enter the code shown in Example 10-4 in the main method.

**Example 10-4  Client application logic**

```java
try {
    // Get the J2EE name space context
    Properties p = new Properties();
    p.put(
        Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
    String provider_url = args[0];
    p.put(Context.PROVIDER_URL, "iiop://" + provider_url);
    InitialContext initCtx = new InitialContext(p);
    // Look up the EJB
    System.out.println("Before lookup " + "url = " + provider_url);
    Object o = initCtx.lookup("HelloWorld");
    MyHelloWorldHome home = (HelloWorldHome) javax.rmi.PortableRemoteObject.narrow(o, HelloWorldHome.class);
    MyHelloWorld myEJB = home.create();
    // Retrieve the text message from the EJB
    String msg = myEJB.printHello();
    System.out.println();
    System.out.println("Message from Hello EJB: " + msg);
    System.out.println();
    System.out.println("HelloClient ran successfully.");
} 
catch (Throwable t) {
    t.printStackTrace();
}
```
6. Add the import statements shown in Example 10-5. The code is similar to the servlet created in 10.1.2, “Creating a servlet that uses the EJB” on page 322.

Example 10-5   Adding import statements

```java
import com.ibm.itso.roch.ejbtest.HelloWorld.*;
import java.util.*;
import javax.naming.*;
```

7. Save the work. You still see some errors, but we can resolve them by setting properties for the Java build.

8. Right-click **MyHelloWorldClient** and select **Properties**.

9. We need to set the Java Build Path for the client application. Select **Java Build Path** in the properties window. In the right pane, select the **Projects** tab and select the EJB project as shown in Figure 10-20.

![Figure 10-20   Java Build Path](image)

10. Click the **Libraries** tab and select **Add External Jars**.

11. Locate the **j2ee.jar** file and add it to the libraries list. It is found in the `<AD install>\plugins\com.ibm.etools.websphere.runtime\lib` directory.

12. Select **Launcher**. In the right pane, select **Java Application** from the drop-down list.

13. Click **OK** to close the Properties dialog and save your changes.

14. Right-click **HelloWorldClient.java** and select **Properties**.
15. Type localhost in the Program Arguments text box as shown in Figure 10-21.

![Properties for the client application](image)

**Figure 10-21  Properties for the client application**

16. Click **OK** and save the work.

**Attention:** There are two Java Runtime Environments (JRE) that come with Application Developer. The Workbench is run by the one found in `<install-dir>\jre`. A second JRE is provided by WebSphere that is a prerequisite to run WebSphere applications. Java projects and J2EE application client projects use the workbench's JRE by default. You get the `java.lang.NoClassDefFoundError: com/ibm/rmi/iiop/GIOPVersionException` exception when you try to obtain the InitialContext.

You need to switch the JRE of your Java projects or J2EE application client projects to use WebSphere's JRE:

a. Open the **Window-> Preference** dialog. Under the Java section, select **Installed JREs**.

b. Add a new JRE. The JRE home directory is `<install-dir>\plugins\com.ibm.etools.server.jdk\jre`.

c. Open the Properties dialog of the project. Select JRE. Select **Use custom JRE for launching** and the newly added JRE.

17. Make sure the server is running. Click the **Run** button on the toolbar.

18. Select **Java Application**.

19. Application Developer switches to the Debug view. You see the following messages in the Console view:

```
Running the J2EE Application Client HelloEJB Sample
Before lookup url = localhost
-- lookup was successful
-- narrow was successful
-- create was successful
```
10.2 Building Java applications with Application Developer

In this section, we use Application Developer to build a Java application similar to the RPG OrderEntry application discussed in Chapter 7, “Overview of the OrderEntry application” on page 231. To make it easy to reuse the classes that we build for our application, we separate the access to the iSeries server from the graphical user interface. We create a package named Support, which contains a number of support classes.

Figure 10-22 shows the classes contained in the Support package. The ItemsDb class is responsible for all access to the iSeries server. It uses the Enterprise JavaBeans described in Chapter 8, “iSeries EJB application development scenario” on page 245, to access the iSeries server.

![Figure 10-22 The Support package](image)

The following list shows the other Support package classes that are supporting classes for applications:

- **Customer**: An object-oriented representation of a Customer table row
- **Item**: An object-oriented representation of an Item table row
- **Order**: An object-oriented representation of an Order table row
- **OrderDetail**: An object-oriented representation of an OrderLine table row

10.2.1 The ItemsDb class

This section investigates the ItemsDb class. It provides a level of abstraction for client applications. The goal is to allow client applications to use the Enterprise JavaBeans without having to deal with the implementation details and complexity. We look at how the ItemsDb class accesses the iSeries resources. The following key methods are provided:

- The `getInitialContext` method creates an InitialContext object.
- The `connect` method establishes a connection to the Java server.
- The `getAllCustomers` method retrieves a list of customers from the server.
- The `getItems` method retrieves a list of items from the server.
The `findRangeOfItems` method retrieves a subset of items from the server.

The `verifyCustomer` method verifies that a customer number is valid.

The `confirmOrder` method places and confirms an order.

The `submitOrder` method places and confirms an order using a shopping cart.

The `connectStateless` method establishes a stateless connection to the Java server for servlets.

The `submitOrderStateless` method places an order given a customer ID and a shopping cart.

All the methods in the `ItemsDb` class, except `connectStateless` and `submitOrderStateless`, use the `OrderEntryClerk` session Enterprise JavaBean to access iSeries resources. The stateless methods use the `OrderPlacement` stateless session EJB.

Our intent is to create an access class that is capable of running on its own. That is, you can use it even without any user interfaces. As shown in Figure 10-23, the advantage of this approach is that you can use the access class in any context. For example, you can use a graphical user interface (GUI) on top of the access class when you want to create a stand-alone Java application. You also use exactly the same class when you want to develop a Java servlet. In this case, replace the GUI frontend with a layer that is capable of running as a servlet. You can also implement a distributed application where some parts of the application run on a server. Our implementation using Enterprise JavaBeans is an example of this. When we call the methods of the Enterprise JavaBeans, we are actually running code on the server.

Separating the layers properly and using a clean and well-defined interface between the layers is important if you want to take advantage of object orientation.

To use Enterprise JavaBeans in an application, you must be able to access them. The way a client program accesses enterprise beans is defined by two interfaces:

- **Home interface**
- **Remote (EJBObject) interface**

The home interface contains methods that describe how you can instantiate an enterprise bean object. The remote interface, by comparison, defines the methods of an enterprise bean that can be accessed by your user program. To access an enterprise bean, your user program goes through the following steps:

1. Obtains a context to the name server (name service context).
2. Looks up the home of the enterprise bean using the name service context.
3. Creates an enterprise bean instance from the enterprise bean home, which returns an enterprise bean proxy object.
4. Accesses the remote methods of the enterprise bean instance through an enterprise bean proxy object. Each call to the enterprise bean proxy object is a remote call that can throw an exception, such as `javax.ejb.EJBException`. 
Import statements

Example 10-6 shows the class description for the ItemsDb class. You must import classes from the javax.naming, javax.ejb, and java.rmi packages to use Enterprise JavaBeans. You must also import the interfaces for the Enterprise JavaBeans that you use. They are found in the com.ibm.itso.roch.wasejb package.

```java
import java.math.*;
import java.util.*;
import com.ibm.itso.roch.wasejb.*;
import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import java.rmi.Remote;
import java.util.*;
import java.io.*;
public class ItemsDb extends java.lang.Object {
```

The getInitialContext method

To use the Java server and the Enterprise JavaBeans, you need to access the Java server's Java Naming Directory Interfaces (JNDI) naming service. This is done by instantiating an InitialContext object. You use the getInitialContext method to create the InitialContext object. The code is shown in Example 10-7.

```java
Example 10-7   The getInitialContext method

public static Context getInitialContext() throws Exception {
    Properties p = new Properties();
    try {
        p.put(Context.INITIAL_CONTEXT_FACTORY, 
              "com.ibm.websphere.naming.WsnInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://" + 
               getSystemName() + ":" + getPort());
        System.out.println("port: "+getPort());
        System.out.println("user: "+getUserId());
        System.out.println("password: "+getPassword());
        System.out.println("system: "+getSystemName());
        InitialContext cx = new InitialContext(p);
        return cx;
    } catch (Exception e) {
        System.out.println("error creating Context " + e.getMessage());
        e.printStackTrace();
        throw (e);
    }
}
```

The InitialContext object is actually the naming service. When it is instantiated, it uses the name of the service provider and the URL where it is located. In this example, we use the IBM WebSphere Advanced Edition JNDI naming service. We pass in the name of the server and the TCP/IP port number in the URL. Optionally, you can use a user ID and password. It is important to notice that this is a user ID and password for the naming service provider, not for the system. This is not the same as the iSeries user profile security.
The connect method

The connect method is used to connect to the Java server. The connect method is shown in Example 10-8.

Example 10-8  The connect method

```java
public String connect() throws Exception {
    try{
        Context ctx = getInitialContext();
        java.lang.Object tempObject= ctx.lookup("OrderEntryClerk");
        OrderEntryClerkHome home =
            (OrderEntryClerkHome)javax.rmi.PortableRemoteObject.narrow(
                (org.omg.CORBA.Object) tempObject,
                com.ibm.itso.roch.wasaejb.OrderEntryClerkHome.class);
        clerk = (OrderEntryClerk) home.create();
    } catch (FinderException fe) {
        System.out.println("Could not find order clerk "+ fe.getMessage());
        return("Could not find order clerk " + fe.getMessage());
    } catch (Throwable t) {
        System.out.println("Could not connect " + t.getMessage());
        return("Could not connect " + t.getMessage());
    }
    System.out.println("Connected to "+ getSystemName() +":" + getPort());
    return "Connected to " + getSystemName() +":" + getPort();
}
```

You call the getInitialContext method to create an InitialContext object. After the InitialContext object is created, the home interface for the bean must be located. You accomplish this through a method call to the lookup method of the InitialContext object.

The home interface for the bean is located by name. The application programmer only needs to know the name of the interface and the class type that is returned. Because the InitialContext object returns any type that is registered with it, the application programmer must cast to the correct type.

You use the static narrow() method to cast the java.lang.Object returned by the lookup method to the correct home interface type. The narrow() method takes the object to be narrowed and the class of the EJB home object to be returned as parameters. We use the home interface to create an instance of the OrderEntryClerk EJB, which we name clerk. The clerk object knows how to do many things associated with the order entry system. For example, the clerk can perform the following tasks:

- Return a list of items from the Item table
- Return a list of customers from the Customer table
- Validate a customer number
- Create an order on the iSeries server

We use the OrderEntryClerk object named clerk to handle the application processing.

The getAllCustomers method

The getAllCustomers method retrieves all the rows from the iSeries Customer table. Example 10-9 shows the code.
Example 10-9  The getAllCustomers method

```java
public java.util.Vector getAllCustomers() {
    java.util.Vector tempCust = new java.util.Vector();
    try {
        tempCust = clerk.findAllCustomers();
    } catch (Exception e) {
        System.out.println("::::::::::::::: Unexpected Error :::::::::::::::");
        e.printStackTrace();
    }

    return tempCust;
}
```

To retrieve all customers, you call the OrderEntryClerk's findAllCustomers() method. It returns a vector containing information for all customers. The vector contains a String array element for each customer in the Customer table. The String array contains entries that correspond to the fields in the rows of the Customer table. You return the vector to the caller of this method.

The getItems method

The getItems method retrieves all rows from the iSeries Item table. Example 10-10 shows the code.

Example 10-10  The getItems method

```java
public java.util.Vector getItems() {
    try {
        ejbItems = clerk.findAllItems();
    } catch (Exception e) {
        System.out.println("::::::::::::::: Unexpected Error :::::::::::::::");
        e.printStackTrace();
    }

    System.out.println("all items returned");
    return ejbItems;
}
```

To retrieve all items, you call the OrderEntryClerk's getItems() method. It returns a vector containing all items. The vector contains a String array element for each item in the Item table. The String array contains entries that correspond to the fields in the rows of the Item table. You return the vector to the caller of this method.

The findRangeOfItems method

The findRangeOfItems method retrieves a subset of rows from the Items table. Example 10-11 shows the code.

Example 10-11  The findRangeOfItems method

```java
public java.util.Vector findRangeOfItems(String itemnoMin, String itemnoMax) {
    try {
        ejbItems = clerk.findRangeOfItems(itemnoMin,itemnoMax);
    } catch (Exception e) {
        System.out.println("::::::::::::::: Unexpected Error :::::::::::::::");
        e.printStackTrace();
    }
```
To retrieve a subset of items, you call the OrderEntryClerk's `findRangeOfItems` method. You pass it parameters containing the value of the first item and the value of the last item to return. It returns a vector containing all items within the range. You return the vector to the caller of this method.

**The `verifyCustomer` method**

The `verifyCustomer` method verifies that a customer number contains a valid row in the Customer table. Example 10-12 shows the code.

```java
public boolean verifyCustomer(String customerId) {
    boolean isValid = false;
    System.out.println("isValid = " + isValid);
    try {
        isValid = clerk.verifyCustomer(customerId);
        System.out.println("isValid = " + isValid);
        if (isValid)
            validCustomerId = customerId;
    } catch (Exception e) {
        return isValid;
    }
    return isValid;
}
```

To verify that a customer exists in the Customer table, you call the OrderEntryClerk's `verifyCustomer` method. You pass it a parameter containing the customer number. It returns a boolean that is true or false, depending on whether the customer is valid. You return the boolean to the caller of this method.

**The `submitOrder` method**

When you are ready to submit an order for processing on the iSeries server, you use the `ItemsDb submitOrder` method. It, in turn, uses methods provided by the OrderEntryClerk bean. Example 10-13 shows the code.

```java
public String submitOrder(Order anOrder) throws Exception {
    try {
        OrderDetail orderLine = null;
        //set the customer number
        clerk.setCustomer(anOrder.getCustomer().getId().toString());
        BigDecimal orderLineCount = new BigDecimal(anOrder.getOrderDetails().size());
        for (java.util.Enumeration itemE = anOrder.getOrderDetails().elements();
```
We use the following steps to submit an order:

1. Set the name of the customer using the setCustomer method.
2. Pass in as parameters, the item number and the quantity for each item you want to order using the addOrderLine method of the OrderEntryClerk.
3. Place the order using the placeOrder method of the OrderEntryClerk. It returns the order number of the order that is created.
4. Return the order number to the caller of this method.

The submitOrder methods demonstrates one of the key advantages of using Enterprise JavaBean technology. You only need to know what the bean does and how to interface to it. Do not be concerned with the details of how it accomplishes its task. If you want to implement the order processing logic in your application without using Enterprise JavaBeans, the application code is much more complex because you must provide the code that does the database access and order processing.

The stateless methods

To properly support servlet clients, we provide two stateless methods in the ItemsDb class that don’t use the OrderEntryClerk session bean at all — connectStateless and submitOrderStateless. The only constraint to using these methods is that the connectStateless method must be called before the submitOrderStateless method. These methods are similar to the connect and submitOrder methods, but use the OrderPlacement session bean (stateless) instead of the OrderEntryClerk session bean.

The connectStateless method (Example 10-14) provides a connection to the OrderPlacement session bean.

Example 10-14 The connectStateless method

```java
public String connectStateless() throws Exception {
    try {
        Context ctx = getInitialContext();
        java.lang.Object tempObject= ctx.lookup("java:comp/env/OrderPlacement");
        OrderPlacementHome home = (OrderPlacementHome)javax.rmi.PortableRemoteObject.narrow(
            (org.omg.CORBA.Object) tempObject,
            com.ibm.itso.roch.wasaejb.OrderPlacementHome.class);
        OrderPlacer = (OrderPlacement) home.create();
    } catch (FinderException fe) {
        System.out.println("Could not find order placer " + fe.getMessage());
        return ("Could not find order placer " + fe.getMessage());
    } catch (Throwable t){
```

```java
    orderLine = (OrderDetail) itemE.nextElement();
    clerk.addOrderLine(orderLine.getItemId(), orderLine.getQuantity());
}
// place the order
String orderNumber = clerk.placeOrder();
return orderNumber;
} catch (Exception e) {
    throw (e);
} }
t.printStackTrace();
    return ("Could not connect " + t.getMessage());
}
return "Connected to " + getSystemName() + ":" + getPort();
}

The submitOrderStateless method, shown in Example 10-15, is used by a client application to place an order. It calls the placeOrder method of the OrderPlacement session bean. The input parameters for this method are a string containing the customer ID and a vector containing the items to order. It returns the order number of the new order.

Example 10-15  The submitOrderStateless method

public int submitOrderStateless(String CustID, Vector OrderLines) throws Exception {
    int returnvalue;
    try {
        returnvalue = (int)OrderPlacer.placeOrder("0001", "1", CustID, OrderLines);
    } catch (Exception e) {
        throw (e);
    }
    return returnvalue;
}

10.2.2 Using the ItemsDb class

Now let's put it all together. First, we run the ItemsDb class using only scripts in the Application Developer Scrapbook. We create a new instance of the ItemsDb class and call its connect method. The ItemsDb class also provides methods to set values for the user ID, password, TCP/IP port, and system name.

Example 10-16  Testing the ItemsDb class in the Application Developer Scrapbook

Support.ItemsDb aItemsDb = new Support.ItemsDb();
aItemsDb.setUserId("cpwejb");
aItemsDb.setPassword("cpwejb400");
aItemsDb.setPort("900");
aItemsDb.setSystemName("localhost");
aItemsDb.connect();

To run the code (Example 10-16) in the Application Developer Scrapbook, select and highlight it, right-click, and select Run. To help test the code, we write messages to the Java console using the System.out.println method. In this example, we connect to the Java server running on the local system and listening on port 900. If the server is active and the security information is valid, we see the following message on the Java console.

Connected to localhost:900

If this is successful, we try to retrieve information from the Customer table on the server. To accomplish this, we use the getAllCustomers method from the ItemsDb class. Example 10-17 shows the code.
Example 10-17  Retrieving customers using the scrapbook

Support.ItemsDb aItemsDb = new Support.ItemsDb();
aItemsDb.setUserId("cpwejb");
aItemsDb.setPassword("cpwejb400");
aItemsDb.setPort("900");
aItemsDb.setSystemName("localhost");
aItemsDb.connect();
java.util.Vector customerList = aItemsDb.getAllCustomers();
for (java.util.Enumeration e = customerList.elements(); e.hasMoreElements();)
    String[] array = (String[]) e.nextElement();
    System.out.println(array[0] + " " + array[1]);

If this is successful, we retrieve the rows from the Customer table and display the first two fields for each row on the Java console. The getAllCustomers method returns a vector. The vector contains a String array element for each customer in the Customer table. The String array contains entries that correspond to the fields in the rows of the Customer table. For each String array element, we print the first two array elements. Example 10-18 shows the Java console output.

Example 10-18  Displaying the customer file rows on the Java console

Connected to localhost:900
0001 OAKLEY
0002 BARBER
0003 ABLE
0004 WILLIS
0005 MULLEN-SCHULTZ
0006 MAATTA
0007 FAIR
0008 COULTER
0009 GOUDET
0010 LLAMES

In this section, we tested the ItemsDb class without using a graphical user interface. In the next section, we build a servlet based OrderEntry application. We use the ItemsDb class to access the iSeries server.

10.3 Building servlets

Servlets are inherently multithreaded. Recall that stateful session beans are used to maintain a state for one client. Even if multiple Web users may be accessing the servlet, multithreading makes it appear as if the servlet is one client. A state is not automatically maintained across method calls for each Web client. This means that if you’re going to use a stateful session bean, you have to explicitly manage that connection.

The easiest and usually best way to connect servlets to session beans is to manage state information within the servlet, using stateless beans to perform business tasks. The stateless methods in the ItemsDb class do this.

As shown in Figure 10-24, we use two servlets. They are found in the tservlets package. They are ItemSessionServlet and CartServlet.
ItemSessionServlet
The ItemSessionServlet servlet is fairly simple. It displays a list of the items available for ordering. It also allows the user to select items to add to their shopping cart. The connection occurs in exactly the same way as we saw previously in the application using the ItemsDb class.

CartServlet
The CartServlet servlet is a command-driven servlet that supports adding items to a shopping cart and placing an order. It uses the ItemsDb class to actually place the order.

Servlet application flow
The application flow happens like this:

1. When the ItemSessionServlet is started, it connects to the OrderEntryClerk session bean using the ItemsDb class. It maintains this object, so all threads use the same object.
2. When a user executes the ItemSessionServlet, it causes the doPost() method to run. The doPost() method executes the getAllItems() method of the ItemsDb class. It returns a vector of items that is used to populate the items table displayed on the form. This is shown in Figure 10-25.
3. The user selects the items they want to order and clicks the Add to Cart button on the form (Figure 10-26).

4. This posts the form to the CartServlet using the Add to Cart command. The form shown in Figure 10-27 appears. The CartServlet creates a shopping cart object in the servlet session object.
5. When the user is done shopping, they can click the **Check Out** button on the form to place an order. This action passes the Check Out command to the CartServlet servlet.

6. The CartServlet servlet posts a verification form, shown in Figure 10-28, and requests a customer number.

7. The user enters a customer number and clicks the **Place Order** button. This causes the `placeOrder` method, shown in Example 10-19, to be called. It calls the `connectStateless` and `submitOrderStateless` methods in the `ItemsDb` class, which in turn, connect to the `OrderPlacement` session bean (which is stateless) and submit the order.
Example 10-19  The placeOrder method in CartServlet

```java
private void placeOrder(PrintWriter out, ShoppingCart cart, String custID) throws IOException {
    flexLog("CartServlet: placeOrder()...");
    Vector OrderLines = new Vector();

    Vector cartItems = cart.getItems();
    if (cartItems.size() > 0) {
        for (int i = 0; i < cartItems.size(); i++) {
            CartItem citem = (CartItem) cartItems.elementAt(i);
            com.ibm.itso.roch.cpwejb.interfaces.OrderDetail thisOrderDetail
                = new com.ibm.itso.roch.cpwejb.interfaces.OrderDetail(citem.getItemId(),
                Float.valueOf(citem.getPrice().replace('$','0')).floatValue(),
                1);
            OrderLines.addElement(thisOrderDetail);
        } // end for
    }
    try{
        ItemsDB MyItemsDB = new ItemsDb();
        MyItemsDB.connectStateless();
        out.println("<H3>Your order has been processed.</H3> <BR>");
        out.println("Order number : ");
        out.println(MyItemsDB.submitOrderStateless(custID, OrderLines));
        out.println("<BR><BR>Thanks for your business");
        out.flush();
    } catch (Exception e) {
        e.printStackTrace();
        out.println(e.getMessage());
    }
```

10.4 Migration from EJB version 1.0 to version 1.1

This section describes features that are new or have changed in the EJB 1.1 specification. It also discusses migration issues for enterprise beans written to version 1.0 of the EJB specification. We mainly concentrate on the changes we made to the code while migrating the OrderEntry application from version 1.0 to version 1.1.

From the client's perspective, enterprise beans written to version 1.1 of the EJB specification appear nearly identical to enterprise beans written to version 1.0. From the application developer's perspective, the following changes need to be made to make enterprise beans written to version 1.0 of the EJB specification compatible with version 1.1 of the specification:

- Enterprise beans written to version 1.1 of the EJB specification are registered in a different part of the JNDI namespace. For example, a client can look up the initial context of a version 1.0 enterprise bean in JNDI by using the `initialContext.lookup()` method as follows:

  ```
  initialContext.lookup("com/ibm/Hello")
  ```

  The JNDI lookup for the equivalent version 1.1 enterprise bean is:

  ```
  initialContext.lookup("java:comp/env/ejb/Hello")
  ```

- The return value of the `ejbCreate()` method must be modified for all entity beans using CMP. The `ejbCreate()` method is now required to return the same type as the primary key; the actual value returned must be null.
You must define and implement an ejbCreate() method for each way in which you want a new instance of an enterprise bean to be created. For each ejbCreate() method that you provide for an entity bean, you must also define a corresponding ejbPostCreate() method. Each ejbCreate() and ejbPostCreate() method pair must correspond to a specific create() method in the home interface that has the same number and types of arguments.

The EJBContext getEnvironment() method is deprecated. Use the JNDI naming context java:comp/env to access the enterprise beans environment. For example, the code in version 1.0 looks like this:

```java
ds = (DataSource) initCtx.lookup("jdbc/" +
getEntityContext().getEnvironment().getProperty("dataSourceName");
```

The EJB version 1.1 of the code looks like this:

```java
String dataSourceName = (String) initCtx.lookup("java:comp/env/dataSourceName");
ds = (DataSource) initCtx.lookup("jdbc/" + dataSourceName);
```

Throwing the java.rmi.RemoteException exception from the bean implementation is deprecated in version 1.1. This exception should be replaced by the javax.ejb.EJBException or a more specific exception such as the javax.ejb.CreateException. The javax.ejb.EJBException class inherits from javax.ejb.RuntimeException and does not need to be explicitly declared in the throws clause. Declare the javax.ejb.EJBException exception in the remote and home interfaces, as required by RMI.

10.5 Installing the OrderEntry application on the server

In this section, we install the OrderEntryApp enterprise application on WebSphere Application Server Advanced Edition using the Administrative Console. We export the .ear file from Application Developer and then deploy it on the WebSphere Application Server Version 4.0 Advanced Edition.

10.5.1 Generating the OrderEntry enterprise application

To generate the OrderEntry enterprise application file, follow these steps:

1. Select File-> Export from the Application Developer main screen.
2. The Export window appears as shown in Figure 10-29. Click Next.
3. In the next window, select the **OrderEntryApp** resource from the drop-down list for the **What resource do you want to export?** field.

4. Enter a local folder for the **Where do you want to export the resources to?** field. Click **Finish** as shown in Figure 10-30.

---

**Figure 10-29**  Exporting an Enterprise Application project

**Figure 10-30**  EAR export window
Once the OrderEntry application is exported, you can see the file in your local folder.

### 10.5.2 Deploying

First we copy the OrderEntryApp.ear file from our local drive to the iSeries IFS. We use the wsws directory. The Administrative Console can now access it when creating enterprise applications. To install the EAR file, follow these steps:

1. Open a command prompt window to start the Administrative Console. Wait until you see the message Console Ready in the Console.

2. In the Console, select the wizard icon and click **Install Enterprise Application**. The Specifying the Application or Module window displays, as shown in Figure 10-31. Make sure that the **Install Application radio** button is selected. Click the **Browse** button next to Path to locate the OrderEntryApp.ear file in the wsws directory. Enter **OrderEntryApp** for the Application name.

![Figure 10-31 Installing the application](image)

3. Keep clicking **Next** until you see the Specifying the Default Datasource for EJB Modules window. Click the **Select Datasource** button. Select the **NativeDS** Datasource. Enter a user ID and password (twice) and click **OK**.

   **Note:** We use a data source named **NativeDS**. For information on how to create a data source for WebSphere Application Server 4.0, see the redbook *WebSphere 4.0 Installation and Configuration on the IBM iSeries Server*, SG24-6815.

4. Click **Next**. In the Specifying Data Sources for individual CMP beans window, select all three of the CMP beans.

5. Click the **Select Datasource** button. Then select the **NativeDS** Datasource.

6. Enter a user ID and password (twice) and click **OK**. Now we have a Data Source for each CMP bean as shown in Figure 10-32.
7. Keep clicking **Next** until you see the Completing the Application Installation window.

8. Click **Finish** to install the application. When the Regenerate the application dialog displays, click **No**.

9. Now we have successfully installed the OrderEntryApp enterprise application on WebSphere Application Server Version 4.0 Advanced Edition. We need to stop the server. Before we restart it, we set up the classpath to find the IBM Toolbox for Java classes used by this application. As shown as Figure 10-33, click the server in the console. Click the **JVM Settings** tab in the right pane. Add the following JAR files to the Classpath Settings and click **Apply**:
   - `:/QIBM/ProdData/Java400/jt400ntv.jar`
   - `:/QIBM/ProdData/http/public/jt400/lib/jt400.jar`
10. Restart the server to make the new application ready. For an external HTTP server, you need to regenerate the Web server plug-ins.

11. Finally test the application and see if everything works properly. To test the application, open the browser and enter the following URL:

   http://systemname:port/OrderEntry2/ItemSessionServlet

   Replace systemname with the name of your iSeries server and the port with the HTTP port number.

### 10.6 Conclusion

In this chapter, the focus of discussion was to create EJB applications using Application Developer. We discussed the steps to create a simple session EJB and then develop a servlet and application that uses the bean.

We also discussed the OrderEntry application in detail. It is an EJB-based version of the RPG OrderEntry application discussed in Chapter 7, “Overview of the OrderEntry application” on page 231.
Interfacing to legacy applications

When developing new e-business applications, you need to consider using already existing applications. In many cases, you have applications written in traditional iSeries languages that perform complex business logic. Rather than re-writing these applications to Java, you may want to interface with them from your new Java-based Web applications.

This chapter discusses using Enterprise JavaBeans to interface with legacy applications. In this case, we interface with the OrderEntry application discussed in Chapter 7, “Overview of the OrderEntry application” on page 231.

Sometimes you need some services from other applications that do not reside on your system. You can reach these applications using middleware software such as MQSeries. We show an example of using MQSeries to interface from Enterprise JavaBeans to other applications.
11.1 Interfacing to legacy applications

There are a number of available options to call an existing traditional application from Java. Some of the most commonly used options are:

- **Distributed Program Call support** is available with IBM Toolbox for Java. You can pass parameters to and receive parameters back from an existing iSeries program. You must handle all conversions between the Java and iSeries format.

- **Program Call Markup Language (PCML)** is a tag language that helps you call iSeries programs while writing less Java code. PCML is based on XML, a tag syntax that you use to describe the input and output parameters for iSeries programs. PCML enables you to define tags that fully describe iSeries programs that will be called by your Java application.

- **Data queues** are iSeries objects that can be used to store data. They are convenient for passing information between applications. If you use a data queue to interface between a Java program and an existing iSeries non-Java program, you must handle data conversions.

- **Java Native Interface (JNI)** for Java is part of the Java Development Kit (JDK). JNI allows Java programs that run in a Java Virtual Machine (JVM) to operate with applications that are written in other languages such as C, C++, and RPG. The IBM Toolbox classes are easier to program than JNI. The advantage of JNI is that both the calling program and the called program run in the same process (job). The other methods start a new process. This makes JNI calls faster at start-up time and less resource intensive.

11.2 Modifying the RPG application

This section offers details about the transition of the RPG code on the host. The changes are made to allow the application to run in one of two modes: as a native application with 5250 display interaction or in conjunction with the OrderEntryClerk Enterprise JavaBean.

11.2.1 Processing the submitted order

The iSeries server RPG program that handles a request to submit an order is ORDENTR. When the OrderEntry application is run from an iSeries server 5250 session (no Java client), ORDENTR is the entry point of the application. It displays the 5250 windows that correspond to the Order Entry window in the Java client version. As shown in Figure 11-1, we use the IBM Toolbox for Java Distributed Program Call interface to call the ORDENTR program from the OrderEntryClerk EJB.
The ORDENTR program must be changed so that it recognizes the fact that it is being invoked from Java. First, the number of parameters are ascertained through the program status data structure using the following statements:

D PgmStsDS     SDS
D NbrParms     *PARMS

If the number of parameters is greater than zero, it is assumed that the program has been invoked as a distributed program.

Since the Java client passes in two parameters, two data structures are declared that map to the parameters. The Enterprise JavaBean passes two strings. The first string is nine characters representing the customer ID (four characters) and the number of detail entries (five characters). A data structure named CustDS is declared for this first parameter using the following statements:

D CustDS        DS
D CustNbr       LIKE(CID)
D OrdLinCnt     5  0

The second parameter is a string that represents a contiguous grouping of detail entries. Each entry has a length of 40, and there are a maximum of 50 entries. A data structure named OrderMODS is declared for this parameter using the following statements:

D OrderMODS     DS OCCURS(50)
D PartNbrX      LIKE(IID)
D PartDescX     LIKE(INAME)
D PartPriceX    5  2
D PartQtyX      5  0

An entry parameter list is added to the initialization subroutine. This ensures that the data structures are loaded with the parameter values passed in using the following statements:

C  *ENTRY        PLIST
C  PARM          CustDS
C  PARM          OrderMODS
As in the other RPG programs, the USROPN keyword is added to the file specification since the file is not opened when invoked as a distributed program. The portion of the file specification with the USROPN keyword is added using the following statements:

```plaintext
...WORKSTN SFILE(ORDSFL:SFIRrn) USROPN
```

The mainline logic of the program is changed to check the number of parameters. If there are parameters, a new subroutine called CmtOrder2 is invoked, and all display file processing is bypassed using the following statements:

```plaintext
C                   IF        NbrParms > *ZERO
C                   EXSR      CmtOrder2
C                   EXSR      EndPgm
C                   ENDIF
```

The CmtOrder2 subroutine is similar to the original CmtOrder subroutine. However, it retrieves the order information from the CustDS and OrderMODS data structures rather than from the display file and sub-file records.

### 11.3 Enhancing the Java application

The Program Call feature of IBM Toolbox for Java allows a Java program to directly execute any non-interactive program object (*PGM) on the iSeries, passing parameters and returning results through parameters. The Java developer must use the data conversion classes from IBM Toolbox for Java to convert input parameters from the Java format to an iSeries data type and convert output parameters from the iSeries format to a Java format.

The advantage of using the Distributed Program Call classes is that native non-interactive programs can be executed in your Java application unchanged. Native program calls can also result in better performance of your Java application when compared with JDBC. In addition, this interface can call programs on the iSeries that do more than just database access. For example, a Java application can call a program that starts nightly job processing, saves libraries to tape, or sends or receives data via communication lines.

#### 11.3.1 Changing the CartServlet servlet

The CartServlet is the responsible for showing the customer’s cart with the selected items and allowing the customer to place an order. In the original application, the CartServlet uses EJBs to do the placement of the order. We change the CartServlet servlet to interface with the RPG program.

**Adding the Place Order RPG option**

We change the CartServlet to provide a second option for placing an order. As shown in Figure 11-2, we add a new button labeled Place Order RPG. Clicking this button calls the RPG program to place the order.
As shown in Example 11-1, the outputOrderForm method in the CartServlet program is updated to display the new button.

Example 11-1  The updated outputOrderForm method

```java
private void outputOrderForm(PrintWriter out, ShoppingCart cart) throws IOException {
    flexLog("CartServlet: outputOrderForm()...");
    try{
        out.println("<CENTER>");
        out.println("<H3> Please verify the following order and enter your customer ID</H3>");
        out.println("<FORM METHOD=POST ACTION="/OrderEntry2/ItemSessionServlet" >");
        outputCartTable(out, cart);
        out.println("<CENTER>");
        out.println("<BR><INPUT TYPE=submit value="Continue Shopping" name="partno">");
        out.println("</FORM>");
        out.println("<FORM METHOD=POST ACTION="/OrderEntry2/CartServlet" >");
        out.println("Enter your Customer ID:<INPUT TYPE=TEXT name="custid"> <BR><BR>");
        out.println("<INPUT TYPE=submit value="Place Order" name="command">");
        out.println("<INPUT TYPE=submit value="Place Order RPG" name="command">"");
        out.println("</CENTER>");
        out.println("</BODY></HTML>");
        flexLog("CartServlet: outputOrderForm() executed.");
    } catch (Exception e) {
        e.printStackTrace();
        flexLog(e.getMessage());
    }
} // end outputItemInformation()
```
Managing Place Order input in CartServlet

Why can we use a single parameter to check whether you click the Place Order or Place Order RPG button? It is because we give both buttons the name command. We check the value of the command button to determine which button was clicked. When the customer clicks the Place Order RPG button, the CartServlet receives the input through the doPost method. The value of the parameter determines what to do. As shown in Example 11-2, if the value is Place Order RPG, the doPost method calls the placeOrderRPG method.

Example 11-2   Changing the doPost method

```java
else if (parameter.equalsIgnoreCase("Place Order RPG")){
    String custID = request.getParameter("custid");
    if (null == custID) custID="";
    // Make sure next order is clean
    session.invalidate();
    placeOrderRPG(out, cart, custID);
}
```

Creating the placeOrderRPG method

The placeOrderRPG method is shown in Example 11-3.

Example 11-3   The CartServlet placeOrderRPG method

```java
private void placeOrderRPG(PrintWriter out, ShoppingCart cart, String custID) throws IOException {
    flexLog("CartServlet: placeOrder()...");
    Vector OrderLines = new Vector();

    Vector cartItems = cart.getItems();
    if (cartItems.size() > 0) {
        for (int i = 0; i < cartItems.size(); i++) {
            CartItem citem = (CartItem) cartItems.elementAt(i);
            com.ibm.itso.roch.cpwejb.interfaces.OrderDetail thisOrderDetail =
                new com.ibm.itso.roch.cpwejb.interfaces.OrderDetail(citem.getItemId(),
                Float.valueOf(citem.getPrice().replace('$$','0')).floatValue(), 1);
            OrderLines.addElement(thisOrderDetail);
        } // end for
    } // end if

    try{
        MyItemsDB.connectStateless();
        out.println("<INPUT TYPE=submit value="Shop Some More..." name="command">");
        out.flush();
    } catch (Exception e) {
```
This method retrieves the items from the shopping cart and adds them to a vector. It uses the ItemsDb class to interface with the OrderPlacement EJB. An instance of the ItemsDb class named MyItemsDB is instantiated. The MyItemsDB object submitOrderStatelessRPG method is called to submit the order to the server. The customer number and the vector containing the items to order are passed to this method. The submitOrderStatelessRPG method returns the order number, and it is displayed on the browser.

11.3.2 Changing the ItemsDb class

We add a new method to the ItemsDb class named submitOrderStatelessRPG.

Example 11-4  The ItemDb submitOrderStatelessRPG method

```java
public int submitOrderStatelessRPG(String CustID, java.util.Vector OrderLines) throws Exception {
    int returnvalue;
    try {
        returnvalue = (int)OrderPlacer.placeOrderRPG("0001", 1, CustID, OrderLines);
    } catch (Exception e) {
        throw (e);
    }
    return returnvalue;
}
```

OrderPlacer is an instance of the OrderPlacement session EJB. We call its placeOrderRPG method. Four parameters are passed in. The warehouse ID and district ID are set to "0001" and 1 respectively. The customer number and a vector containing the items to order are passed in. If everything works successfully, the placeOrderRPG method returns the order number of the order created.

11.3.3 Changing the OrderPlacement session bean

Now we need to enhance the OrderPlacement session bean to call the RPG program to place an order. We add a new method to the OrderPlacement bean named placeOrderRPG. This is the method that is called from the ItemsDb class as we saw in the previous topic. We also add a new internal method to OrderPlacement named submitOrder(String header, String detail). The submitOrder method contains the actual IBM Toolbox for Java calls. It is called from the placeOrderRPG method. It cannot be called from a client application because we do not add it to the remote interface.

First, we use Application Developer to externalize the name of the program to call. We also store the iSeries user ID and password to use when creating the AS400 object. To add these values to the bean properties, follow these steps:

1. In the WebSphere Application Developer, open the J2EE perspective, as shown in Figure 11-3.
2. In the J2EE view, expand **EJB Modules**.

3. Right-click the **ToWebSphere** EJB Module and select **Open With-> EJB Editor** as shown in Figure 11-4.

4. Go to the Environment pane.

5. Select the **OrderPlacement** EJB and click the **Add** button.

6. Select (New Variable) and change it to **dataSourceName**.

7. Select the value column for the dataSourceName property and enter **NativeDS**.

8. Add the variables as shown in Table 11-1.

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataQueueName</td>
<td>String</td>
<td>/QSYS.LIB/TEAM41.LIB/ORDERS.DTAQ</td>
</tr>
<tr>
<td>library</td>
<td>String</td>
<td>TEAM41</td>
</tr>
<tr>
<td>program</td>
<td>String</td>
<td>ORDENTR</td>
</tr>
<tr>
<td>user</td>
<td>String</td>
<td>A valid user ID</td>
</tr>
<tr>
<td>password</td>
<td>String</td>
<td>A valid password</td>
</tr>
</tbody>
</table>

Figure 11-5 shows the EJB editor with all the environment variables.
The placeOrderRPG method

Now we create the placeOrderRPG method as shown in Example 11-5.

Example 11-5  The placeOrderRPG method

```java
public float placeOrderRPG(String wID, int dID, String cID, java.util.Vector orderLines) throws EJBException {
    System.out.println("into placeOrderRPG");
    String orderNumber = null;
    String custID = cID;
    Vector items = orderLines;
    try {
        // Maximum of 50 order lines
        String numLines = null;
        if (items.size() >= 10)
            numLines = "000" + Integer.toString(items.size());
        else
            numLines = "0000" + Integer.toString(items.size());
        String header = custID + numLines;
        String details = null;
        InitialContext initCtx = new InitialContext();
        ItemHome iHome = (ItemHome) initCtx.lookup("Item");
        for (int i = 0; i < items.size(); i++) {
            OrderDetail od = (OrderDetail) items.elementAt(i);
            Item item = iHome.findByPrimaryKey(new ItemKey(od.getItemID()));
            String qty;
            if (od.getItemQty() < 10)
                qty = "0000" + Integer.toString(od.getItemQty());
            else
                qty = "000" + Integer.toString(od.getItemQty());
            StringBuffer priceBuffer = new StringBuffer(5);
            String priceString = Float.toString(item.getItemPrice());
            int decimalPosition = priceString.indexOf(‘.’);
            String priceString1 = priceString.substring(0, decimalPosition);
            String priceString2 = priceString.substring(decimalPosition + 1);
            if (priceString2.length() == 1) {
                priceString2 += "0";
            }
            priceString = priceString1 + priceString2;
            for (int j = 0; j < 5 - priceString.length(); j++) {
```

**Figure 11-5  EJB Editor - Environment pane**
priceBuffer.append('0');

priceBuffer.append(priceString);

// set the name making sure trailing blanks are there
StringBuffer nameBuffer = new StringBuffer(24);
nameBuffer.append(item.getItemName());
for (int j = nameBuffer.length() + 1; j < 25; j++) {
    nameBuffer.append(' ');
}

if (details == null)
    details = od.getItemID() + nameBuffer + priceBuffer + qty;
else
    details += od.getItemID() + nameBuffer + priceBuffer + qty;

// call the submitOrder method to place the order
orderNumber = submitOrder(header, details);
// Clear out the state of the session bean at this point
items = new Vector();
custID = null;
} catch (Exception e) {
    throw new EJBException(e.getMessage());
}
return Float.parseFloat(orderNumber);

The placeOrderRPG method is responsible for calling the submitOrder method, which contains the actual IBM Toolbox for Java code that calls the RPG program.

Before we call the submitOrder method, we set two parameters:

- The first parameter is a string that is a concatenation of the customer ID and the number of entries in the order. This data is fixed in length. The first four bytes are designated for the customer ID, while the last five bytes are for the number of entries. This character data will be interpreted as two separate zoned numeric fields on the iSeries server. These nine bytes of data are the first nine bytes in the string returned by the Order.toString() method.

- The second parameter is a block of character data that represents an array corresponding to the order information. Each element in the array corresponds to an item to be ordered. Each row is a fixed length String with the different elements of an entry beginning at the following offsets:
  - item id: offset 0
  - item desc: offset 6
  - item qty: offset 30
  - item price: offset 35

The submitOrder method

Calling a native iSeries program using The IBM Toolbox for Java Distributed Program Call class involves the following steps:

1. Connect to the iSeries server by creating an AS400 object.
2. Create a ProgramCall object.
3. Define and initialize a ProgramParameter array for passing parameters to and from the called program.
4. Use the Data Conversion classes to convert input parameter values from the Java to iSeries format.
5. Use the `setProgram` method to specify the qualified name of the program to call and parameters to use.
6. Execute the program using the `run` method.
7. If the `run` method fails, obtain detailed error information via AS400Message objects.
8. Retrieve output parameters using the `getOutputData` method of the `ProgramParameter` object.
9. Convert the output parameter values using the data conversion classes.

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

The AS400DataType is an interface that defines the methods required for data conversion. A Java program uses data types when individual pieces of data need to be converted.

Conversion classes exist for the following types of data:

- Numeric
- Text (character)
- Composite (numeric and text)

The data description classes build on the data conversion classes to convert all fields in a record with a single method call. The RecordFormat class allows the program to describe data that makes up a DataQueueEntry, ProgramCall parameter, a record in a database file accessed through record-level access classes, or any buffer of iSeries data. The Record class allows the program to convert the contents of the record and access the data by field name or index.

The IBM Toolbox for Java provides classes for building on the data types classes to handle converting data one record at a time instead of one field at a time. For example, suppose a Java program reads data off a data queue. The data queue object returns a byte array of iSeries data to the Java program. This array can potentially contain many types of iSeries data. The application can convert one field at a time out of the byte array by using the data types classes. Or, the program can create a record format that describes the fields in the byte array. That record then does the conversion.

Record format conversion can be useful when you are working with data from the program call, data queue, and record-level access classes. The input and output from these classes are byte arrays that can contain many fields of various types. Record format converters can make it easier to convert this data between the iSeries format and Java format.

Conversion with the record format uses three classes:

- **Field description** classes identify a field or parameter with a data type and a name.
- A **record format** class describes a group of fields.
- A **record class** joins the description of a record (in the record format class) with the actual data.

The `submitOrder` method, shown in Example 11-6, uses the IBM Toolbox for Java Distributed Program Call class (DPC) to call the RPG program named `ORDENTR`. It passes two parameters to the ORDENTR program. The first parameter contains information about the customer. The second parameter contains information about the items to order. The RPG program contains all the logic for creating an order and updating the appropriate tables.
Example 11-6  The submitOrder method

```java
private String submitOrder(String header, String detail) throws java.rmi.RemoteException {
    String newOrder = null;
    try {
        AS400 as400 = new AS400("localhost",
            mySessionCtx.getEnvironment().getProperty("user"),
            mySessionCtx.getEnvironment().getProperty("password"));
        ProgramCall ordEntrPgm = new ProgramCall(as400);
        QSYSObjectPathName pgmName = new QSYSObjectPathName(mySessionCtx.getEnvironment().getProperty("library"),
            mySessionCtx.getEnvironment().getProperty("program"), "PGM");
        ProgramParameter[] parmList = new ProgramParameter[2];
        // set the first parameter which is the order header
        AS400Text txt1 = new AS400Text(9);
        byte[] headerInfo = txt1.toBytes(header);
        parmList[0] = new ProgramParameter(headerInfo, 9);
        AS400Text txt2 = new AS400Text(detail.length());
        byte[] detailInfo = txt2.toBytes(detail);
        parmList[1] = new ProgramParameter(detailInfo);
        ordEntrPgm.setProgram(pgmName.getPath(), parmList);
        if (ordEntrPgm.run() != true) {
            // If you get here, the program failed to run.
            // Get the list of messages to determine why
            // the program didn't run.
            AS400Message[] messageList = ordEntrPgm.getMessageList();
            throw new CpwejbException("OrderEntryClerkBean: " +
                messageList[0].getText());
        }
        newOrder = ((String) (new AS400Text(9).toObject(parmList[0].getOutputData(),
            0))).substring(0, 4);
    } catch (Exception e) {
        throw new RemoteException("Error submitting order: " + e.getMessage());
    }
    return newOrder;
}
```

We first create an AS400 object. The system name is localhost. Using the properties that we
retrieve from the deployment descriptor, we set the user ID and password.

Next, we set the name of the program to call. Again we use properties to set these values. Before we call the program, we set two parameters:

- The first parameter is a string that is a concatenation of the customer ID and the number of entries in the order.
- The second parameter is a block of character data that represents an array corresponding to the order information.

After we run the program, we check to see whether the program is a success or failure. If the program fails, we retrieve the error messages. If it is successful, the new order number is returned in the first parameter. We return it to our caller.

In this example, an RPG program is called from an Enterprise JavaBean. The server must find the RPG program. This is accomplished by changing the current library for the user profile that calls the program to the library containing the program. In this case, the RPG program in stored in a library named TEAM41.
11.4 Using data queues to interface to legacy applications

A data queue is a system object that exists on the iSeries server. Data queues have the following characteristics:

- They can be accessed by many jobs simultaneously.
- They can be used to hold data being passed back and forth between jobs.
- Messages on a data queue are free format; fields are not required like they are in a database.
- The data queue is a fast means of communications between jobs on the iSeries server. This makes it an excellent way to communicate or synchronize iSeries jobs.

The messages on a data queue can be ordered in one of three ways:

- **LIFO**: The last message (newest) placed on the data queue is the first message taken off of the queue.
- **FIFO**: The first message (oldest) placed on the data queue is the first message taken off of the queue.
- **KEYED**: Each message on the data queue has a key associated with it. A message can only be taken off of the queue by requesting the key associated with it.

The DataQueue classes allow Java programs to interact with iSeries data queues. They provide a complete set of interfaces for accessing the iSeries data queues from your Java program. It is an excellent way to communicate between Java programs and iSeries programs. You can write the iSeries program in any language.

A required parameter of each data queue object is the AS400 object that represents the iSeries server that has the data queue or where the data queue is to be created. Using the data queue classes causes the AS400 object to connect to the server. Each data queue object requires the integrated file system path name of the data queue. The type for the data queue is DTAQ.

11.4.1 Interfacing to data queues from EJBs

As shown in Figure 11-6, we use data queue support to interface between the OrderPlacement enterprise bean and an RPG program named PRTORDERP. The RPG program prints order requests.

The OrderPlacement bean places order information on an iSeries data queue each time it places an order. PRTORDERP is a never-ending RPG program that runs in the batch subsystem. It waits for entries to appear on the data queue. When an entry appears, it uses the information to create a print request. The output produced by PRTORDERP is shown in Figure 11-7.
11.4.2 The writeDataQueue method

We add the writeDataQueue method to the OrderPlacement EJB. It is called from the placeOrder method. It places the order information on the data queue so it can be read and printed by the RPG program. We do not add the writeDataQueue method to the remote interface because we do not want to make it available to client programs. It can only be called by methods of the OrderPlacement EJB. See Example 11-7.
Example 11-7  The writeDataQueue method

```java
private void writeDataQueue(String wID, int dID, String cID, float oID) throws Exception {
    AS400 as400 = new AS400("localhost", "*current", "*current");
    CharacterFieldDescription as4CustomerID = new CharacterFieldDescription(new AS400Text(4), "customerID");
    PackedDecimalFieldDescription as4DistrictID = new PackedDecimalFieldDescription(new AS400PackedDecimal(3,0), "districtID");
    CharacterFieldDescription as4WarehouseID = new CharacterFieldDescription(new AS400Text(4), "warehouseID");
    PackedDecimalFieldDescription as4OrderID = new PackedDecimalFieldDescription(new AS400PackedDecimal(9,0), "orderID");
    String dataQueueName = props.getProperty("DataQueueName");
    DataQueue dqOutput = new DataQueue(as400, dataQueueName);

    RecordFormat rfOutput = new RecordFormat();
    rfOutput.addFieldDescription(as4CustomerID);
    rfOutput.addFieldDescription(as4DistrictID);
    rfOutput.addFieldDescription(as4WarehouseID);
    rfOutput.addFieldDescription(as4OrderID);
    // Create wrappers for dID and oID because setField only works on objects.
    BigDecimal dIDObject = new BigDecimal(dID);
    BigDecimal oIDObject = new BigDecimal(oID);
    // Set up the data queue entry field values
    Record recordOutput = rfOutput.getNewRecord();
    recordOutput.setField("customerID", cID);
    recordOutput.setField("warehouseID", wID);
    recordOutput.setField("orderID", oIDObject);
    recordOutput.setField("districtID", dIDObject);
    // Send the data queue entry
    dqOutput.write(recordOutput.getContents());
    return;
}
```

The AS400 object allows special values for system name, user ID, and password when the Java program is running on the iSeries Java Virtual Machine. They are `localhost` for system name and `*current` for user ID and password.

When you run a program on the iSeries JVM, be aware of some special values and other considerations:

- If the system name, user ID, or password is not set on the AS400 object, the AS400 object connects to the current system by using the user ID and password of the job that started the Java program.
- The special value, `localhost`, can be used as the system name. In this case, the AS400 object connects to the current system.
- The special value, `*current`, can be used as the user ID or password on the AS400 object when the Java program is running on the JVM of one iSeries server and is accessing resources on another iSeries server. In this case, the user ID and password of the job that started the Java program on the source system are used when connecting to the target system.
- The Java program cannot set the password to "*current" if you are using record-level access. When you use record-level access, "localhost" is valid for system name and "*current" is valid for user ID. However, the Java program must supply the password.
- The value "*current" works only on systems running at Version 4 Release 3 (V4R3) and later. The password and user ID must be specified on systems running at V4R2.
In this example, we use record format conversion. It can be useful when you are working with data from data queues. The input and output from these classes are byte arrays that can contain many fields of various types. Record format converters can make it easier to convert this data between the iSeries format and the Java format.

We show the call to write the entry to the data queue. Since we have set up a record format to handle the data conversions. We only have to set the fields of the record format before writing the entry to the data queue.

11.5 Using MQSeries to interface to legacy applications

MQSeries is a middleware product that runs on many platforms. It uses queues to allow applications to interface with other applications. It is similar to iSeries data queue support. However, data queue support only runs on the iSeries server. With MQSeries, you can interface to applications running on many different environments, including the iSeries server. The applications can be on the same system or different systems.

This section shows you how to send data to an MQSeries queue using Java application programming interfaces (APIs). These APIs allow you to integrate your application with any other application that can read messages from an MQSeries queue. We change the application discussed in 11.4, “Using data queues to interface to legacy applications” on page 363, to use an MQSeries queue instead of a data queue. This allows us to send the order information to any platform capable of running MQSeries.

The MQSeries API classes are included in the com.ibm.mq.jar file. To use MQSeries in Application Developer, you must add this file to the Java Build Path of your project properties. To run an application using MQSeries on WebSphere Application Server, you need to add this file to the classpath, as shown in Figure 11-9.

Figure 11-9 Including the com.ibm.mq.jar file in the JVM settings
When you use the MQSeries API, you must understand these concepts:

- **Hostname** is the machine that allocates the MQSeries server. This machine can be the same one that is running your application or a different one.
- **Queue Manager** provides message queueing services for applications.
- **Channel** connects an MQSeries client to a queue manager on a server system and transfers MQ calls and responses.
- **Queue** is an MQSeries object. Clients can put messages on, and get messages from, a queue.

### 11.5.1 The MQCon class

To encapsulate all the MQSeries APIs, we create a class that provides all the functionality that applications require. To make application methods easier to write, all the MQSeries functions are found in this class.

### Declaration of the MQCon class

First we import the classes that are included into the com.ibm.mq package. We create the variables that define the MQSeries objects. These objects are hostname, queue manager, channel, and the queue. We also declare a queue manager object. The MQCon class is defined as shown in Example 11-8.

**Example 11-8  The MQCon class definition**

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

public class MQCon
{
    private String hostname;
    private String qManager;
    private String channel;
    private String queue;

    private MQQueueManager qMgr;
}
```

### The MQCon constructor

The constructor for the MQCon class (Example 11-9) accepts the values of the declared variables. When you create an MQCon object, you pass in hostname, queue manager, channel, and queue. In the constructor these parameters are set to the class variables. We also need to set the hostname and the channel for the MQSeries environment. We set two environment transport properties.

**Example 11-9  The MQCon constructor**

```java
public MQCon(String hostname_, String qManager_, String channel_, String queue_)
{
    super();
}
```
Connecting to the queue manager

We create a method to make the connection to the queue manager (Example 11-10). This method can be used when you need to interact with MQSeries. The method returns true if the connection is successful. You must handle an MQException that can be thrown when trying to connect to the queue manager.

Example 11-10  The connectTOQM method

```java
public boolean connectTOQM()
{
    boolean res = true;
    try
    {
        qMgr = new MQQueueManager(qManager);
        System.out.println("connected");
    }
    catch (MQException ex)
    {
        res = false;
        String resS = "ERROR MQSeries : Completion Code " + ex.completionCode + 
                       " Reason Code " + ex.reasonCode;
        System.out.println(resS);
    }
    return res;
}
```

Putting messages in a queue

To put messages in the queue, we create a method named putMQ (Example 11-11). This method receives the message through a parameter. If the message is sent to the queue correctly, the method returns the message ID. If an error occurs, the method returns "ERROR" and an explanation of the error.

The method connects to the queue manager using the connectTOQM method. Then it opens the queue using simple options. A MQMessage instance is created, and we can write messages to it. Finally, we close the queue, disconnect from the queue manager, and return the message ID value.

Example 11-11  The putMQ method

```java
public byte[] putMQ(String msg)
{
    byte[] res;
    try
```
```java
{  
    if (this.connectTOQM())  
    {  
        int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |  
                          MQC.MQOO_OUTPUT;  
        MQQueue mqQueue = qMgr.accessQueue(queue,  
                                            openOptions);  
        MQMessage mqMsg = new MQMessage();  
        mqMsg.writeUTF(msg);  
        MQPutMessageOptions pmo = new MQPutMessageOptions();  
        mqQueue.put(mqMsg, pmo);  
        System.out.println("MQCon.putMQ():Message:<>" +  
                            new String(mqMsg.messageId) + ">> +  
                            msg + ");  
        res = mqMsg.messageId;  
        mqQueue.close();  
        qMgr.disconnect();  
    }  
    else  
    {  
        res = new String("ERROR MQSeries connection:" + msg).getBytes();  
    }  
}  
catch (MQException ex)  
{  
    res = new String("ERROR MQSeries : Completion Code " + ex.completionCode +  
                     " Reason Code " + ex.reasonCode).getBytes();  
    System.out.println(res);  
}  
catch (java.io.IOException ex)  
{  
    res = new String("ERROR MQSeries IO:" + ex).getBytes();  
    System.out.println(res);  
}  
return res;
```

**Reading messages**

We also create a method named `getMQ` (Example 11-12). This method is not necessary for our application, but we include it to show how to get a message from a queue. This method receives the message Id of the message to read.

The method connects to the queue manager using the `connectTOQM` method. A `MQMessage` instance is created. We read the message using the `readUTF` method. Finally, we close the queue, disconnect from the queue manager, and return the message.

**Example 11-12  The `getMQ` method**

```java
public String getMQ(byte[] msgId)  
{  
    String res = "ERROR";  
    try
```
11.5.2 Including MQSeries in the OrderEntry application

In this section, we show how to include MQSeries calls in the OrderEntry application.

Changing the CartServlet servlet
The CartServlet is the responsible of showing the customer’s cart with the selected items and allowing the customer to place an order. We change the CartServlet servlet to use the MQSeries interface.
Showing the Place Order MQ option

We add a new button to the CartServlet output form. It allows customers to place and order and have the order information placed on a MQSeries queue. This interface is similar to the data queue example discussed in 11.4, “Using data queues to interface to legacy applications” on page 363. The difference is that data queues only work on the iSeries server, while MQSeries works on many platforms. The new form is shown in Figure 11-10.

![Figure 11-10 The place order page](image)

Example 11-13 shows the updated outputOrderForm method. We add a new button with a type of submit and a value of Place Order MQ.

Example 11-13 The outputOrderForm

```java
private void outputOrderForm(PrintWriter out, ShoppingCart cart) throws IOException {
    flexLog("CartServlet: outputOrderForm()...");
    try{
        out.println("<CENTER>");
        out.println("<H3> Please verify the following order and enter your customer ID</H3>");
        out.println("<FORM METHOD=POST ACTION="/OrderEntry2/ItemSessionServlet" >");
        outputCartTable(out, cart);
        out.println("<BR><INPUT TYPE=submit value="Continue Shopping" name="partno">");
        out.println("</CENTER>");
        out.println("</FORM>");
        out.println("<FORM METHOD=POST ACTION="/OrderEntry2/CartServlet" >");
        out.println("Enter your Customer ID:<INPUT TYPE=TEXT name="custid"> <BR><BR>");
        out.println("<INPUT TYPE=submit value="Place Order" name="command">");
        out.println("<INPUT TYPE=submit value="Place Order RPG" name="command">");
        out.println("<INPUT TYPE=submit value="Place Order MQ" name="command">");
        out.println("</CENTER>");
        out.println("</FORM>");
        out.println("</BODY></HTML>");
        flexLog("CartServlet: outputOrderForm() executed.");
    } catch (Exception e) {
```
Managing the Place Order input in CartServlet

We use a single parameter to check whether you click the Place Order, the Place Order RPG, or the Place Order MQ button. This is because we give all the buttons the name command. We check the value of the command button to determine which button was clicked. When the customer clicks the Place Order MQ button, the CartServlet receives the input through the doPost method. The value of the parameter determines what to do. As shown in Example 11-14, if the value is "Place Order MQ", the doPost method calls the placeOrderMQ method.

Example 11-14 Changing the doPost method

```java
else if (parameter.equalsIgnoreCase("Place Order MQ")){
    String custID = request.getParameter("custid");
    if (null == custID) custID="";
    // Make sure next order is clean
    session.invalidate();
    placeOrderMQ(out, cart, custID);
}
```

The placeOrderMQ method

To process the order with the RPG program, we create a new method. This method is called placeOrderMQ and is shown in Example 11-15.

Example 11-15 The placeOrderMQ method in CartServlet

```java
private void placeOrderMQ(PrintWriter out, ShoppingCart cart, String custID) throws IOException {
    flexLog("CartServlet: placeOrderMQ()...");
    Vector OrderLines = new Vector();
    Vector cartItems = cart.getItems();
    if (cartItems.size() > 0) {
        for (int i = 0; i < cartItems.size(); i++) {
            CartItem citem = (CartItem) cartItems.elementAt(i);
            com.ibm.itso.roch.cpwejb.interfaces.OrderDetail thisOrderDetail
                = new com.ibm.itso.roch.cpwejb.interfaces.OrderDetail(citem.getItemId(),
                Float.valueOf(citem.getPrice().replace('$','0')).floatValue(),
                1);
            OrderLines.addElement(thisOrderDetail);
        } // end for
        try{
            MyItemsDB.connectStateless();
            out.println("<FORM METHOD="get"
                ACTION="/OrderEntry2/ItemSessionServlet">");
            out.println("<H3>Your order has been processed and sent to an MQ queue.</H3>
                <BR>");
```
out.println("Order number : ");
out.println(MyItemsDB.submitOrderStatelessMQ(custID, OrderLines));
out.println("<BR><BR>Thanks for your business<BR><BR>");
out.println("<BR><BR><INPUT TYPE=submit value="Shop Some More..." name="command">");
out.flush();
}
catch (Exception e) {
e.printStackTrace();
out.println(e.getMessage());
}
}

This method retrieves the items from the shopping cart and adds them to a vector. It uses the ItemsDb class to interface with the OrderPlacement EJB. An instance of the ItemsDb class named MyItemsDB is instantiated. The MyItemsDB object submitOrderStatelessMQ method is called to submit the order to the server. The customer number and the vector containing the items to order are passed to this method. The submitOrderStatelessMQ method returns the order number; it is displayed on the browser.

11.5.3 Changing the ItemsDb class

As we saw in the previous section, the CartServlet uses an instance of the ItemDb class to place an order. We add the submitOrderStatelessMQ method (Example 11-16) to this class.

Example 11-16 The submitOrderStatelessMQ method

```java
public int submitOrderStatelessMQ(String CustID, java.util.Vector OrderLines) throws Exception {
    int returnvalue;
    try {
        returnvalue = (int)OrderPlacer.placeOrderMQ("0001", 1, CustID, OrderLines);
    } catch (Exception e) {
        throw (e);
    }
    return returnvalue;
}
```

OrderPlacer is an instance of the OrderPlacement session EJB. We call its placeOrderMQ method. Four parameters are passed in. The warehouse ID and district ID are set to “0001” and 1 respectively. The customer number and a vector containing the items to order are passed in. If everything works successfully, the placeOrderMQ method returns the order number of the order created.
11.5.4 Changing the OrderPlacement session bean

We need to enhance the OrderPlacement session bean to send a message to an MQSeries queue after placing the order. We add a new method to the OrderPlacement bean named `placeOrderMQ`. This is the method that is called from the `ItemsDb.submitOrderStatelessMQ` method. We also add a new internal method to the OrderPlacement EJB named `writeMQ`. The `writeMQ` method creates an instance of an MQCon object in order to send the message to the queue. The `writeMQ` method is called from the `placeOrderMQ` method. It cannot be called from a client application because we do not add it to the remote interface.

We use Application Developer to externalize the name of the MQSeries resources such as hostname, queue manager, channel, and queue. To add these values to the bean properties, follow these steps:

1. In the WebSphere Application Developer, open the J2EE perspective, as shown in Figure 11-11.

![Figure 11-11 Opening the J2EE perspective](image)

2. In the J2EE view, expand EJB Modules.

3. Right-click the ToWebSphere EJB Module and select Open With-> EJB Editor, as shown in Figure 11-12.

![Figure 11-12 Opening the EJB Editor](image)

4. Go to the Environment pane.

5. Select the OrderPlacement EJB and click the Add button.

6. Select (New Variable) and change it to hostname.
7. Select the value column for the hostname property and enter the name of the server that contains the MQSeries server.

We add the environment variables shown in Table 11-2.

**Table 11-2  Properties for OrderPlacement EJB**

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostname</td>
<td>String</td>
<td>The name of the machine that contains the MQServer</td>
</tr>
<tr>
<td>qmanager</td>
<td>String</td>
<td>The name of the Queue Manager</td>
</tr>
<tr>
<td>channel</td>
<td>String</td>
<td>The name of the Channel</td>
</tr>
<tr>
<td>queue</td>
<td>String</td>
<td>The name of the queue</td>
</tr>
</tbody>
</table>

Figure 11-13 shows the EJB Editor with our values.

Now we create the `placeOrderMQ` method as shown in Example 11-17. This method is similar to the original `placeOrder` method. The only difference is we call the MQSeries interface instead of the data queue interface.

**Example 11-17  The placeOrderMQ method.**

```java
public float placeOrderMQ(String wID, int dID, String cID, Vector orderLines) throws EJBException {
    float oID = 0;
    try {
        //The InitialContext will let us retrieve references to the entity beans we need.
        InitialContext initCtx = new InitialContext();
        //Get the Order Number from the District entity bean.
        DistrictHome dHome = (DistrictHome) initCtx.lookup("District");
        DistrictKey districtID = new DistrictKey(dID, wID);
        District district = (District) dHome.findPrimaryKey(districtID);
        int oIDint;
        oIDint = district.getNextOrderId(true); //'true' tells the District to increment the
        oID = oIDint;
    }
    //...
The `writeMQ` method, shown in Example 11-18, uses the MQCon class to send the message to MQSeries. First, the MQSeries resource names are obtained from the OrderPlacement properties using the InitialContex object. We create an String object that contains the order data. A new MQCon instance is created. We put the order on the queue using the `putMQ` method.

```
Example 11-18   The writeMQ method
private void writeMQ(String wID, int dID, String cID, float oID) throws Exception
{
    InitialContext initCtx = new InitialContext();

    String hostname = (String)initCtx.lookup("java:comp/env/hostname");
    String channel = (String)initCtx.lookup("java:comp/env/channel");
    String qmanager = (String)initCtx.lookup("java:comp/env/qmanager");
    String queue = (String)initCtx.lookup("java:comp/env/queue");

    String order= wID + "=" + dID + "=" + cID + "=" + oID;
    MQCon mqcon = new MQCon(hostname, qmanager, channel, queue);
```

```
```java
byte[] res = mqcon.putMQ(order);
System.out.println(new String(res));
```

With the `writeMQ` method, we only put a message into a queue. MQSeries can send the message to a program on the same system or another system. This is controlled by the `hostname` variable.

**Testing the MQSeries application**

In the OrderEntry application, we place an order using the Place Order MQ button as shown in Figure 11-14.

![Figure 11-14 Using the Place Order MQ button](image)

The application returns an order confirmation message as shown in Figure 11-15. The order information has also been placed on the MQSeries queue. MQSeries can route the message to a program on the iSeries server or to another platform.

![Figure 11-15 Order sent to the MQSeries queue message](image)

For the MQSeries server, we use a Windows 2000 workstation. We can see the message in the queue browser as shown in Figure 11-16.
We select the message from the browser and click the **Properties** button. If we select the Data pane, we see the order data as shown in Figure 11-17.

For further information about using MQSeries from a Java application, refer to *MQSeries: Using Java*, SC34-5456.
11.6 Using XML to interact with applications

The previous sections explained how to send messages to another application using MQSeries or data queues. Sometimes these messages are difficult to read because they became an array of characters without any meaning. The receiver of these messages needs to know the exact structure of the data. With XML, we can send messages that include the structure of the data. This makes it easier for the receiver of the message to process it. Also, the message may be read by many receivers. Each one can read only the information it needs.

You can send an XML file as a message through a queue. You need to convert the file into a String object and then send it to the queue.

This section shows how to create XML messages and send them through MQSeries or data queues. We enhance the OrderEntry application and change the information we send to the MQSeries queue to use XML.

11.6.1 Using XML

For interacting with XML, we need to create XML files and write Java code that manages the information in these files. There are several ways to handle XML with Java code:

- **Document Object Model (DOM)**: DOM defines a set of interfaces to access tree-structured XML documents. The root of the inheritance tree is a Node, that defines the necessary methods to navigate and manipulate the tree-structure of XML documents.

- **Simple API for XML (SAX)**: SAX is an event-driven lightweight API for accessing XML documents and extracting information from them. It cannot be used to manipulate the internal structures of XML documents. As the document is parsed, the application using SAX receives information about the various parsing events.

We use DOM in our example.

The XML file

First, we create an XML file with the structure of the data we want to send. In this file, we include all the fields in the order record that we place on the MQSeries queue. Remember that the order information we send contains four fields: warehouse id, district id, customer number, and order number. The order.XML file is shown in Example 11-19.

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
  <warehouse>warehouse</warehouse>
  <district>district</district>
  <customer>customer</customer>
  <orderNum>orderNum</orderNum>
</order>
```

To create this file with Application Developer, follow these steps:

1. In the main menu, click **Perspective-> Open-> XML** as shown in Figure 11-18.
2. In the Navigator view, right-click the folder that will contain the XML file. Then, in the pop-up menu, select **New-> XML File** as shown in Figure 11-19.

3. Select **Create XML file from scratch** as shown in Figure 11-20 and click **Next**.
4. Enter *order.xml* as the name of the XML file, as shown in Figure 11-21, and click **Finish**.

![Figure 11-21 XML file name](image)

5. Once the XML file is created the XML editor opens. This editor has two views, Design and Source. In the Design view, the XML is represented graphically, and in the Source view, we see the XML code itself.

   Change to the Design view in the XML editor as shown in Figure 11-22.

![Figure 11-22 Design view in the XML editor](image)

6. Right-click the **XML element** and in the pop-up menu, select **Add after-> New element** as shown in Figure 11-23.
7. Enter order as the Element Name in the New Element window and click **OK** as shown in Figure 11-24.

8. Right-click the order element, and in the pop-up menu, select **Add Child** -> **New Element** as shown in Figure 11-26.
Chapter 11. Interfacing to legacy applications

9. Enter warehouse as the new element name and click OK.

Now we have a child element of the order element as shown in Figure 11-27.

10. In the right column, enter warehouse as the element value as shown in Figure 11-28.

11. As we did with warehouse, we add district, customer, and numorder as child elements as shown in Figure 11-29.
12. Save the file by pressing Ctrl-S.

Now we can switch to the Source view to see the actual XML source. This is shown in Figure 11-30.

![The order.xml file in the Design view](image1.png)

**Figure 11-29** The order.xml file in the Design view

**The XmlDoc class**

For this example, we create a Java class that encapsulates the XML interface methods. The XmlDoc class is a simple class that contains the code we use to interface with XML in our application. The XmlDoc class uses the DOM interface.

**XmlDoc declaration and constructor**

There are a number of packages we need to import in order to use the DOM parsers. See Example 11-20. The first two are common Java packages. The other three packages are needed in order to use DOM. We declare a private document called `doc`. This is the root node of the tree of our XML document.

**Example 11-20** XmlDoc declaration and constructor

```java
import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.InputSource;

public class XmlDoc
{
    private org.w3c.dom.Document doc = null;

    public XmlDoc()
    {
```
Parsing the XML document
First, we need to get the document tree. A document tree is a tree representation of a well-formed XML document. This is called *parsing the XML document*. We use the `parseXmlUri` method (Example 11-21) to do this. The input `String uri` parameter contains the path and the name of the XML file. We instantiate an `org.w3c.dom.Document` object that we put into the `doc` variable declared in the class definition.

Example 11-21  The `parseXmlUri` method

```java
public boolean parseXmlUri(String uri)
{
    boolean parsed = true;

    try
    {
        DocumentBuilderFactory dF = DocumentBuilderFactory.newInstance();
        DocumentBuilder DOMBuilder = dF.newDocumentBuilder();
        doc = DOMBuilder.parse(uri);
    }
    catch (Exception e)
    {
        parsed = false;
        System.out.println("XmlDoc.parseXmlUri(String):Not parsed:"+e);
    }
    return parsed;
}
```

We also include another method, the `parseXmlString` method (Example 11-22), to parse an XML document. In this case, the input parameter is a `String` that contains the XML file itself. This is useful when you receive an XML file from an input source such as MQSeries.

Example 11-22  The `parseXmlString` method

```java
public boolean parseXmlString(String xmlString)
{
    boolean parsed = true;

    StringReader sr = new StringReader(xmlString);
    InputSource iSrc = new InputSource(sr);
    try
    {
        DocumentBuilderFactory dF = DocumentBuilderFactory.newInstance();
        DocumentBuilder DOMBuilder = dF.newDocumentBuilder();
        doc = DOMBuilder.parse(iSrc);
    }
    catch (Exception e)
    {
        parsed = false;
        System.out.println("XmlDoc.parseXmlString(String):Not parsed:"+e);
    }
    return parsed;
}
```
One of these two methods is the first method we call when using the XmlDoc class. If we get a true return, we have the DOM tree in the doc object.

**Changing the values of the XML tree**

Once we get the XML DOM tree in our private doc object, we need to put the values of the order into the XML fields. For example, customer will became a real customer ID, not the word “customer” as it is in the XML file. We create a method called setValuesXML. This method has a java.util.Properties object as an input parameter and calls the setValuesXML(Node, Properties) method as shown in Example 11-23.

**Example 11-23 The setValuesXML(java.util.Properties) method**

```java
public void setValuesXML(java.util.Properties data) {
    setValuesXML(getDoc(),data);
}
```

A second method, setValuesXML(Node, Properties), contains the code for changing the XML tree. Using a recursive algorithm, it goes through the whole tree and changes all the values of the nodes as shown in Example 11-24.

**Example 11-24 The setValuesXML(Node, java.util.Properties) method**

```java
private void setValuesXML(Node node, java.util.Properties data) {
    int type = node.getNodeType();
    switch (type) {
    case Node.DOCUMENT_NODE:
        setValuesXML(((Document)node).getDocumentElement(), data);
        break;
    case Node.ELEMENT_NODE:
        NodeList children = node.getChildNodes();
        if (children != null) {
            int len = children.getLength();
            for (int i = 0; i < len; i++)
                setValuesXML(children.item(i), data);
        }
        break;
    case Node.TEXT_NODE:
        String valor = data.getProperty(node.getParentNode().getNodeName(), node.getNodeValue());
        node.setNodeValue(valor);
        break;
    }
}
```
We put all the information that we want to change as properties in the Properties object. For example, the XML file contains a field named customer. We update the customer property with the value of the customer:

```java
orderProp.put("customer","0001");
```

Recursion is used in case the node is a DOCUMENT_NODE, to go through all the possible children of an ELEMENT_NODE. When a TEXT_NODE is found, we change its value to the value included in the Properties object.

**Converting the XML data into a String object**

When the XML tree is populated with the correct values, we need to convert it into a String object in order to send it as a message to a data queue or an MQSeries queue. We obtain a String object using the `toString()` method. This is shown in Example 11-25.

**Example 11-25  The toString() method**

```java
public String toString()
{
    String sDoc = "";
    sDoc = toString(getDoc(),sDoc);
    return sDoc;
}
```

The `toString()` method calls a second `toString` method. This second method is a private method and accepts two parameters. The first parameter is an XML node, and the second is the String we want to create. Again we use a recursive algorithm to go through the XML tree and get all the nodes and add them to the String object. The returned String object contains the XML document. The `toString(Node, String)` method is shown in Example 11-26.

**Example 11-26  The toString(Node, String) method**

```java
private String toString(Node node, String sDoc)
{
    int type = node.getNodeType();
    switch (type)
    {
    case Node.DOCUMENT_NODE:
        {
            sDoc = sDoc +
            toString(((Document)node).getDocumentElement(), sDoc);
            break;
        }
    case Node.ELEMENT_NODE:
        {
            sDoc = sDoc + "<" + node.getNodeName();
            NamedNodeMap attrs = node.getAttributes();
            for (int i=0; i< attrs.getLength(); i++)
            {
                Node attr = attrs.item(i);
                sDoc = sDoc + " " + attr.getNodeName() +
            }
        }
    case Node.TEXT_NODE:
        {
            sDoc = sDoc + attrs.item(i);
        }
    default:
        {
            sDoc = sDoc + attrs.item(i);
        }
    }
    return sDoc;
}
```
Other methods
It may be useful to place all the data contained in an XML file into a Properties object. For
that, we create the toProperties() method. This method has a further call to a private
toProperties method (Example 11-27).

Example 11-27  The toProperties() method

```java
public Properties toProperties()
{
    Properties data= new Properties();
    toProperties(doc, data);
    return data;
}
```

The toProperties(Node, Properties) method again uses a recursive algorithm to retrieve
the data. It goes through the whole DOM tree and puts a property into the Properties object
when it finds a TEXT_NODE node.

Example 11-28  The toProperties(Node, Properties) method

```java
private void toProperties(Node node, Properties data)
{
    int type = node.getNodeType();
    switch (type)
    {
    case Node.DOCUMENT_NODE:
    {
        toProperties(((Document)node).getElementsByTagName(), data);
        break;
    }
    case Node.TEXT_NODE:
    {
        sDoc = sDoc + node.getNodeValue();
        break;
    }
    }
    sDoc = sDoc + "\n" + attr.getNodeValue() + "\n";
    }
sDoc = sDoc + ">";

    NodeList children = node.getChildNodes();
    if (children != null)
    {
        int len = children.getLength();
        for (int i = 0; i < len; i++)
            sDoc = toString(children.item(i), sDoc);
    }
    break;
}
```

break;
}
case Node.ELEMENT_NODE:
{
    NodeList children = node.getChildNodes();
    if (children != null)
    {
        int len = children.getLength();
        for (int i = 0; i < len; i++)
            toProperties(children.item(i), data);
    }
    break;
}
case Node.TEXT_NODE:
{
    data.put(node.getParentNode().getNodeName(),node.getNodeValue());
    break;
}
}

---

Enhancing the OrderEntry application with XML

Now we can use the XmlDoc class in our OrderEntry application. We follow these steps:

1. Open the J2EE perspective.
2. Expand EJB Modules.
3. Expand the ToWebSphere EJB Module.
4. Expand the OrderPlacement EJB.
5. Double-click OrderPlacementBean to open it.
6. We add the writeMQXml method as shown in Example 11-29.
7. We change the OrderPlacement EJB placeOrderMQ method to call the writeMQXml method after placing an order.

Example 11-29 The writeMQXml method

```java
private void writeMQXml(String wID, int dID, String cID, float oID) throws Exception
{
    InitialContext initCtx = new InitialContext();
    String hostname = (String)initCtx.lookup("java:comp/env/hostname");
    String channel = (String)initCtx.lookup("java:comp/env/channel");
    String qmanager = (String)initCtx.lookup("java:comp/env/qmanager");
    String queue = (String)initCtx.lookup("java:comp/env/queue");

    XmlDoc xmlDoc = new XmlDoc();
    xmlDoc.parseXmlUri("order.xml");

    Properties orderProp = new Properties();
    orderProp.put("warehouse",wID);
    orderProp.put("district",dID);
    orderProp.put("customer",cID);
    orderProp.put("ordernum",oID);

    xmlDoc.setValuesXML(orderProp);
```
String orderXml = xmlDoc.toString();

MQCon mqcon = new MQCon(hostname, qmanager, channel, queue);
byte[] res = mqcon.putMQ(orderXml);

The `writeMQXml` method receives the order information as input parameters. The logic of the `writeMQXml` method is:

1. Obtain an instance of the `XmlDoc` class named `xmlDoc`.
2. Call the `parseXMlUri` method to parse the `order.xml` XML file.
3. Add the input parameters as properties to a Properties object named `orderProp`.
4. Call the `setValuesXML` method passing in the Properties object to set the values of the order in the XML document.
5. Convert the XML document to a String object.
6. Place the String object on an MQSeries queue.

The name of the XML file contains the complete path its location on the server. In this case, it is in the root of the WebSphere instance directory. If you run the application again and browse the messages in the MQSeries server queue, you see the XML file in it as shown in Figure 11-31.

![Figure 11-31 The XML message in the MQSeries queue](image)

### 11.7 Conclusion

This chapter showed you how Enterprise JavaBeans can interface with other applications.

The IBM Toolbox for Java provides many options for interfacing to legacy applications:

- Distributed Program Call (DPC)
- Data Queues
- Program Control Markup Language (PCML)
We added a new method to the OrderEntryClerk EJB named `placeOrderRPG`. It uses the Distributed Program Call interface to call the ORDENTR RPG program to place an order. A client program can call this method to interface with the RPG program. It doesn’t need to know the details about how the Distributed Program Call interface works. The implementation details are handled by the OrderEntryClerk bean.

We added a new `writeDataQueue` method to the OrderPlacement EJB. It uses the data queue support to interface with the PRTORDERP RPG program. This method is used internally by the OrderPlacement bean. Whenever its `placeOrder` method is called to place an order, it in turn, calls the `writeDataQueue` method. The PRTORDERP RPG program reads the entries off the data queue and prints an order request document. Data queue support only works on the iSeries server.

We added a new `writeMQ` method to the OrderPlacement EJB. It uses MQSeries queues support to interface with a MQSeries Queue Manager and place messages on MQSeries queues. MQSeries allows you to interface with applications running on any platform that supports MQSeries.

We also showed an example that uses XML. XML is useful for passing information from one application to another. An XML document contains not only the data, but a description of what the data means.
The Command package

In this chapter, we modify the EJB-based OrderEntry application to use the Command package. It provides a framework that can improve distributed application performance by providing a mechanism for collecting sets of requests and submitting them as a unit of work. The objective is to reduce the number of individual remote calls. This architecture is very helpful for a 3-tier distributed environment as recommended by the IBM Framework for e-business. In this environment, application logic and business services are separated into components that communicate with each other across a network. Application developers must be aware of the performance impacts of distributing application components across a network.
12.1 The IBM Framework for e-business

As shown in Figure 12-1, IBM Framework for e-business is at the core of the IBM e-business software strategy. IBM developed the Framework for e-business to help customers build, run, and manage successful e-business applications. When customers adopt the Framework, they adopt a proven methodology, products, and tools.

![Figure 12-1  IBM Framework for e-business](image)

Generally speaking, the Framework is a distributed computing model, which assumes that e-business applications comprise multiple executable program modules that run on different hardware and software platforms linked by networking. This model is designed to support clients with high-function Web application and enterprise servers. The Framework for e-business architecture provides a full range of services for developing and deploying e-business applications. Because it is based on industry standards, the Framework has the ability to “plug and play” multiple components provided by any vendor.

Figure 12-2 shows the e-business application model.

![Figure 12-2  The e-business application model](diagram)
The architecture is composed of the following key elements:

- Clients based on a Web browser or Java applet model that enables universal access to Framework applications and on-demand delivery of application components.
- A network infrastructure that provides such services as TCP/IP, directory, and security whose capabilities can be accessed via open, standard interfaces, and protocols.
- Application server software that provides a platform for e-business applications and includes an HTTP server, database and transaction services, mail and groupware services, and messaging services.
- Application integration that allows disparate applications to communicate and provides Web access to existing data and applications.
- A Web application programming environment that provides the server-side Java programming environment for creating dynamic and robust e-business applications.
- e-business application services that provide higher level application specific functionality to facilitate the creation of e-business solutions.
- Systems management functions that accommodate the unique management requirements of network computing across all elements of the system, including users, applications, services, infrastructure, and hardware.
- Development tools to create, assemble, deploy, and manage applications.

12.2 Distributed Java applications

Distributed applications are defined by the ability to use remote resources as if they were local. However, this remote work affects the performance of distributed applications. Distributed applications can improve performance by using remote calls sparingly. If a server does several tasks for a client, the application can run more efficiently if the client bundles requests together, which reduces the number of individual remote calls. The Command package provides a mechanism for collecting sets of requests to be submitted as a unit.

In addition to giving you a way to reduce the number of remote invocations a client makes, the Command package provides a generic way of making requests. A client instantiates the command, sets its input data, and tells it to run. The Command infrastructure determines the target server and passes a copy of the command to it. The server runs the command, sets any output data, and copies it back to the client. The package provides a common way to issue a command, locally or remotely, and independently of the server's implementation. Any server (an enterprise bean, a Java Database Connectivity (JDBC) server, a servlet, and so on) can be a target of a command if the server supports Java access to its resources and provides a way to copy the command between the client's JVM and its own JVM.

This chapter demonstrates how to use the Command framework in the OrderEntry application. The Model-View-Controller architecture is applied to the application to improve the encapsulation of business logic and provide a separation between the modules.

12.3 The Command package

A Web application is essentially a series of interactions between a client and a particular Web site. When the client and server run on different JVMs, the Web application needs to enable remote method invocation (RMI) via server-specific protocols. This can become considerably inefficient in terms of both time and resource, particularly when crossing a network.
The command bean infrastructure allows a command to be run within the environment of a target server. Therefore, multiple accesses by the command to server resources avoid distributed overhead. Any server can be a target if it supports Java access to its resources and provides a protocol to copy the command bean between a Web application JVM and the command server JVM.

Command beans allow the server side of the Web application to be partitioned into efficient units of interaction. The Web application parts, such as the interaction controller and user interface logic, are independent of the style of the command bean’s implementation and independent of where the command bean is physically executed.

Command beans that execute locally (in other words, in the same JVM as the Web application) simply implement the Command interface. If a command bean is to execute remotely on another server, it implements the TargetableCommand interface, which is an extension of the Command interface that allows for remote execution. This is done by extending the TargetableCommandImpl class. Regardless of whether the command bean executes locally or remotely, the JSP or servlet executes the command bean in the same way. If it is conceivable that a command could ever be remote, then it should implement the TargetableCommand interface. Then a change in deployment does not cause a source code change.

With the release of WebSphere 3.5, the Command model is formalized in the Command package (com.ibm.websphere.command) and extended to accommodate command shipping (called TargetableCommand). The concept behind command shipping is to intercept execute() methods, ship the command to a better execution point, execute it, and then ship it back to the caller.

WebSphere 3.5.3 introduces a command target for EJBs named EJBCommandTarget. This is the command target for commands that are executed on an EJB server. It uses a server-side entity bean to allow concurrent access by different transactions to execute a command on the server.


12.3.1 The Command interface

The most basic features of the Command framework are specified in the Command interface, which extends java.io.Serializable. It contains the following three abstract methods:

- execute()
- isReadyToCallExecute()
- reset()

These methods control the life cycle of a command. When a command has been instantiated, it is in the new state. Running a command often requires a set of input properties. The isReadyToCallExecute() method can be used to check whether the requirement of input has been satisfied. It is called on the client side before the command is passed to the server for execution. If the isReadyToCallExecute() method returns true, the command is in the initialized state, and the execute() method is ready to be invoked. After the execute() method is successfully run, the command goes from the initialized state to the executed state. This is demonstrated in Figure 12-3.
The `reset()` method retains the previously set input properties, but resets the output properties of the object back to the original values prior to any execute calls (such as, null or 0). It converts the command from an executed state back to the initialized state. Resetting the output properties prevents exposing outdated output values of previous command executions to later executions.

Resetting the same command instance is convenient if there are several complex input properties that do not need to be changed for the next invocation. This reuse also avoids the overhead of instantiation and garbage collection.

![Figure 12-3   The Command life cycle](image)

### 12.3.2 Facilities for creating commands

The steps to write a command are:

1. Create a command interface.
2. Implement the command.
3. Implement the target.

In practice, most commands implement the `TargetableCommand` interface, which allows the command to be executed remotely. Example 12-1 shows the structure of a command interface for a targetable command.

#### Example 12-1   The structure of an interface for a targetable command

```java
import com.ibm.websphere.command.*;
public interface MyCommand extends TargetableCommand {
    // Declare application methods here
}
```

The interface `TargetableCommand` extends `com.ibm.websphere.command.Command`. In addition to the three methods described previously, the `TargetableCommand` interface declares the methods shown in Table 12-1.

#### Table 12-1   `TargetableCommand` methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setCommandTarget()</code></td>
<td>Specifies the target object of a command.</td>
</tr>
<tr>
<td><code>getCommandTarget()</code></td>
<td>Returns the target object of a command.</td>
</tr>
<tr>
<td><code>setCommandTargetName()</code></td>
<td>Specifies the target name to a command.</td>
</tr>
<tr>
<td><code>getCommandTargetName()</code></td>
<td>Returns the target name of a command.</td>
</tr>
</tbody>
</table>
WebSphere J2EE Application Development for the iSeries Server

<table>
<thead>
<tr>
<th>Methods</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasOutputProperties()</td>
<td>Indicates whether the command has output that must be copied back to the client. The implementation class also provides the setHasOutputProperties() method, for setting the output of this method. By default, the hasOutputProperties() method returns true.</td>
</tr>
<tr>
<td>setOutputProperties()</td>
<td>Saves the output values from the command for return to the specific client.</td>
</tr>
<tr>
<td>performExecute()</td>
<td>Encapsulates the application specific work. It is called by the execute() method declared in the Command interface.</td>
</tr>
</tbody>
</table>

The performExecute() method is the only method that must be implemented by the application developer. The others have been implemented in the TargetableCommandImpl class in the same package, which also implements the execute() method declared in the Command interface.

As we have discussed, a targetable command extends the TargetableCommand interface, which allows the client to direct a command to a particular server. The TargetableCommand interface and the TargetableCommandImpl class provide two ways for a client to specify a target:

- **setCommandTarget() method**: Allows the client to set the target object directly on the command
- **setCommandTargetName() method**: Allows the client to refer to the server by name, an approach that is useful when the client is not directly aware of server objects.

A targetable command also has corresponding getCommandTarget() and getCommandTargetName() methods.

A command needs to identify its target. Because there is more than one way to specify the target and because different applications can have different requirements, the Command package does not specify a selection algorithm. Instead, it provides a TargetPolicy interface with one method, getCommandTarget(), and a default implementation. This allows applications to devise custom algorithms for determining the target of a command when appropriate. The TargetableCommandImpl abstract class provides two methods for setting and returning the target policy associated with the command:

- void setTargetPolicy(com.ibm.websphere.command.TargetPolicy)
- com.ibm.websphere.command.TargetPolicy getTargetPolicy()

### 12.4 Creating a command

In this section, we create a simple command named GetCustomerCmdEJB. The purpose of this command is to retrieve customer information using the Command framework. In this example, we use the EJBCommandTarget class. It is a command target for commands that target Enterprise JavaBeans on an EJB server.

#### 12.4.1 Benefits of the Command package framework

As shown in the previous chapters, the Customer entity bean has methods for retrieving any customer record from the corresponding database table. We could simply obtain an instance of the bean by invoking the findByPrimaryKey() method and then retrieving all the information we need about that customer through getter methods. However, this is not efficient, especially when the client and the EJB run on different JVMs. This is because it...
Chapter 12. The Command package

requires multiple RMI calls. As illustrated in the Figure 12-4, if we rely on the getter methods on the Customer bean, each of these calls is a remote call. With the approach of command shipping, which allows bundling of these calls, we greatly reduce the remote calls and improve performance.

![Diagram](image)

Figure 12-4 Comparing the original method to the command method

12.4.2 Using the EJBCommandTarget class

In this example, we use the EJBCommandTarget class. The EJBCommandTarget class is provided by the WebSphere Command framework to allow us to run a command in a designated server without providing our own command target implementation.

The EJBCommandTarget object serves as a wrapper for an EJB command target. The CommandServerEntityBean is the WebSphere implementation of the EJB CommandTarget class. It is a stateless bean managed persistent bean, so there is no database table associated with this bean. The command developer can set the EJBCommandTarget object as the target for a command. The WebSphere Application Server administrator deploys the CommandServerEntityBean into the application server where the command is executed.

12.4.3 Creating the GetCustomerCmdEJB command

In this section, we define the GetCustomerCmdEJB interface that extends the TargetableCommand class. It defines the business logic methods of the command, as shown in Example 12-2.
Example 12-2  The GetCustomerCmdEJB interface

```java
import com.ibm.websphere.command.*;
import com.ibm.itso.roch.wasaejb.*;

public interface GetCustomerCmdEJB extends TargetableCommand{
    String getAddressLine1();
    String getAddressLine2();
    String getCity();
    Customer getCustomer();
    String getFirstName();
    String getMiddleInitials();
    String getPhone();
    String getState();
    String getZip();
    void setCustomer(Customer cust);
}
```

The GetCustomerCmdEJBImpl class extends the TargetableCommandImpl and implements the GetCustomerCmd interface as shown in Example 12-3.

Example 12-3  Definition of the GetCustomerCmdEJBImpl class

```java
import java.lang.reflect.*;
import com.ibm.websphere.command.*;
import com.ibm.itso.roch.wasaejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import java.util.*;

public class GetCustomerCmdEJBImpl extends TargetableCommandImpl implements GetCustomerCmdEJB {
    private Customer customer = null;
    private String customerID = null;
    private String firstName = null;
    private String lastName = null;
    private String middleInitials = null;
    private String addressLine1 = null;
    private String addressLine2 = null;
    private String city = null;
    private String state = null;
    private String zip = null;
    private String phone = null;
    private float creditLimit = 0;
    private float balance = 0;
    private float yearToDateBalance = 0;
}
```

We place the GetCustomerCmdEJB interface and the GetCustomerCmdEJBImpl class, as well as the other command-related interfaces or classes described in this chapter, into the OrderEntryCommandPkg package.
To implement the business logic defined in the GetCustomerCmdEJB interface, we add the methods defined in the GetCustomerEJBCmd interface to the GetCustomerCmdEJBImpl class as shown in Example 12-4.

**Example 12-4  Implementation of the methods defined in the GetCustomerCmdEJB interface**

```java
public String getFirstName(){
    return firstName;
}
public String getMiddleInitials(){
    return middleInitials;
}
public String getLastName(){
    return lastName;
}
public String getAddressLine1(){
    return addressLine1;
}
public String getAddressLine2(){
    return addressLine2;
}
public String getCity(){
    return city;
}
public String getState(){
    return state;
}
public String getZip(){
    return zip;
}
public String getPhone(){
    return phone;
}
public void setCustomer(Customer cust){
    customer = cust;
}
public Customer getCustomer(){
    return customer;
}
```

The most important method in the GetCustomerCmdEJBImpl class that needs to be implemented is the `performExecute()` method. We define this method as shown in Example 12-5.

**Example 12-5  The performExecute() method in the GetCustomerCmdEJBImpl class**

```java
public void performExecute() throws Exception {
    try {
        Context initCtx = getInitialContext();
        CustomerHome customerHome = (CustomerHome) initCtx.lookup("Customer");
        CustomerKey custKey = new CustomerKey(customerID);
        customer = (Customer) customerHome.findByPrimaryKey(custKey);
        customerID = customer.getCustomerID();
        firstName = customer.getFirstName();
        lastName = customer.getLastName();
        middleInitials = customer.getMiddleInitials();
        addressLine1 = customer.getAddressLine1();
    }
}
```
addressLine2 = customer.getAddressLine2();
city = customer.getCity();
state = customer.getState();
zip = customer.getZip();
phone = customer.getPhone();
setHasOutputProperties(true);
} catch (Exception ex) {
    ex.printStackTrace();
    throw ex;
}
}

The performExecute() method should be programmed using the server programming model. The other methods should be programmed as if server facilities are not available.

We obtain an InitialContext object and look up the home interface of the Customer EJB. We use the findByPrimaryKey method to obtain an instance of the Customer bean.

The last method in the GetCustomerCmdEJBImpl class that must be implemented is the isReadyToCallExecute() method as shown in Example 12-6. It tells the Command infrastructure when the execute() method is ready to be invoked.

Example 12-6  The isReadyToCallExecute() method in the GetCustomerCmdEJBImpl class

```java
public boolean isReadyToCallExecute() {
    if (customerID != null)
        return true;
    else
        return false;
}
```

As discussed previously, implementation of the reset() method is optional. It helps to reuse the command and to save the overhead of instantiating a new command and garbage collecting the old instance. We define the reset() method as shown in Example 12-7.

Example 12-7  The reset() method in the GetCustomerCmdEJBImpl class

```java
public void reset() {
    customer = null;
    customerID = null;
    firstName = null;
    lastName = null;
    middleInitials = null;
    addressLine1 = null;
    addressLine2 = null;
    String city = null;
    String state = null;
    String zip = null;
    String phone = null;
    creditLimit = 0;
    balance = 0;
    yearToDateBalance = 0;
}
```
To accept a command, the target must implement the CommandTarget interface. In this example, the EJBCommandTarget class implements the CommandTarget interface. The Command package provides mechanisms for determining where to run the command.

A target policy associates a command with a target. Later, you will see how to use a target policy to associate the getCustomerCmdEJB command with the EJBCommandTarget class as its target. The the implementor of the CommandTarget interface (the EJBCommandTarget class, in our example) is responsible for ensuring the proper execution of a command in the desired target server environment. EJBCommandTarget is a command target for commands that are executed in an EJB server. It provides the client-side RMI/IIOP stub for the server-side entity bean that runs the command on the server.

12.4.4 Using the GetCustomerCmdEJB command

We test the GetCustomerCmdEJB command using a JSP and servlet. The servlet demonstrates how to use the command from a client application. We follow these steps:

1. Create a JavaServer Page file named RetrieveCustomerInfo.jsp. It displays an HTML page that allows a user to enter a customer ID.

2. After the user enters a customer ID and clicks the Submit button, the JSP invokes the GetCustomerServlet class. It creates an instance of the GetCustomerCmdEJB command, setting its target as EJBCommandTarget.

3. The command is executed on the iSeries EJB server.

4. The command is returned to the client with customer information. The customer information is displayed on the browser.

The JSP and servlet code are for the purpose of testing the remote execution of the command. They are not a part of the OrderEntry application. Later we show how to use the commands in the OrderEntry application. Example 12-8 shows the JSP code.

```html
<html><head>
<title>Retrieve Customer Information</title>
</head><body><center>
<p></p>
<p></p>
<p><b><font face="Arial" size="3" color="#0000FF">Please enter Customer ID below:</font></b><br>
<form method="POST" action="/OrderEntry2/GetCustomerServlet">
<p align="center"> <input type="text" name="CustomerID" size="20"><br><br>
<input type="submit" value="Submit" name="Submit" type="Reset" name="Reset"></p>
</center></form></body></html>
```

The action of the Submit button is to execute GetCustomerServlet, which is installed in the OrderEntry Web application.

Example 12-9 shows the code for the init() method for the GetCustomerServlet class.
Example 12-9  The code for the GetCustomerServlet init method

```java
public void init(ServletConfig config) throws ServletException {
    get_custCmd = new GetCustomerCmdEJBImpl();
    policy = (TargetPolicyDefault) get_custCmd.getTargetPolicy();
    policy.registerCommand(
        "OrderEntryCommandPkg.GetCustomerCmdEJBImpl",
        "com.ibm.websphere.commandUtil.EJBCommandTarget";
    }
}
```

In the `init()` method, we create an instance of the GetCustomerEJBImpl class named `get_custCmd`. We create a new TargetPolicyDefault object named `policy` and register the EJBCommandTarget class as the command target for the GetCustomerCmdEJB command. Since targetable commands can be run remotely in another JVM, the Command package provides mechanisms for determining where to run the command. A target policy associates a command with a target and is specified through the TargetPolicy interface.

We can design customized target policies by implementing this interface, or we can use the provided TargetPolicyDefault class. If we use this default implementation, the command determines the target by looking through an ordered sequence of four options:

1. The CommandTarget value
2. The CommandTargetName value
3. A registered mapping of a target for a specific command
4. A defined default target

If it finds no target, it returns null.

The TargetPolicyDefault class provides methods for managing the assignment of commands with targets (registerCommand, unregisterCommand, and listMappings), and a method for setting a default name for the target (setDefaultTargetName). The default target name is `com.ibm.websphere.command.LocalTarget`, where `LocalTarget` is a class that runs the command's `performExecute` method locally.

Example 12-10 shows the code for the `doPost()` method.

Example 12-10  Code for the GetCustomerServlet doPost method

```java
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String newCustID = request.getParameter("CustomerID");
    get_custCmd = new GetCustomerCmdEJBImpl(policy);
    try {
        get_custCmd.setCustomerID(newCustID);
        get_custCmd.execute();
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>";
    out.println("<head><title>Get Customer Information Servlet</title></head>";
    out.println("<body>";
    out.println("<p><center><b><font face="Arial" size="6" color="Blue">" +
        get_custCmd.getFirstName() + " " +
        get_custCmd.getLastName()
    out.println("</p>";
    }
```
In the doPost method, we create an instance of the command, called \textit{getCustCmd}. We initialize the command's parameters and execute it. This causes the \textit{performExecute} method of the command to be invoked.

Since we registered EJBCommandTarget as the target for the command, the command is copied to the target server. The server runs the command, sets any output data, and copies it back to the client. When this is finished, the returned command contains the required customer information.

Assume the application has been deployed on WebSphere Application Server, run the RetrieveCustomerInfo JSP file by opening a browser session and entering:

http://sysname:port/OrderEntry2/RetrieveCustomerInfo.jsp

Enter a customer ID in the entry field and click \textbf{Submit}. The display appears like the example in Figure 12-5.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure12-5.png}
\caption{Entering the customer ID in the JSP file}
\end{figure}

If it works correctly, we see the first name and last name of the customer displayed on the browser (Figure 12-6). The servlet demonstrates that a command can be shipped to and executed on the EJB server, as well as carry any outputs back to the client.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure12-6.png}
\caption{Displaying the customer information}
\end{figure}
12.5 Creating your own command targets

The Command framework provides a number of ways to create and implement commands. This section examines additional ways to use commands with Enterprise JavaBeans. Rather than using the EJBCommandTarget implementation, we can design commands that directly interface with our EJBs. You will see an example of using a custom target policy (CustomTargetPolicy).

We create a command named GetCustomerCmd, which is similar to GetCustomerCmdEJB, but directly interfaces with the Customer EJB. We also create the OrderPlacementCmd command, which uses the OrderPlacement session bean to place orders.

12.5.1 Constructing the GetCustomerCmd command and its target

The overall design in this example is to first construct an instance of the command, look up the Customer bean, and instantiate an instance of it using the findByPrimaryKey() method. Then a reference to this Customer object is assigned to the command so that the bean instance becomes the command target. The command is copied to the target server. The server runs the command, sets any output data, and copies it back to the client. When this is finished, the returned command copy contains the needed customer information.

First, we define the GetCustomerCmd interface that extends TargetableCommand. It defines the business logic methods of the command as shown in Example 12-11.

Example 12-11  The GetCustomerCmd interface

```java
import com.ibm.websphere.command.*;
import com.ibm.itso.roch.wasaejb.*;

public interface GetCustomerCmd extends TargetableCommand {
    String getAddressLine1();
    String getAddressLine2();
    String getCity();
    Customer getCustomer();
    String getFirstName();
    String getMiddleInitials();
    String getPhone();
    String getState();
    String getZip();
    void setCustomer(Customer cust);
}
```

The GetCustomerCmdImpl class extends TargetableCommandImpl and implements the GetCustomerCmd interface as shown in Example 12-12.

Example 12-12  GetCustomerCmdImpl and its variables declaration

```java
import java.lang.reflect.*;
import com.ibm.websphere.command.*;
import com.ibm.itso.roch.wasaejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import java.util.*;
import java.lang.*;

public class GetCustomerCmdImpl extends TargetableCommandImpl implements GetCustomerCmd {
```
private Customer customer = null;
private String customerID = null;
private String firstName = null;
private String lastName = null;
private String middleInitials = null;
private String addressLine1 = null;
private String addressLine2 = null;
private String city = null;
private String state = null;
private String zip = null;
private String phone = null;
private float creditLimit = 0;
private float balance = 0;
private float yearToDateBalance = 0;

Now the question is how to initialize a GetCustomerCmdImpl instance. We define the constructor of the class as shown in Example 12-13. This constructor references a TargetPolicy object as a parameter. Then it calls the setTargetPolicy() methods inherited from the TargetableCommandImpl class to set the target policy, which associates the command with a specified target object.

Example 12-13  Constructor of the GetCustomerCmdImpl
public GetCustomerCmdImpl(CommandTarget target, TargetPolicy targetPolicy) {
    setTargetPolicy(targetPolicy);
}

As discussed earlier, there is more than one way to specify a command target. In this application, we adopt the approach of defining our own CustomTargetPolicy class that implements the TargetPolicy and the Serializable interfaces. Example 12-14 shows the definition of the CustomTargetPolicy used for targeting the Customer entity bean.

Example 12-14  Definition of CustomTargetPolicy used by the Customer command
import java.io.*;
import java.util.*;
import java.beans.*;
import com.ibm.websphere.command.*;

public class CustomTargetPolicy implements TargetPolicy, Serializable{

    public CommandTarget getCommandTarget (TargetableCommand command) {
        CommandTarget target = null;
        try {
            target = (CommandTarget) Beans.instantiate(null,
                    "com.ibm.itso.roch.wasaejb.CustomerBean");
        } catch (java.io.IOException ioEx) {
            ioEx.printStackTrace();
        }
    }
}
In the getCommandTarget() method, we use the static instantiate() method of the beans class defined in the java.beans package to search for CustomerBean and to instantiate an instance of it. A reference to the instance is returned. This instance becomes the target of the GetCustomerCmd command. As shown in Example 12-13, a reference of a CustomTargetPolicy object is passed into the GetCustomerCmdImpl constructor so the command knows its target of execution.

To implement the business logic defined in the GetCustomerCmd interface, we add the methods defined in the GetCustomerCmd interface to the GetCustomerCmdImpl class as shown in Example 12-15.

Example 12-15  Implementation of the methods defined in the GetCustomerCmd interface

```java
public String getFirstName(){
    return firstName;
}
public String getMiddleInitials(){
    return middleInitials;
}
public String getLastName(){
    return lastName;
}
public String getAddressLine1(){
    return addressLine1;
}
public String getAddressLine2(){
    return addressLine2;
}
public String getCity(){
    return city;
}
public String getState(){
    return state;
}
public String getZip(){
    return zip;
}
public String getPhone(){
    return phone;
}
public void setCustomer(Customer cust){
    customer = cust;
}
public Customer getCustomer(){
    return customer;
}
```

Next, we create the performExecute() method. We define this method as shown in Example 12-16. We explain the invocation of this method later when we discuss how the command is used in the OrderEntry application.
**Example 12-16  The performExecute() method in the GetCustomerCmdImpl class**

```java
public void performExecute() throws Exception {
    Customer customer = getCustomer();
    customerID = customer.getCustomerID();
    firstName = customer.getFirstName();
    lastName = customer.getLastName();
    middleInitials = customer.getMiddleInitials();
    addressLine1 = customer.getAddressLine1();
    addressLine2 = customer.getAddressLine2();
    city = customer.getCity();
    state = customer.getState();
    zip = customer.getZip();
    phone = customer.getPhone();
}
```

The last method in the GetCustomerCmdImpl class that must be implemented is `isReadyToCallExecute()` as shown in Example 12-17. It tells the Command infrastructure when the `execute()` method is ready to be invoked.

**Example 12-17  The isReadyToCallExecute() method in the GetCustomerCmdImpl class**

```java
public boolean isReadyToCallExecute() {
    if (customer != null)
        return true;
    else
        return false;
}
```

As discussed earlier, implementation of the `reset()` method is optional. It helps to reuse the command and to save the overhead of instantiating a new command and garbage collecting the old instance. We define the `reset()` method as shown in Example 12-18.

**Example 12-18  The reset() method in the GetCustomerCmdImpl class**

```java
public void reset() {
    customer = null;
    customerID = null;
    firstName = null;
    lastName = null;
    middleInitials = null;
    addressLine1 = null;
    addressLine2 = null;
    String city = null;
    String state = null;
    String zip = null;
    String phone = null;
    creditLimit = 0;
    balance = 0;
    yearToDateBalance = 0;
    targetPolicy = new TargetPolicyDefault();
}
```
To accept a command, the target must implement the CommandTarget interface. The implementor of the CommandTarget interface (the Customer entity bean in this example) is responsible for ensuring the proper execution of a command in the desired target server environment. Therefore, before we ship the command to its target server for execution, we have to modify the CustomerBean entity bean so that it implements the CommandTarget interface and its single `executeCommand()` method as shown in Example 12-19. Note this method must be added to the remote interface so that it can be seen by the client. Further, we need to re-generate the deployed code and deploy the modified Customer bean to the server to replace the original Customer bean.

Example 12-19  Implementation of the CommandTarget interface by the CustomerBean

```java
import com.ibm.websphere.command.*;
import OrderEntryCommandPkg.*;

public class CustomerBean implements EntityBean, CommandTarget {
    // Original code

    public TargetableCommand executeCommand (TargetableCommand command) throws RemoteException, CommandException {
        try {
            command.performExecute();
        } catch (Exception ex) {
            if (ex instanceof RemoteException) {
                RemoteException remoteException = (RemoteException) ex;
                if (remoteException.detail != null){
                    throw new CommandException (remoteException.detail);
                } throw new CommandException (ex);
            }
        }
        if (command.hasOutputProperties()) {
            return command;
        }
        return null;
    // Original code
}
```

As shown, this method references a TargetableCommand as a parameter and returns it after calling the `performExecute()` method of the command. Therefore, we can put all the business logic in the `performExecute()` method. The Command package infrastructure will copy the command to the target server where the `performExecute()` method is invoked.

Using the GetCustomerCmd command in the OrderEntry application

Now that the command has been initialized and its target is set, we are ready to ship the command to the target server for execution. The Command package provides a convenient way to do this.

1. We simply call the `execute()` method in the command. Remember `execute()` is defined in the Command interface and is implemented in the TargetableCommandImpl class.

2. When `execute()` is invoked, a copy of the command is shipped to its target server. The `executeCommand()` method on the target will be called automatically.
3. After that, the command is shipped back to the client. In our example, the Customer command is copied to the Customer bean, populated with the customer information, and carried back by the Command infrastructure to the client site where the command was instantiated.

To use the command in the context of the OrderEntry application, we create a Java servlet called OrderEntryControllerServlet. We place this servlet and the others described in this chapter into a package called tservlets (Figure 12-7). The details of the OrderEntryControllerServlet servlet and the Model-View-Controller architecture are discussed later. For now, we focus on how to run the GetCustomerCmd command.

The init() method of the OrderEntryControllerServlet first obtains an initial context object by calling the getInitialContext() method. It then looks up the CustomerHome home interface (Example 12-20). Therefore, after the servlet is started, a reference to the CustomerHome is already available.

Example 12-20   The init() and the getInitialContext() methods in the OrderEntryControllerServlet

```java
public void init(ServletConfig config) throws ServletException {
    super.init(config, SuperServlet.SYSTEM);
    system = config.getInitParameter("system");
    userid = config.getInitParameter("userid");
    password = config.getInitParameter("password");
    port = config.getInitParameter("port");

    Context initCtx = null;
    try{
        initCtx = getInitialContext();
    }
    catch (Exception ex) {
        System.out.println("Error in getting initial context: " + ex.getMessage());
    }
    try {
        cHome = (CustomerHome) initCtx.lookup("Customer");
    }
    catch (NamingException ne) {
        System.out.println("Error in looking up Customer entity bean: " + ne.getMessage());
    }
}

private static Context getInitialContext() throws Exception {
    Properties p = new Properties();
    try {
        p.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://" + getSystem() + ":" + getPort());
        InitialContext cx = new InitialContext(p);
        return cx;
    }
    catch (Exception e) {
        System.out.println("Error creating Context " + e.getMessage());
        e.printStackTrace();
        throw (e);
    }
```
Example 12-21 shows how to instantiate and run the GetCustomerCmd command:

1. We declare an instance of the command and obtain a reference to the remote Customer bean by calling the private method getCustomerByID().

2. We obtain a new CustomTargetPolicy object and instantiate a GetCustomerCmd command by passing the new CustomTargetPolicy instance into the GetCustomerCmdImpl constructor.

3. After the command is created, we call the setCustomer() method followed by the execute() method. As described earlier, calling the execute() method on a targetable command automatically invokes the executeCommand() method on the target object. Therefore, when the command is copied back to the client, it contains the customer information we need.

```java
GetCustomerCmd getCustCmd = null;
com.ibm.itso.roch.wasaejb.Customer customer = null;
try {
    customer = getCustomerByID(newCustID);
} catch (RemoteException re) {
    System.out.println(re.getMessage());
}
CustomTargetPolicy customPolicy = new CustomTargetPolicy();
getCustCmd = new GetCustomerCmdImpl(customPolicy);
try {
    getCustCmd.setCustomer(customer);
    getCustCmd.execute();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
getCustCmd.reset();
```

The update customer command

We create a second command that works with the Customer EJB. It is named UpdateCustomerCmd. Its implementation is done in the UpdateCustomerCmdImpl class. They are placed in the OrderEntryCommandPkg. The purpose of this command is to update customer information based on information passed in from a client application. It is similar to the GetCustomerCmd command and provides the same advantages.

12.5.2 The OrderPlacementCmd command

This command is designed to place an order by invoking the OrderPlacement session bean. The constructor of the OrderPlacementCmdImpl class (Example 12-22) is similar to the constructor of the GetCustomerCmdImpl class in that they both set the command target by instantiating a TargetPolicy object. After setting the command target, it obtains the next order ID by calling the getNextOrderID() private method and looks up the OrderPlacement bean by invoking the lookupOrderPlacement() method.
Example 12-22  The constructor of the OrderPlacementCmdImpl class
public OrderPlacementCmdImpl(CommandTarget target) {
    TargetPolicy targetPolicy = new CustomTargetPolicy2();
    setTargetPolicy (targetPolicy);

    // Get the next order id
    try {
        getNextOrderID();
    }
    catch (RemoteException re){
        System.out.println("Fail to get next order id: " + re.getMessage());
    }
    lookupOrderPlacement();
}

Example 12-23 and Example 12-24 show the getNextOrderID() method and the lookupOrderPlacement method, respectively.

Example 12-23  The getNextOrderID() method
private void getNextOrderID() throws RemoteException {

    com.ibm.itso.roch.wasaejb.District district = null;
    Context initCtx = null;
    DistrictHome dHome = null;

    // Get the initial context
    try{
        initCtx = getInitialContext();
    }
    catch (Exception ex) {
        System.out.println("Error in getting initial context: " + ex.getMessage());
    }
    // Look up District
    try {
        dHome = (DistrictHome) initCtx.lookup("District");
    }
    catch (NamingException ne) {
        System.out.println("Error in looking up District entity bean: " + ne.getMessage());
    }

    // Find by primary key for District
    try {
        DistrictKey distKey = new DistrictKey(dID, wID);
        district = (com.ibm.itso.roch.wasaejb.District) dHome.findByPrimaryKey(distKey);
    }
    catch (Exception e) {
        throw new RemoteException(e.getMessage());
    }
    // Return the next order id
    nextOrderID = district getNextOrderId(true);
}
Example 12-24  The lookupOrderPlacement() method

private void lookupOrderPlacement() {
    Context initCtx = null;
    OrderPlacementHome opHome = null;
    try {
        initCtx = getInitialContext();
    } catch (Exception ex) {
        System.out.println("Error in getting initial context: "+ ex.getMessage());
    }
    try {
        opHome = (OrderPlacementHome)initCtx.lookup("OrderPlacement");
    } catch (NamingException ne) {
        System.out.println("Error in looking up OrderPlacement: "+ ne.getMessage());
    }
    try {
        orderPlacement = opHome.create();
    } catch (Exception ex) {
        System.out.println("Error in creating orderPlacement: "+ ex.getMessage());
    }
}

After we instantiate an instance of the OrderPlacementCmdImpl class, the instance already contains a new order ID and a reference to the remote OrderPlacement session bean. This is a modification to the original Order Entry design. We separate getting the new order ID from the order placement processing to avoid the situation where the District table is locked while the order placement is in process. Example 12-25 shows the performExecute() method. It simply executes the placeOrder() method on the target, which is the OrderPlacement bean instance.

Example 12-25  The performExecute() method in the OrderPlacementCmdImpl class

public void performExecute() throws Exception {
    orderPlacement.placeOrder(wID, dID, customerID, orderLines, nextOrderID);
}

12.6 Applying the Model-View-Controller architecture

This section looks at the Model-View-Controller (MVC) architecture. It explains how to apply it to the OrderEntry application and how the Command framework fits into the picture. The logic of the OrderEntry application is similar with the application we developed in 5.3, “Developing a new application with Application Developer” on page 195, with Application Developer. The difference is that we use normal Java classes to access the database in that chapter, while here we use Enterprise JavaBeans.

12.6.1 Re-designing the OrderEntry application using the MVC architecture

The MVC architecture was designed to reduce the programming effort required to build systems that make use of multiple, synchronized presentations of the same data. Its central characteristics are that the model, the controllers, and the views are treated as separate entities, and that changes made to the model should be reflected automatically in the views.
We can readily apply the MVC framework to a Web application. The idea is to split the application into three sections. A servlet handles any requests from the browser and acts as a controller. We put the business logic (the model) in Enterprise JavaBeans. Finally, we use JavaServer Pages to display the view. Figure 12-8 provides a high level overview of this architecture.

To apply the MVC architecture to the OrderEntry application, we use one servlet, the OrderEntryControllerServlet servlet, to handle all the user requests passed from the Web browser. Example 12-26 shows the doPost() method for this servlet.

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {
    HttpSession session = request.getSession(false);
    response.setContentType("text/html");
    response.setHeader("Pragma","no-cache");
    response.setHeader("Cache-Control","no-cache, must-revalidate");
    String action = null;
    String customerID = null;
    action = request.getParameter("action");
    customerID = request.getParameter("customerID");
    PrintWriter out = response.getWriter();
    if (session == null) {
        out.println("<BIG>Session object not available: getSession returned null.</BIG>");
        return;
    }
    if (action.equals("AddToCart")){
        addToCart(session, request, response, customerID);
    }
    else if (action.equals("ShowCart")){
        displayCart(customerID, response, session);
    }
}
```

Figure 12-8   The Model-View-Controller architecture

To apply the MVC architecture to the OrderEntry application, we use one servlet, the OrderEntryControllerServlet servlet, to handle all the user requests passed from the Web browser. Example 12-26 shows the doPost() method for this servlet.
else if (action.equals("CheckOut")){
    placeOrder(session, response);
}
else if (action.equals("UpdateCustomerInformation")){
    updateCustomer(request, response, session);
}
else if (action.equals("ContinueShopping")){
    response.sendRedirect(response.encodeRedirectURL("StartOrderEntry.jsp"));
}
else if (action.equals("ShopSomeMore")){
    Vector itemVector = (Vector)(session.getAttribute("sessionlist.items"));
    String custID = (String)(session.getAttribute("customerID"));
    if (session != null)
        session.invalidate();
    session = request.getSession(true);
    session.setAttribute("sessionlist.items", itemVector);
    response.sendRedirect(response.encodeRedirectURL("StartOrderEntry.jsp"));
}

This servlet performs various tasks depending on the requested action by the user. For example, if the user wants to purchase their selected items, the doPost() method calls the placeOrder() method. It processes the order and redirects the screen to the CheckOut JavaServer Page. The CheckOut JSP displays a message Your order has been processed, along with the order number. Therefore, the OrderEntryControllerServlet plays a role as a controller. It invokes the appropriate EJBs at run time to handle business logic and delegates the presentation logic to the JSPs.

We now discuss, in detail, the MVC-based version of the OrderEntry application that incorporates the techniques of the Command package. The application starts by calling a servlet named StartOrderEntryServlet. This servlet is different from the OrderEntryControllerServlet servlet and does not play the role of a controller. It simply invokes the OrderEntryClerk session bean to obtain all the available items that can be ordered. Once the item list is available, it calls the StartOrderEntry JSP to display these items, as shown in Figure 12-9.
The user is prompted to enter their customer ID if they want to order any items. The Add To Cart and Show Cart buttons are disabled until the user enters their customer ID and selects at least one item.

If the user clicks the AddToCart button, a hidden field named action in the StartOrderEntry JSP is assigned the value AddToCart. Therefore, when the OrderEntryControllerServlet is invoked, it knows what action the user wants to perform. As shown in Example 12-26, the doPost() method calls the addToCart() method in response to the AddToCart action. Example 12-27 shows the addToCart() method.

```
private void addToCart(HttpSession session, HttpServletRequest request, HttpServletResponse response, String customerID) {
    ShoppingCart cart = null;
    try {
        cart = (ShoppingCart) session.getAttribute("shopcart.selected");
    } catch (ClassCastException c) {}  
    if (cart == null) {
        cart = new ShoppingCart();
        flexLog("cart did not exist in session at this time");
    } else {
        flexLog("cart retrieved from session successfully, it has " + cart.getItems().size() + " elements");
    }
```

Example 12-27 The addToCart () method in the OrderEntryControllerServlet
Vector parts = (Vector) session.getAttribute("sessionlist.items");

String[] value = request.getParameterValues("index");
String[] quantity = request.getParameterValues("quantity");

if (value != null) {
    int j = 0;
    for (int i = 0; i < value.length; i++) {
        j = Integer.parseInt(value[i]);
        String[] data = (String[]) parts.elementAt(j);

        Integer qty = new Integer (quantity[j]);
        CartItem aCartItem = new CartItem(data[0], data[1], data[2], data[3], qty);
        cart.getItems().addElement(aCartItem);
    }
    session.setAttribute("shopcart.selected", cart);
    displayCart(customerID, response, session);
}

The addToCart method reads the selected items and their quantities from the user inputs and adds each item to the shopping cart, which is a vector of the CartItem objects. Then it calls the displayCart() method to display the selected items and the customer information as shown in Figure 12-10.
The `displayCart()` method in the `OrderEntryControllerServlet` (Example 12-28) first obtains the `Customer` bean instance based on the customer ID. Then it retrieves the customer information using the `GetCustomerCmd` command, which we discussed earlier. Finally, it directs the outputs to the `OutputCart JSP` to display the shopping cart content and the customer contact information.

**Example 12-28  The displayCart() method in the OrderEntryControllerServlet**

```java
private void displayCart(String newCustID, HttpServletResponse response, HttpSession session) {
    session.setAttribute("custUpdateFlag", "false");
    if (newCustID != null && !newCustID.equals("")) {
        session.setAttribute("customerID", newCustID);
        GetCustomerCmd getCustCmdImpl = null;
        com.ibm.itso.roch.wasaejb.Customer customer = null;
        try {
            customer = getCustomerByID(newCustID);
            session.setAttribute("customerObj", customer);
        } catch (RemoteException re) {
            System.out.println(re.getMessage());
        }
        CustomTargetPolicy customPolicy = new CustomTargetPolicy();
        getCustCmdImpl = new GetCustomerCmdImpl(customPolicy);
        try {
            getCustCmdImpl.setCustomer(customer);
            getCustCmdImpl.execute();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        session.setAttribute("cFirstName", getCustCmdImpl.getFirstName());
        session.setAttribute("cMiddleInitials", getCustCmdImpl.getMiddleInitials());
        session.setAttribute("cLastName", getCustCmdImpl.getLastName());
        session.setAttribute("cAddressLine1", getCustCmdImpl.getAddressLine1());
        session.setAttribute("cAddressLine2", getCustCmdImpl.getAddressLine2());
        session.setAttribute("cCity", getCustCmdImpl.getCity());
        session.setAttribute("cState", getCustCmdImpl.getState());
        session.setAttribute("cZip", getCustCmdImpl.getZip());
        session.setAttribute("cPhone", getCustCmdImpl.getPhone());
        getCustCmdImpl.reset();
    }
    try {
        response.sendRedirect(response.encodeRedirectURL("/OutputCart.jsp"));
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```
If the user wants to update their contact information, they can enter the new data and then click the Update Customer Information button. In response to this request, the OrderEntryControllerServlet invokes the updateCustomer() method (Example 12-29).

**Example 12-29 The updateCustomer() method in the OrderEntryControllerServlet class**

```java
private void updateCustomer(HttpServletRequest request, HttpServletResponse response, HttpSession session) {
    String custID = (String)(session.getAttribute("customerID"));

    String firstName = request.getParameter("firstName");
    String middleInitials = request.getParameter("middleInitials");
    String lastName = request.getParameter("lastName");
    String addressLine1 = request.getParameter("addressLine1");
    String addressLine2 = request.getParameter("addressLine2");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");
    String phone = request.getParameter("phone");

    UpdateCustomerCmd updateCustCmdImpl = null;

    CustomTargetPolicy customPolicy = new CustomTargetPolicy();
    updateCustCmdImpl = new UpdateCustomerCmdImpl(customer, customPolicy);

    updateCustCmdImpl.setFirstName(firstName);
    updateCustCmdImpl.setMiddleInitials(middleInitials);
    updateCustCmdImpl.setLastName(lastName);
    updateCustCmdImpl.setAddressLine1(addressLine1);
    updateCustCmdImpl.setAddressLine2(addressLine2);
    updateCustCmdImpl.setCity(city);
    updateCustCmdImpl.setState(state);
    updateCustCmdImpl.setZip(zip);
    updateCustCmdImpl.setPhone(phone);

    try{
        updateCustCmdImpl.setCustomer(customer);
        updateCustCmdImpl.execute();
        session.setAttribute("custUpdateFlag", "true");

        session.setAttribute("cFirstName", firstName);
        session.setAttribute("cMiddleInitials", middleInitials);
        session.setAttribute("cLastName", lastName);
        session.setAttribute("cAddressLine1", addressLine1);
        session.setAttribute("cAddressLine2", addressLine2);
        session.setAttribute("cCity", city);
        session.setAttribute("cState", state);
        session.setAttribute("cZip", zip);
        session.setAttribute("cPhone", phone);
    }
    catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```

try {
    response.sendRedirect(response.encodeRedirectURL("/OutputCart.jsp"));
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
}

Similar to the retrieval of customer information, we use a command named `UpdateCustomerCmd` to update the customer contact information. The approach allows us to reset all the customer data when we ship the command to the Customer entity bean and run it on the server. It saves many RMI calls compared to directly calling the setter methods on the Customer bean. Example 12-30 shows the `performExecute()` method in the `UpdateCustomerCmdImpl` class.

```
Example 12-30  The performExecute() method in the UpdateCustomerCmdImpl class

public void performExecute() throws Exception {
    Customer customer = getCustomer();
    customer.setFirstName(firstName);
    customer.setLastName(lastName);
    customer.setMiddleInitials(middleInitials);
    customer.setAddressLine1(addressLine1);
    customer.setAddressLine2(addressLine2);
    customer.setCity(city);
    customer.setState(state);
    customer.setZip(zip);
    customer.setPhone(phone);
}
```

After we update the customer information, the message Your contact information has been updated appears on the same screen (Figure 12-11).
At this point, if the user wants to check out their items by clicking the Check Out button, the doPost() method in the OrderEntryControllerServlet calls the placeOrder() method shown in Example 12-31 to process the order.

Example 12-31  The placeOrder() method in the OrderEntryControllerServlet

```java
private void placeOrder (HttpSession session, HttpServletResponse response) throws IOException {
    OrderPlacementCmdImpl opCmdImpl = new OrderPlacementCmdImpl(null);

    Vector orderLines = new Vector();
    ShoppingCart cart = (ShoppingCart)session.getAttribute("shopcart.selected");
    String custID = (String)session.getAttribute("customerID");
    Vector cartItems = cart.getItems();

    if (cartItems.size() > 0) {
        for (int i = 0; i < cartItems.size(); i++) {
            CartItem citem = (CartItem) cartItems.elementAt(i);

            com.ibm.itso.roch.cpwejb.interfaces.OrderDetail thisOrderDetail = new com.ibm.itso.roch.cpwejb.interfaces.OrderDetail(citem.getItemId(), Float.valueOf(citem.getPrice().replace('$','0')).floatValue(), citem.getItemQuantity().intValue());

            orderLines.addElement(thisOrderDetail);
        }
    }

    opCmdImpl.setOrderLines (orderLines);
```

![Image of a customer contact form](image-url)
opCmdImpl.setCustomerID(custID);
}
try {
    opCmdImpl.execute();
    session.setAttribute("orderNumber", opCmdImpl.getNewOrderID() + ");
    response.sendRedirect(response.encodeRedirectURL("/CheckOut.jsp");
} catch (Exception ex){
    ex.printStackTrace();
}

As we previously discussed, we process the order request through the OrderPlacementCmd command. After the command is successfully run on the OrderPlacement session bean, the program sets the new order ID as a session variable and displays it on the next screen using the CheckOut JSP (Figure 12-12).

![Figure 12-12  Displaying the order number](image)

Finally, if the user wants to shop for more items, they can select the Shop Some More... button. In response to this, the OrderEntryControllerServlet terminates the existing session, re-initiates a new session, and associates with it the item list. It then redirects the page to the StartOrderEntry JSP.

### 12.7 Conclusion

There are two major problems that the Command framework attempts to address. One problem is performance. The granularity of artifacts on the server (such as objects, tables, procedure calls, files, and so on) often causes a single client-initiated business logic request to involve several round-trip messages between the client and server. This might entail extra calls to perform the business task and then impose additional calls to retrieve the results of that task.
If the client and target server are not in the same JVM, these calls go between processes and are, therefore, expensive in terms of computer resources. If the calls must go over a network, they are even more costly. To prevent unnecessary delays and improve application performance, it is advantageous to perform a business task in as few interactions between the client and server sides as is natural to the task. Command beans provide the necessary building blocks to achieve this.

A second problem is that there are several possible styles for how business logic can be implemented. This includes EJB, JDBC direct database access, JDBC access to stored procedures, the Common Connector Framework, file system access, and so on. In addition to different implementation programming models, each of these styles has a different way to invoke a request to execute business logic. Because the Command framework is generic and extensible, it can hide all these different types of server invocation mechanisms under a simple and uniform mechanism.

We used the Command framework to enhance the OrderEntry application. First, we used the GetCustomerCmd command to bundle multiple client requests together to retrieve information with greater efficiency. Second, we applied a similar approach to updating customer information and processing order requests.
Additional material

This Redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this Redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246559

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the Additional materials and open the directory that corresponds with the Redbook form number SG246559.

Using the Web material

The additional Web material that accompanies this Redbook includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sg246559ins.zip</td>
<td>A zip file that contains all the example applications</td>
</tr>
<tr>
<td>SG246559.dat</td>
<td>VisualAge for Java code in a repository file</td>
</tr>
<tr>
<td>readme.pdf</td>
<td>Instructions for restoring the programming examples</td>
</tr>
</tbody>
</table>

System requirements for downloading the Web material

The following system configuration is recommended:

- **Hard disk space:** 10 MB is required for download material
- **Operating System:** Windows 98, NT, or 2000
- **Processor:** 300 MHZ or higher
- **Memory:** 128 MB
How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Read the file named readme.pdf to view information about how to restore the program code to VisualAge for Java and the iSeries server.
Related publications

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

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 428.

- WebSphere 4.0 Installation and Configuration on the IBM @server iSeries Server, SG24-6815
- IBM WebSphere V4.0 Advanced Edition Handbook, SG24-6176

Other resources

These publications are also relevant as further information sources:

- Work Management Guide, SC41-5306
- MQSeries: Using Java, SC34-5456
Referenced Web sites

These Web sites are also relevant as further information sources:

- Javasoft home page: http://www.javasoft.com
- Servlet specifications: http://java.sun.com/products/servlet/2.2/
- Linar J-Intrega homepage: http://www.linar.com

How to get IBM Redbooks

Search for additional Redbooks or redpieces, view, download, or order hardcopy from the Redbooks Web Site

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web Site for information about all the CD-ROMs offered, updates and formats.
### Abbreviations and acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAFP</td>
<td>Advanced Function Printing</td>
</tr>
<tr>
<td>ASP</td>
<td>Active Server Pages</td>
</tr>
<tr>
<td>APA</td>
<td>All Points Addressable</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Window Toolkit</td>
</tr>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>CGI</td>
<td>Communications Gateway Interface</td>
</tr>
<tr>
<td>CPW</td>
<td>Commercial Processing Workload</td>
</tr>
<tr>
<td>DAX</td>
<td>Data Access Builder</td>
</tr>
<tr>
<td>DCM</td>
<td>Digital Certificate Manager</td>
</tr>
<tr>
<td>DDM</td>
<td>Distributed Data Management</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic link library</td>
</tr>
<tr>
<td>DPC</td>
<td>Distributed Program Call</td>
</tr>
<tr>
<td>EAB</td>
<td>Enterprise Access Builder</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBeans</td>
</tr>
<tr>
<td>FFST</td>
<td>First Failure Support Technology</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>GWAPI</td>
<td>Go Web Server API</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transmission Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transmission Protocol Secure</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>ICAPI</td>
<td>IBM Connection API</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet inter-ORB protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ITO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>JAR</td>
<td>Java archive</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java database connectivity</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Toolkit</td>
</tr>
<tr>
<td>JFC</td>
<td>Java Foundation Classes</td>
</tr>
<tr>
<td>JIT</td>
<td>Just in Time Compiler</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>MI</td>
<td>Machine interface</td>
</tr>
<tr>
<td>MIME</td>
<td>Multi-purpose Internet Mail Extensions</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>NSAPI</td>
<td>Netscape API</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open database connectivity</td>
</tr>
<tr>
<td>OOA</td>
<td>Object oriented analysis</td>
</tr>
<tr>
<td>OOD</td>
<td>Object oriented design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object oriented programming</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>PEX</td>
<td>Performance Explorer</td>
</tr>
<tr>
<td>PCML</td>
<td>Program Call Markup Language</td>
</tr>
<tr>
<td>PDML</td>
<td>Panel Definition Markup Language</td>
</tr>
<tr>
<td>PTDV</td>
<td>Performance Trace Data Visualizer</td>
</tr>
<tr>
<td>PTF</td>
<td>Program temporary fix</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote method invocation</td>
</tr>
<tr>
<td>RAWT</td>
<td>Remote Abstract Windowing Toolkit</td>
</tr>
<tr>
<td>SCS</td>
<td>SNA Character Set</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>SLIC</strong></td>
<td>System Licensed Internal Code</td>
</tr>
<tr>
<td><strong>S-MIME</strong></td>
<td>Secure Multi-purpose Internet Mail Extensions</td>
</tr>
<tr>
<td><strong>SNA</strong></td>
<td>System Network Architecture</td>
</tr>
<tr>
<td><strong>SSL</strong></td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td><strong>TCP</strong></td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td><strong>TIMI</strong></td>
<td>Technology Independent Machine Interface</td>
</tr>
<tr>
<td><strong>UML</strong></td>
<td>Unified Methodology Language</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td><strong>VCE</strong></td>
<td>Visual Composition Editor</td>
</tr>
<tr>
<td><strong>WAS</strong></td>
<td>WebSphere Application Server</td>
</tr>
<tr>
<td><strong>WWW</strong></td>
<td>World Wide Web</td>
</tr>
<tr>
<td><strong>XML</strong></td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Index

Symbols
*current 365
*PGM 354

A
AAT (Application Assembly Tool) 15, 28, 41, 52, 59, 74–77, 88
ABC Company 232
customer transaction 232
database 232
access package 31
ace.jar 128
ActiveX components 228
activity diagram 52
addCookie method 61
address object 246
adminclient 102
Administrative Console 102
Apache Tomcat 147, 149, 176
applets 7–8
application assembler 17–18, 225
Application Assembly Tool (AAT) 15, 28, 41, 52, 59, 74–77, 88
application client 7
application component provider 18
application deployer 226
Application Developer 145–146, 175, 177, 357
automatic build 156
creating an XML file 379
customizing 155
Debug perspective 171
default perspective 156
EJB Test client 283
exporting an EAR file 194
exporting Java code 173
exporting to an EAR file 213
Filter button 180
getting started 149
Help perspective 151
HTML files 190
import code 153
importing a schema 266
initialization parameters 184
J2EE specification 182
JRE 331
migration 195
navigation 150
overview 146
Page Designer 190
Run button 171
running Java code 169
Scrapbook 204
VisualAge for Java behavior 164
XML Source view 384
application.xml 16, 183
application-client.xml 76
architecture of sample servlet application 30
architecture, 2-tier versus 3-tier 251
AS400 object 357, 360, 362–363, 365
AS400DataType 361
AS400JDBCConnectionPoolDataSource 12, 188, 281
AS400JDBCDriver 50, 185
AS400Message 361
asynchronous communication 13
authentication 10
authorization 10
automatic build 156

B
basic authentication 10
BEA WebLogic Application Server 222
bean component content 222
bean component types 222
bean implementation (OrderPlacementBean) 255
bean managed 222
bean-level attributes 253
bean-managed persistence (BMP) 123, 251, 307
BigDecimal 290, 293
BIN 290
BlueStone Software 222
BMP (bean-managed persistence) 123, 307
bookmark 162–163
bootstrap port 121
bottom-up mapping 148, 265–266
building Java applications 340
using Enterprise JavaBeans 315
building servlets 340
business data 248
business processes 251
business tier 195
business transaction 221

C
CalJSP 100
CalJSP servlet 55
cannot register JDBC driver 185
CartItem 198
CartServlet 340, 354, 370, 372–373
CartServlet class 69
CartServlet servlet 341
CCBootstrapHost 139
CCBootstrapPort 139
certificate authentication 10
collection 367
CHAR 290
Checkout JSP 423
Checkout.jsp 198
child element 383
class files 28
class.forName 33
client module 76
client-side component 7
CMP (container-managed persistence) 16, 123, 265
CNInitialContextFactory 43, 47
Code Assist 147–148
collection 266
Column 244
COM (Component Object Model) 219, 228
com.ibm.mq package 367
com.ibm.mq.jar 366
command bean 396
command creation 398
Command interface 396
execute() method 396
isReadyToCallExecute() method 396
reset() method 396
Command package 128, 395–396
framework 395
framework benefits 398
CommandTarget interface 403
Common Object Request Broker Architecture (CORBA) 219, 228
communication 5
compatibility test suite (CTS) 1–2
component contract 220
Component Object Model (COM) 219, 228
component provider 17
components 4, 7
confirmOrder method 333
connect method 332, 335
connection pool 41
connection pooling mechanism 44
connectStateless method 333, 338
connectTOQM method 368–369
container 6
container managed 222
container provider 226
container-managed persistence (CMP) 16, 123, 265
container-managed transaction 12
content assist tool 167
Context object 42
context root 84, 180, 183
controller 8, 211
controller servlet 197, 203
cookies 60–63
CORBA (Common Object Request Broker Architecture) 219, 228
creating a command 397–398
CSTMR 241
CTS (compatibility test suite) 1
current library 362
cursor shape 160
custom target policy 406
Customer class 332
Customer JavaBean 198
Customer table 239, 248
Customer Table Layout (CSTMR) 241
customer transaction 232

data tier 195
Data Conversion classes 360
data definition language (DDL) 149, 296
data queue 233, 254, 256, 352, 361, 364, 371, 375, 387
interfacing to legacy applications 363
writeDataQueue method 364
datasource 26, 41–43, 47, 49–50, 137, 188, 208, 249, 277, 347
connection pooling mechanism 44
object 42
servlet example 41
version 41
DataSourceBean 249
datastore 254
DATE 290
DB2 XML Extender 149
DB2Driver 89
DB2StdXDataSource 12
DBA (database administrator) 149
dbConnection variable 32
DDL (data definition language) 149, 296
dbConnection 41
DECIMAL 290
default font 157
default perspective 156
default_app.webapp 40, 57
Deployed_MyEJBs.jar 114
Deployed_OrderEntryBeans.jar 135
deployer 17–18
deployment descriptor 15, 17, 118, 223, 249, 252–253, 257, 319, 364
example 16
development environment for WebSphere 22
digest authentication 10
distributed application model 3
distributed applications 395
distributed Java applications 395
distributed object architectures 228
Distributed Program Call 352, 361
Distributed Program Call class 354
distributed transaction 226
District 414
District table 239, 248
District Table Layout (Dstrct) 241
Document Object Model (DOM) 379, 384, 386, 388
document root 27
DOCUMENT_NODE 387
doDelete() method 24
doGet() method 24
DOM (Document Object Model) 379, 384, 386, 388
Index 433

dooPost() method 24
dooPut() method 24
DOUBLE 290
double 290
Dstrct 241
DTAQ 363
DTD 148
Editor 148
E
EAR 16–17, 76–77, 79, 147, 174, 194
EAR (Enterprise Archive) file 14
EAR file export 194, 213
EBCDIC 361
EJB 148
component model 218
migration to version 1.1 344
roles 224
application assembler 225
application deployer 226
container provider 226
Enterprise JavaBean provider 224
server architecture 221
specification 224
technology on the iSeries server 226
EJB 1.1 5
EJB container 219–220
EJB interoperability 228
EJB module 76
EJB server 219–221
EJB server provider 226
EJB specification 216
EJB technology
in the Java architecture 217
on the iSeries server 228
EJB Test Client 283, 320
EJB to RDB mapping 272, 274–275, 293
EJBCommandTarget 396
EJBCommandTarget class 399, 404
ejbCreate() method 251, 310, 344
ejbdeploy tool 279
EJBException 318, 345
ejb-jar.xml 16, 76, 252, 257, 275, 319
ejbLoad method 251, 311
ejbRemove method 251, 313
ejbStore method 251, 312
ELEMENT_NODE 387
e-mail 5
encodeURL 62
Enterprise Archive (EAR) file 14
Enterprise JavaBean 215, 315
application development scenario 315
architecture 216, 219
components 219
container 220
contract 220
provider 224
services 219
specification 216, 224
entity beans 219–220, 222
application development scenario 248
bean-managed persistence 222, 251
container-managed persistence 222
entity Enterprise JavaBeans 248
environment variable 249
execute() method 396
executeCommand() method 410
exporting class files to iSeries IFS 28

F
field 244
Field description classes 361
FIFO 363
filter warnings 180
findAllCustomers method 262
findAllItems method 262
findByPrimaryKey 251
findRangeOfItems method 262, 333, 336
flexLog() method 36
FLOAT 290
float 290
FLTDBL 290
foreign-key 288
form-based authentication 10

G
garbage collection 6
GenericServlet 37
getAllCustomers method 332, 335
getAllItems method 341
getAttribute 63
getAttributeNames 63
getCommandTarget() method 398
getCommandTargetName() method 398
getCustCmd 404–405
GetCustomerCmd command 412
GetCustomerCmdEJB command 398–399, 403
GetCustomerCmdEJB interface 399
GetCustomerCmdEJBImpl class 400–401
GetCustomerCmdImpl 412
GetCustomerEJBCmd interface 401
GetCustomerEJBImpl 404
getCustomerForOrder method 262
getInitialContext method 332, 335
getItems method 332, 336
getItemsForOrder method 262
getMainPropertiesFile() method 36
getMQ method 369
gerRequestDispatcher() method 56
gerServerRoot() method 36
gerServletConfig method 37
getter 276
getter methods 201–202
getValue 63
getValueNames 63
getWriter() method 26
GIOPVersionException 331
H
Hashtable 47
Hashtable object 43
hasOutputProperties() method 398
HelloEJBServlet.jar 108
HelloTestServlet 325
HelloWorld EJB 316
HelloWorldClient.jar 108
Help perspective 151
Help Search tab 151
hidden fields 60
History table 239
home interface 223, 333
home interface (OrderPlacementHome) 254
hostname 367
HTTP server 22
HTTP/HTTPS 7
HttpServletRequest 25–26
HttpServletResponse 25–26
HttpSession Interface 62
HttpSession object 63
I
IBM Developer Kit for Java 12
IBM Enterprise Toolkit for AS/400 26
IBM Framework for e-business 393–394
IBM Toolbox for Java 12, 128, 181, 189, 352, 360–361
IBM WebSphere Business Components 229
ibm-ejb-jar-bnd.xmi 275
ibm-ejb-jar-ext.xmi 275
ibm-web-bnd.xml 324
ibm-web-ext.xml 325
IDE 26
IDL (Interface Definition Language) 13, 228
IIOP (Internet inter-ORB protocol) 228
IllegalStateException 55
IMAP 14
implementing a state 260
import statement 334
incremental save 147
index 244
InitialContext object 326, 334
int 290
INTEGER 290
integrated development environment (IDE) 216
Interface Definition Language (IDL) 13, 228
interfacing to legacy applications 351
Internet Inter-ORB Protocol 13
Internet shopping application 315
interoperability 228
iSeries database 29
iSeries Developer Kit for Java 227–228
iSeries EJB application development scenario 245
iSeries server
  EJB technology 226, 228
  Java environment 227
  isolation level 12
  isReadyToCallExecute() method 396, 402
Item 243
Item class 332
Item EJB 250
Item table 29, 239, 248
Item Table Layout (ITEM) 243
ItemPool.html 190
ItemPoolServlet 49, 184, 188
Items 198
ItemsDb class 332, 341, 373
  connect method 335
  findRangeOfItems 336
  getAllCustomers method 335
  getAllItems method 341
  getInitialContext method 334
  getItems method 336
  submitOrder method 337
  using without a GUI 339
  verifyCustomer method 337
ItemServlet 183
ItemServlet class 36
ItemServlet on iSeries 41, 52, 59, 74
ItemServlet.doGet method 39
ItemServlet.doPost() method 37
ItemServlet.init() method 37
ItemServlet.outputItemInformation() method 39
ItemSessionServlet 340
ItemSessionServlet class 64
ItemSessionServlet servlet 341
ItsoEjb module 277
J
J2C 6
J2EE 3–4, 23, 28, 75, 182
  communication 13
  component object model 7
  components 4, 7
  container 6
  containers 6–7
  features 147
  package levels in WebSphere 5
  packaging 14
  platform technology 4
  roles 17
  services 4, 9
J2EE (Java 2 Enterprise Edition) 1–2
J2EE 1.2 3, 5, 147
J2EE 1.3 3, 6
J2EE communication 5
J2EE perspective 153, 178
j2ee.jar 330
J2ME (Java 2 Micro Edition) 2
J2SE (Java 2 Standard Edition) 2
JAF (JavaBean Activation Framework) 12, 14
JAF 1.0 5
JAR (Java Archive) file 14
Java
  applications 315
  architecture with EJB technology 217
  code export 173
  features 147
running code 169
Java 2 Enterprise Edition (J2EE) 2
Java 2 Micro Edition (J2ME) 2
Java 2 Platform, Enterprise Edition (J2EE) 1
Java 2 Standard Edition (J2SE) 2
Java applications 315
Java Archive (JAR) file 14
Java Build Path 182
Java Database Connectivity (JDBC) 10
Java for the iSeries server 227
Java Messaging Service (JMS) 13
Java Naming Directory Interface (JNDI) 9, 334
Java Native Interface (JNI) 352
Java perspective 152, 166–167, 171
Java Transaction API (JTA) 11
Java Transaction Service (JTS) 11
java.io 25
java.rmi package 334
JavaBean Activation Framework (JAF) 12, 14
JavaBean Proxy 149
JavaBeans 215
JavaMail 5, 12–14
JavaMail 1.1 5
JavaScript 192–193
JavaServer Pages (JSP) 21
  separating presentation logic 8
JavaServer Web Development Kit (JSWDK) 22
JavaSoft Web site 24
javax.ejb package 334
javax.naming package 334
javax.servlet 24–25
javax.servlet.http 24–25
javax.sql 42
JCA 148
JDBC 9, 41, 208, 248, 277, 290, 298, 395
  accessing an iSeries database 29
  JDBC CatalogSupport 31
  JDBC ItemCatalog 32
JDBC (Java Database Connectivity) 10
JDBC 2.0 5, 41
JDBC 2.0 Core APIs 10
JDBC 2.0 Extension APIs 10
JDBC driver 184
JDBC level 2.0 5
JDBC CatalogSupport 203
JDBC CatalogSupport class 31, 44
JDBC CatalogSupport.getRows() method 31
JDBC ItemCatalog 32
JDBC ItemCatalog.connectToDB() method 32
JDBC ItemCatalog.getAllV() method 34
JDBC ItemCatalog.getRecord() method 34
JDBC PoolCatalog 199, 209
JDBC PoolCatalog class 46
JDBC Pool Customer JavaBean 198
JDI based debugger 147
JDK
  performance improvements 6
JDK 1.3 147
J-Integra 228
JMS 5, 13
asynchronous 13
loosely coupled 13
peer to peer 13
point-to-point messaging 13
publish-subscribe messaging 13
reliable 13
JMS (Java Messaging Service) 13
JMS 1.0 5
JMS Transaction (JMS/XA) 6, 14
JMS/XA (JMS Transaction) 6, 14
JNDI 4, 43, 110, 115, 125, 127, 132, 148, 203, 223, 277,
  319, 326, 334, 344
  Global 9
  Local 9
JNDI (Java Naming Directory Interface) 9, 47, 334
JNDI 1.2 5
JNDI Explorer 283, 320
JNDI namespace 327
JNI (Java Native Interface) 352
JRE 6, 147, 331
jsessionid 62
JSP
  design 54
  life cycle 52
  specification 52
  support in WebSphere Version 3.5 52
JSP 0.91 52
JSP 1.1 5, 57
JSWDK (JavaServer Web Development Kit) 22
jt400.jar 128, 138, 182, 186, 269, 348
jt400ntv.jar 138, 348
JTA (Java Transaction API) 11
JTA 1.0 5
JTS (Java Transaction Service) 11
JUnit 153, 163, 167, 169, 173
junit.samples package 167
junit.samples.AllTests 172
junit37src.jar 155

K
key descriptor 272
KEYED 363

L
launchclient 121, 139
launcher 169
legacy applications 4, 351–352
  using data queues to interface 363
libraries resource property 189, 282
library 244
life cycle of a JSP 52
life-cycle methods 251, 255
LIFO 363
Linar J-Integra 228
Links View 148
localhost 365
LocalTarget 404
logical file 244
loosely coupled communication 13
M
Map.mapxm 275
mapping 265
  bottom-up 266
  meet-in-the-middle 296
  top-down 288
Master.css 325
meet-in-the-middle mapping 148, 265, 296
metadata 147, 149, 270, 275–276, 325
META-INF folder 319
method-level attributes 253
Microsoft 228
middleware 3–4, 11, 351, 366
MIME 14, 37
Model-View-Controller (MVC) architecture 196, 395,
  411, 414, 416
MQCon class 367, 376
MQException 368
QMMessage instance 368
MQSeries 351, 366, 370, 374, 379, 385, 387, 390–391
  Channel 367
  Hostname 367
  Queue 367
  Queue Manager 391
multithreaded 340
multi-tier application environment 3
MVC (Model-View-Controller) architecture 196, 414, 416
MyEJBs.jar 108
MyHelloWorldClient.jar 328
MyHelloWorldClient.jar 115
MyHelloWorldServlet 28, 79
MyOrderEntryClient.jar 135

N
narrow() method 327, 335
National Language Support 54
NativeDS 49, 99, 347, 358
Navigator view 165
Netscape 61
nonejb.jar 124, 127–128
Novera 222
nservlets package 31
null capable fields 305
NUMERIC 290

O
object diagram 246
OMG COS Naming Service 228
Open perspective button 150
Operations Navigator 296
ORDENTR 233–234, 352–353, 361
Order class 332
order entry clerk 247
order information format 360
Order Line table 239
Order Line Table Layout (ORDLIN) 242
Order table 239, 248
Order Table Layout (ORDERS) 242
order.XML file 379
OrderDetail 203
OrderDetail class 332
OrderDetail JavaBean 198
OrderEntry application 231
  application flow 233
  architecture with objects 246
  customer transaction 232
  customer transaction flow 234
  database 232
  database layout 240
  database table structure 239
  starting the application 234
OrderEntryApp enterprise application 123, 136
OrderEntryBeans.jar 124
OrderEntryClerk 124–125, 132–133, 247, 257, 352, 416
OrderEntryClerk EJB 251–252, 257, 333, 335, 337
  business methods 261
  ejbCreate method 260
  implementing state 260
OrderEntryClerk session bean 338
OrderEntryClerkBean 257
OrderEntryClerkBean class 259
OrderEntryClerkHome 257
OrderEntryClient.jar 124
OrderEntryCommandPkg package 400
OrderEntryControllerServlet 198, 206, 209, 211, 411,
  415–416, 419
OrderEntryServlet,war 135
OrderEntryWar.jar 124
OrderPlacement 124, 253, 412, 414
OrderPlacement EJB 251–252
  deployment descriptor 253, 258
    home interface 254
    implementation 259
    placeOrder method 256
    remote interface 255
OrderPlacement object 248
OrderPlacementBean 253
OrderPlacementBean (bean implementation) 255
OrderPlacementCmdImpl 412
OrderPlacementHome 253
OrderPlacementHome (home interface) 254
OrderProcess 206
OrderProcess JavaBean 198
ORDERS 242
ORDERS table 250
ORDLIN 242
ORDLIN table 250
Outline view 164
OutputCart 212
OutputCart.jsp 198
outputHeader() method 36
outputOrderForm method 371

P
PACKED 290
Page Designer 190–191, 194
PageCompile 53
parseXmlString method 385
parsing 148
UpdateCustomerCmdImpl  412, 421
URL encoding  61
URL mapping  183
URL rewriting  60–63
UserTransaction objects  9

V
VectorTest.java  167
verifyCustomer method  262, 333, 337
versioning  147
View  244
VisualAge for Java  21, 26–27, 75, 107, 124, 146, 164, 175, 195, 276
  changing the document root  27
VisualAge for Java Enterprise Edition 3.0  22
VisualAge for Java Enterprise Edition 3.5  22, 26
  testing the servlet  28
VisualAge for Java Scrapbook  339

W
WAR  16–17, 91, 148, 174, 177, 183
WAR (Web Application Archive)  14
WAR files  14
Warehouse table  239
Web Application Archive (WAR)  14
Web container  10
Web features  148
Web module  76
Web perspective  148
Web Server Plug-in  87
Web Services  146, 149
web.xml  16, 76, 207, 324
Web.xml editor  183
WebLogic  222
WebSphere 3.5  176, 396
WebSphere 4.0 JDK level  6
WebSphere Administrative Domain  188
WebSphere application development environment  22
WebSphere Application Server  3
  servlet support  22
WebSphere Application Server Advanced Edition 3.5
  222
  connection pools  248
  JSP support  21
  servlet support  21
WebSphere Application Server Version 3.5 JSP support
  52
WebSphere Business Components  229
WebSphere connection pool  248
WebSphere Enterprise Edition 4.0  228
WebSphere package levels for J2EE  5
WebSphere Studio  23, 175
WebSphere Studio Application Developer  107, 145–146
WebSphere Test Environment (WTE)  26–27, 279, 287
  workspace  177
writeDataQueue method  364
writeMQ method  374
writeMQXml method  389–390
WsnInitialContextFactory  43

wsws.jar  75, 79
Wws_App  178, 183
wswsitem.jar  75, 79
wswjsp.jar  75, 79
wswspool.jar  75, 79
wswssession.jar  75
WTE (WebSphere Test Environment)  26, 279, 287
WYSIWYG  148, 195

X
X/Open  227
XML  147, 149
xmi  275
xmi file  271
XML child element  383
XML document  385
XML editor  381
XML element  381
XML features  148
XML file creation  379
XML Schema Editor  148
XML Source Editor  148
XML tree  386
XmlDoc class  384, 386, 389
XSL  148
XSL Trace Editor  149

Z
Zip file  154
ZONED  290
WebSphere J2EE Application Development for the IBM iSeries Server

Build and deploy J2EE compliant applications for WebSphere 4.0

Use Application Developer to build iSeries servlets, JSPs, and EJBs

Learn how to interface to legacy applications

WebSphere Application Server 4.0 delivers the Java 2 Enterprise Edition (J2EE) implementation. It is the IBM strategic Web application server and a key IBM iSeries product for enabling e-business applications. The iSeries server and WebSphere Application Server are a perfect match for hosting e-business applications.

You can build J2EE applications using WebSphere Studio Application Developer — a new IBM application development environment. This is the follow-on product for VisualAge for Java and WebSphere Studio. It combines the best of these products into one integrated development environment.

This IBM Redbook shows you how to build and deploy iSeries J2EE applications and how to use them to access iSeries resources. It also shows you how to use your iSeries server as a Java server. It is written for anyone who wants to use Java servlets, JavaServer Pages, and Enterprise JavaBeans on the iSeries server.

This redbook provides many practical programming examples with detailed explanations of how they work. The examples were developed using VisualAge for Java Enterprise Edition 4.0 and WebSphere Studio Application Developer 4.02. They were tested using WebSphere Application Server 4.0.2 Advanced Edition and Advanced Edition Single Server. To effectively use this book, you should be familiar with the Java programming language and object-oriented application development.

For more information: ibm.com/redbooks

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

SG24-6559-00 ISBN 0738425117