## Contents

**Figures** .................................................................................................................. vii

**Tables** ....................................................................................................................... ix

**About This Book** ...................................................................................................... xi
  Prerequisite Knowledge ............................................................................................... xi
  How to Read Syntax Diagrams ....................................................................................... xi
  Example Syntax Diagram .............................................................................................. xiii
  How to Send Your Comments ....................................................................................... xiii

**Summary of Changes** ............................................................................................... xv
  Changes to the Current Edition of This Book for IMS Version 8 ........................................ xv
  Changes to This Book for IMS Version 8 ........................................................................ xv
  Library Changes for IMS Version 8 ................................................................................ xvi
  New, Revised, and Eliminated Titles .............................................................................. xvi
  Organizational Changes ................................................................................................ xvi
  Deleted Information ...................................................................................................... xvi
  Accessibility Enhancements .......................................................................................... xvii
  User Assistive Technologies .......................................................................................... xvii
  Accessible Information ................................................................................................ xvi
  Keyboard Navigation of the User Interface ..................................................................... xvii

**Chapter 1. Getting Started with IMS Java** ................................................................ 1
  IMS Java System Requirements ..................................................................................... 1
  Installing IMS Java ........................................................................................................ 1
  Administering IMS Java ................................................................................................ 3
  IMS Java Class Library Summary .................................................................................. 4
  General Restrictions for Applications ........................................................................... 5
  Where to Find More Information about IMS Java ....................................................... 5

**Chapter 2. JMP and JBP Applications** .................................................................... 7
  Running the IMS Java IVP in a JMP Region ..................................................................... 8
  Running the IMS Java IVP in a JBP Region ................................................................. 11
  Running the IMS Java Sample Application from a JMP Region .................................... 13
  Configuring JMP and JBP Regions for DB2 UDB for z/OS Database Access ............... 15
  Developing JMP Applications ....................................................................................... 16
    Subclassing the IMSFieldMessage Class to Define Input Messages ......................... 16
    Subclassing the IMSFieldMessage Class to Define Output Messages ..................... 17
    Implementing the main Method ............................................................................... 18
  JMP Programming Models ........................................................................................... 19
    JMP Application Without Rollback ........................................................................... 19
    JMP Application that Uses Rollback .......................................................................... 20
    JMP Application that Accesses IMS or DB2 UDB for z/OS Data ............................... 20
  Additional Message Handling Considerations for JMP Applications ......................... 21
    Conversational Transactions ..................................................................................... 21
    Handling Multi-Segment Messages .......................................................................... 23
    Coding and Accessing Messages with Repeating Structures ................................. 24
    Flexible Reading of Multiple Input Messages .......................................................... 25
  Developing JBP Applications ....................................................................................... 27
  JBP Programming Models ........................................................................................... 27
    JBP Application without Rollback ............................................................................. 27
    JBP Application using Rollback ................................................................................ 28
    JBP Application that Accesses DB2 UDB for z/OS or IMS Data ............................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDFLD Statement</td>
<td>109</td>
</tr>
<tr>
<td>INCLUDE Statement</td>
<td>109</td>
</tr>
<tr>
<td>Comment Statement</td>
<td>110</td>
</tr>
<tr>
<td>Running the DLIModel Utility</td>
<td>110</td>
</tr>
<tr>
<td>Running the DLIModel Utility as a JCL Job</td>
<td>110</td>
</tr>
<tr>
<td>PROC Statement Parameters</td>
<td>111</td>
</tr>
<tr>
<td>Step 1 EXEC Statement Parameters</td>
<td>111</td>
</tr>
<tr>
<td>Step 1 DD Statements</td>
<td>112</td>
</tr>
<tr>
<td>Step 2 EXEC Statement Parameters</td>
<td>112</td>
</tr>
<tr>
<td>Step 2 DD Statements</td>
<td>112</td>
</tr>
<tr>
<td>Running the DLIModel Utility From UNIX Systems Services</td>
<td>113</td>
</tr>
<tr>
<td>Examples of Using the DLIModel Utility</td>
<td>113</td>
</tr>
<tr>
<td>JMP IVP Metadata Sample</td>
<td>114</td>
</tr>
<tr>
<td>JBP IVP Metadata Sample</td>
<td>115</td>
</tr>
<tr>
<td>Sample Metadata with COBOL Copybook XML</td>
<td>116</td>
</tr>
<tr>
<td>Chapter 8. Problem Determination</td>
<td>123</td>
</tr>
<tr>
<td>Exceptions</td>
<td>123</td>
</tr>
<tr>
<td>How Exceptions Map to DL/I Status Codes</td>
<td>123</td>
</tr>
<tr>
<td>SQLException Objects</td>
<td>124</td>
</tr>
<tr>
<td>XML Tracing for IMS Java</td>
<td>124</td>
</tr>
<tr>
<td>WebSphere Application Server Security Requirements for XML Tracing</td>
<td>125</td>
</tr>
<tr>
<td>Enabling XML Tracing</td>
<td>125</td>
</tr>
<tr>
<td>Tracing the IMS Java Library Methods</td>
<td>126</td>
</tr>
<tr>
<td>Tracing Your Application</td>
<td>126</td>
</tr>
<tr>
<td>Debugging an Unresettable JVM in a JMP or JBP Region</td>
<td>127</td>
</tr>
<tr>
<td>Appendix A. Preparing to Run the Dealership Samples</td>
<td>129</td>
</tr>
<tr>
<td>Modifying IMS Stage 1 Input Statements</td>
<td>129</td>
</tr>
<tr>
<td>Loading the Dealership Sample Databases</td>
<td>129</td>
</tr>
<tr>
<td>Appendix B. SQL Keywords</td>
<td>133</td>
</tr>
<tr>
<td>Appendix C. IMS Java Hierarchical Database Interface</td>
<td>135</td>
</tr>
<tr>
<td>Application Programming Using the DLIConnection Object</td>
<td>135</td>
</tr>
<tr>
<td>Creating a DLIConnection Object</td>
<td>135</td>
</tr>
<tr>
<td>Creating an SSAList Object</td>
<td>136</td>
</tr>
<tr>
<td>Accessing IMS Data Using SSAs</td>
<td>136</td>
</tr>
<tr>
<td>Notices</td>
<td>137</td>
</tr>
<tr>
<td>Trademarks</td>
<td>139</td>
</tr>
<tr>
<td>Bibliography</td>
<td>141</td>
</tr>
<tr>
<td>IMS Version 8 Library</td>
<td>141</td>
</tr>
<tr>
<td>Publication Collections</td>
<td>142</td>
</tr>
<tr>
<td>Accessibility Titles Cited in this Book</td>
<td>142</td>
</tr>
<tr>
<td>Index</td>
<td>143</td>
</tr>
</tbody>
</table>
Figures

1. JMP or JBP Application That is Using IMS Java ........................................... 8
2. IVP Input Screen for IMS Java JMP ............................................................. 10
3. IVP Output Screen for IMS Java JMP .......................................................... 11
4. Subclass IMSFieldMessage: Input Message Sample Code ............................ 17
5. Subclass IMSFieldMessage: Output Message Sample Code .......................... 18
6. main Method Sample Code ......................................................................... 19
7. Defining a SPA Message .............................................................................. 22
8. Reading a SPA Message ............................................................................. 22
9. Writing a SPA Message ............................................................................. 22
10. Sample Output Message with Repeating Structures .................................... 24
11. Defining the Primary Input Message ............................................................ 25
12. Defining Separate Input Messages for Each Request ................................... 26
13. Message-Reading Logic ........................................................................... 27
14. WebSphere Application Server for z/OS EJB Using IMS Java .................... 35
15. DB2 UDB for z/OS Stored Procedure Using IMS Java ............................... 59
16. Sample JAVAENV Data Set ....................................................................... 60
17. CICS Application Using IMS Java ............................................................. 67
18. Sample Dealership Database ..................................................................... 74
19. Relational Representation of the Dealership Database ............................... 75
20. Segment Occurrences in the Dealership Database ....................................... 76
21. Relational Representation of Segment Occurrences in the Dealership Database 76
22. Example of SELECT Statement Query Results ......................................... 78
23. Sample Relational Database Query ............................................................. 79
24. Sample Hierarchical Database Query ........................................................ 79
25. Sample INSERT Statement ...................................................................... 80
26. Sample DELETE Statement ...................................................................... 81
27. Sample UPDATE Statement ...................................................................... 81
28. PCB-Qualified SQL Query Example ........................................................... 82
29. Establishing a Connection to the Dealership Database ............................... 85
30. Sample PSB for the Sample Dealership Application .................................... 91
31. DBD for the Sample Dealership Database ................................................... 91
32. Sample DLIModel IMS Java Report for the Dealership Sample Database .... 92
33. Example JDBC Application ...................................................................... 94
34. DLIModel Utility Inputs and Outputs .......................................................... 98
35. Sample DLIModel Utility Procedure .......................................................... 111
36. DBD for the IVP Database ....................................................................... 114
37. PSB for the JMP IVP ............................................................................... 114
38. Control Data Set for JMP IVP .................................................................. 114
39. DLIModel IMS Java Report for JMP IVP ................................................... 115
40. DBD for the IVP Database ....................................................................... 115
41. PSB for the JBP IVP ............................................................................... 116
42. Control Data Set for JBP IVP .................................................................. 116
43. DLIModel IMS Java Report for JBP IVP ................................................... 116
44. PSB for COBOL Copybook XMI Example ............................................... 117
45. Physical DBD for COBOL Copybook XMI Example ................................. 117
46. UNIX System Services Command for Control Statements Example .......... 117
47. Top-Level Control Data Set for COBOL Copybook XMI Example ............. 118
48. Second-Level Control Data Set for COBOL Copybook XMI Example ........ 119
49. Copybook for COBOL Copybook XMI Example ....................................... 120
50. Equivalent Control Statements for COBOL Copybook XMI Example .......... 120
51. DLIModel Java Report for COBOL Copybook XMI Example .................... 121
52. IMSException Class Example ................................................................. 124
53. Setting a Trace within a Static Method ...................................................... 126

© Copyright IBM Corp. 2000, 2005
### Tables

1. How to Read Syntax Diagrams .................................................. xi
2. Supported SQL Aggregate Functions and Their Supported Data Types ............... 84
3. Supported JDBC Data Types ................................................... 87
4. ResultSet.getxxx Methods to Retrieve JDBC Types ........................... 87
5. Mapping from COBOL Formats to DLITypeInfo Constants and Java Data Types .... 88
6. DLITypeInfo Constants and Java Data Types Based on the PICTURE Clause ........ 89
7. Copybook Formats Mapped to DLITypeInfo Constants ................................ 89
About This Book

This information is available in PDF and BookManager® formats, and also as part of the DB2 Information Management Software Information Center for z/OS Solutions. To view the information within the DB2 Information Management Software Information Center for z/OS Solutions, go to http://publib.boulder.ibm.com/infocenter/dzichelp. To get the most current versions of the PDF and BookManager formats, go to the IMS™ Library page at http://www.ibm.com/software/data/ims/library.html.

This book provides application development and deployment information for IMS Java™, a function of IMS that allows you to write Java application programs that access IMS databases from multiple systems.

Information about IMS Java is also available from the IMS Web site. Go to www.ibm.com/ims and link to the IMS Java page.

Prerequisite Knowledge

To configure your system for IMS Java, you must understand system administration for your system (IMS, WebSphere® Application Server for z/OS®, CICS®, or DB2® UDB for z/OS). For IMS system administration, you should know the concepts in IMS Version 8: Administration Guide: System.

To create IMS Java metadata classes, which is a required step in writing IMS Java applications, you must understand IMS databases. IMS database concepts are described in IMS Version 8: Administration Guide: Database Manager.

To write Java applications, you must thoroughly understand the Java language and JDBC. This book assumes that you know Java and JDBC. It does not explain any Java or JDBC concepts.

How to Read Syntax Diagrams

This book contains syntax diagrams.

Each syntax diagram begins with a double right arrow and ends with a right and left arrow pair. Lines that begin with a single right arrow are continuation lines. You read a syntax diagram from left to right and from top to bottom, following the direction of the arrows.

Table 1 describes the conventions that are used in syntax diagrams in this information:

<table>
<thead>
<tr>
<th>Table 1. How to Read Syntax Diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Convention</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Convention</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image5" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Name:**

Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.

**Punctuation marks and numbers**
Enter punctuation marks (slashes, commas, periods, parentheses, quotation marks, equal signs) and numbers exactly as shown.

**Uppercase values**
Keywords, their allowable synonyms, and reserved parameters appear in uppercase letters for OS/390®. Enter these values exactly as shown.

**Lowercase values**
Keywords, their allowable synonyms, and reserved parameters appear in lowercase letters for UNIX®. Enter these values exactly as shown.

**Lowercase values in italic (for example, name)**
Supply your own text or value in place of the name variable.

A b symbol indicates one blank position.
Other syntax conventions include the following:
- When you enter commands, separate parameters and keywords by at least one blank if there is no intervening punctuation.
- Footnotes are shown by a number in parentheses, for example, (1).
- Parameters with number values end with the symbol #.
- Parameters that are names end with ‘name’.
- Parameters that can be generic end with the symbol ‘*’.

**Example Syntax Diagram**

Here is an example syntax diagram that describes the `hello` command.

```
Hello
     ┌───────────────┐
     │ Name          │
     │               │
     │               │
     └───────────────┘
```

**Name:**

```
, (1)
name
```

**Greeting:**

```
, — your_greeting
```

**Notes:**

1. You can code up to three names.
2. Compose and add your own greeting (for example, how are you?).

According to the syntax diagram, these commands are all valid versions of the `hello` command:

```
hello
hello name
hello name, name
hello name, name, name
hello, your_greeting
hello name, your_greeting
hello name, name, your_greeting
hello name, name, name, your_greeting
```

The space before the `name` value is significant. If you do not code `name`, you must still code the comma before `your_greeting`.

**How to Send Your Comments**

Your feedback is important in helping us provide the most accurate and highest quality information. If you have any comments about this or any other IMS information, you can do one of the following:

- Go to the IMS home page at [www.ibm.com/ims](http://www.ibm.com/ims). There you will find an online feedback page where you can enter and submit comments.
• Send your comments by e-mail to imspubs@us.ibm.com. Be sure to include the title, the part number of the title, the version of IMS, and, if applicable, the specific location of the text you are commenting on (for example, a page number in the PDF or a heading in the Information Center).
Summary of Changes

Changes to the Current Edition of This Book for IMS Version 8

This edition, which is available in softcopy format only, includes technical and editorial changes. It also has major organizational changes.

This version contains the following new information:

- "Installing IMS Java" on page 1
- "Where to Find More Information about IMS Java" on page 5
- "Running the IMS Java Sample Application from a JMP Region" on page 13
- "Running the IMS Java Sample Application on DB2 UDB for z/OS" on page 63
- "Running the IMS Java Sample Application on CICS" on page 69
- "Developing Enterprise Applications that Access IMS DB" on page 54
- "XML Tracing for IMS Java" on page 124
- "Debugging an Unresettable JVM in a JMP or JBP Region" on page 127

The following information has been deleted:

- Manually creating IMS Java metadata classes. You should always use the DLIModel utility to generate metadata classes. See Chapter 7, “DLIModel Utility,” on page 97.
- IMSTrace facility. This facility has been deprecated. Use the XMLTrace facility instead. See “XML Tracing for IMS Java” on page 124.

This version also contains major organizational changes:

- The parts have been removed.
- The overview information for each environment has been moved to the chapter for each environment.
- The configuration and installation verification tasks for all environments have been split into their own sections.
- Application development information for JMP and JBP applications have been split into their own sections.
- Chapter 6, “JDBC Access to IMS Data,” on page 73 has been reorganized to improve retrievability of reference information.
- Sample JCL jobs have been added throughout the book to provide an alternative to directly using UNIX System Services.

Changes to This Book for IMS Version 8

This book has undergone major organizational changes in addition to technical changes.

This book contains new information about the following enhancements to IMS Java application support:

- JMP and JBP IMS dependent regions for a Java Virtual Machine (JVM) environment
- DLIModel utility for creating IMS Java metadata classes
- DB2 stored procedures, CICS, WebSphere Application Server for z/OS support
High Performance Java (HPJ) information has been removed from this book because HPJ is not supported by IMS Version 8.

Library Changes for IMS Version 8

Changes to the IMS Library for Version 8 include the addition of new titles, the elimination of one title, organizational changes, and accessibility enhancements. Changes are indicated by a vertical bar (|) to the left of the changed text.

New, Revised, and Eliminated Titles

The following list details major changes to the IMS Version 8 library:

- **IMS Version 8: Common Service Layer Guide and Reference**
  The library includes new information: **IMS Version 8: Common Service Layer Guide and Reference** (CSL). This information is available only in PDF and BookManager formats.
- The information formerly titled **IMS Version 7: Common Queue Server and Base Primitive Environment Guide and Reference** has been divided in the IMS Version 8 library:
  - **IMS Version 8: Base Primitive Environment Guide and Reference**
  - **IMS Version 8: Common Queue Server Guide and Reference**
- The information formerly titled **IMS Version 7: Installation Volume 1: Installation and Verification** is now titled **IMS Version 8: Installation Volume 1: Installation Verification**. All installation information is now in the IMS Version 8 Program Directory.
- **IMS Version 8: Sample Operating Procedures**
  This information is no longer produced for the IMS library from IMS Version 8 and after.
- The information formerly titled **IMS Version 8: IMS Java User’s Guide** is now titled **IMS Version 8: IMS Java Guide and Reference**

Organizational Changes

Organizational changes to the IMS Version 8 library include changes to:

- **IMS Version 8: Application Programming: Database Manager**
- **IMS Version 8: Application Programming: EXEC DLI Commands for CICS and IMS**
- **IMS Version 8: Application Programming: Transaction Manager**
- **IMS Version 8: Messages and Codes, Volume 1**
- **IMS Version 8: Utilities Reference: Database and Transaction Manager**


The section titled “DL/I StatusCodes” will now only appear in **IMS Version 8: Messages and Codes, Volume 1**
The section titled “MFS Language Utility” has been renamed to “MFS Language Utility Control Statements” and has been moved from IMS Version 8: Application Programming: Transaction Manager to IMS Version 8: Utilities Reference: Database and Transaction Manager.

Deleted Information

OS/390 does not support the Virtual Fetch function any longer. Consequently, all information associated with Virtual Fetch has been deleted from the following IMS Version 8 information:

- IMS Version 8: Administration Guide: System
- IMS Version 8: Failure Analysis Structure Tables (FAST) for Dump Analysis
- IMS Version 8: Installation Volume 2: System Definition and Tailoring
- IMS Version 8: Messages and Codes, Volume 1
- IMS Version 8: Messages and Codes, Volume 2

Accessibility Enhancements

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS products, including IMS, enable users to:

- Use assistive technologies such as screen-readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

User Assistive Technologies

Assistive technology products, such as screen readers, function with the user interfaces found in IMS. Consult the assistive technology documentation for specific information when using it to access these interfaces.

Accessible Information

Online information for IMS Version 8 is available in BookManager format, which is an accessible format. All BookManager functions can be accessed by using a keyboard or keyboard shortcut keys. BookManager also allows you to use screen readers and other assistive technologies. The BookManager READ/MVS product is included with the OS/390 base product, and the BookManager Library Reader™ (for workstations) is available on the IMS Licensed Product Kit (CD), which is available for downloading from IBM® at www.ibm.com

Keyboard Navigation of the User Interface

Users can access IMS user interfaces using TSO/E or ISPF. Refer to the z/OS: TSO/E Primer, z/OS: TSO/E User’s Guide, z/OS: ISPF User’s Guide. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.
Chapter 1. Getting Started with IMS Java

IMS Java is a function of IMS that allows you to write Java application programs that access IMS databases from many different locations:

- IMS JMP (Java message processing) and JBP (Java batch processing) dependent regions
- IBM WebSphere Application Server for z/OS
- IBM CICS Transaction Server for z/OS
- IBM DB2 Universal Database™ for z/OS stored procedures

IMS Java implements the JDBC API, which is the standard Java interface for database access. JDBC uses SQL (structured query language) calls. The IMS Java implementation of JDBC supports a selected subset of the full facilities of the JDBC 2.1 API.

In addition to JDBC, IMS Java has another interface to the IMS databases called the **IMS Java hierarchical database interface**. This interface is similar to the standard IMS DL/I database call interface, and provides lower-level access to IMS database functions than the JDBC interface. However, JDBC is the recommended access interface to IMS databases and this book focuses on JDBC. For information about the IMS Java hierarchical database interface, see Appendix C, “IMS Java Hierarchical Database Interface,” on page 135.

The following topics provide additional information:

- “IMS Java System Requirements”
- “Installing IMS Java”
- “Administering IMS Java” on page 3
- “IMS Java Class Library Summary” on page 4
- “General Restrictions for Applications” on page 5
- “Where to Find More Information about IMS Java” on page 5

**IMS Java System Requirements**

To use IMS Java to write application programs that access IMS databases, the following software and OS/390 components are required:

- IMS Version 8 with the IMS Java FMID
- IBM SDK for z/OS Java 2 Technology Edition, Version 1.3.1 or later
- OS/390 Version 2 Release 10 or later
- UNIX System Services available at runtime
- Hierarchic File System (HFS) on OS/390. For information about preparing an HFS, see *z/OS: UNIX System Services File System Interface Reference*.

**Installing IMS Java**

IMS Java is delivered in a separate FMID. Before you can install the IMS Java FMID with SMP/E, you must prepare HFS, which is described in this topic.

**Prerequisite:** Install IMS Version 8 and run the standard IMS IVPs. For details about how to run the IMS IVPs, see *IMS Version 8: Installation Volume 1: Installation Verification*. 
To install IMS Java:

1. Allocate a data set for HFS:

```
//HFSALLOC JOB parameters
noticed: To run this job:
// 1) Add JOB statement parameters to meet your requirements. */
// 2) For DSNAME, change hfsdsn to the name of the new file */
// 3) For VOLUME, change volid to the volser ID of the DASD */
// that will contain the IMS Java HFS data set. */
noticed: ALLOCATE EXEC PGM=IDCAMS,DYNAMNBR=200
noticed: SYSPRINT DD SYSOUT=* noticed: SYSIN DD *
ALLOCATE -
  DSNAME('hfsdsn') -
  RECFM(U) -
  LRECL(0) -
  BLKSIZE(32760) -
  DSORG(PO) -
  VOLUME(volid) -
  DSNTYPE(HFS) -
  NEW CATALOG -
  SPACE(15,5) CYL -
  DIR(200) -
  UNIT(SYSALLDA)
```

2. Define the mount point directory to mount the HFS:

```
//HFSMOUNT JOB parameters
noticed: To run this job:
// 1) Add JOB statement parameters to meet your requirements. */
// 2) For FILESYSTEM, change hfsdsn to the name of the file system that you specified in the HFSALLOC job. */
// 3) For MOUNTPOINT, change /PathPrefix to the high-level directory name. The directory name must be preceded with a forward slash (/), for example, /apps or /ims/apps. */
// This string must match the PathPrefix string in the DFSJSMKD job. */
noticed: MOUNT EXEC PGM=IKJEFT01
noticed: SYSTSPRT DD SYSOUT=* noticed: SYSTSIN DD *
MOUNT FILESYSTEM('hfsdsn') /* MOUNT HFS */ +
MOUNTPOINT('/PathPrefix') TYPE(HFS) MODE(READWR)
```

3. Run the sample installation job DFSJSMKD. DFSJSMKD runs the DFSJKMDR REXX script, which creates the HFS paths for IMS Java.

4. Using SMP/E, install the IMS Java FMID.

5. Expand the samples.tar file. This file is installed in pathprefix/usr/lpp/ims/imsjava81/samples and contains the files for the IVP and sample applications. You can expand this file by using a JCL job or a UNIX System Services command.

- To expand samples.tar by using a JCL job, edit and run the following job:

```
//UNTARJOB JOB parameters
noticed: To run this job:
// 1) Add JOB statement parameters to meet your requirements. */
// 2) Change PathPrefix to the name of the file system that you specified in the HFSALLOC job. */
// 3) Change path to directory where your login ID has write permission */
```

//UNTAR EXEC PGM=BPXBATCH,
//PARM='sh cd PathPrefix/usr/lpp/ims/imsjava81; tar -xvf samples/samples.tar'
//SYSPRINT DD SYSOUT=* 
//STDOUT DD PATH='path/untar.out',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//STDERR DD PATH='path/untar.err',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU

To expand samples.tar by using a UNIX System Services command, issue the following command from the pathprefix/usr/lpp/ims/imsjava81 directory:

tar -xvf samples/samples.tar

The contents of samples.tar are placed in the directory pathprefix/usr/lpp/ims/imsjava81. If you want the contents of samples.tar expanded in another directory, issue the command from directory where you want it expanded, and in the command, use the full path to samples.tar.

Next: The next step depends on the environment that your application will run in:

- JMP region: "Running the IMS Java IVP in a JMP Region" on page 8
- JBP region: "Running the IMS Java IVP in a JBP Region" on page 11
- WebSphere Application Server for z/OS: "Configuring WebSphere Application Server V5 for z/OS for IMS Java" on page 36
- DB2 UDB for z/OS stored procedure: "Configuring DB2 UDB for z/OS for IMS Java" on page 59
- CICS: "Configuring CICS for IMS Java" on page 67

Administering IMS Java

This topic provides the high-level tasks to administer IMS Java: from installing the IMS Java function to deploying your Java application. This topic does not contain application programming information.

To administer IMS Java:

1. Install and configure the required OS/390 software for IMS Java. See "IMS Java System Requirements" on page 1 for a list of required software and OS/390 components that must be installed before you can use IMS Java.
2. Install IMS Java. See "Installing IMS Java" on page 1.
3. Continue configuration for your environment, if necessary:
   - WebSphere Application Server for z/OS: "Configuring WebSphere Application Server V5 for z/OS for IMS Java" on page 36
   - DB2 UDB for z/OS stored procedure: "Configuring DB2 UDB for z/OS for IMS Java" on page 59
   - CICS: "Configuring CICS for IMS Java" on page 67
4. Run the IVP for your environment:
   - JMP region: "Running the IMS Java IVP in a JMP Region" on page 8
   - JBP region: "Running the IMS Java IVP in a JBP Region" on page 11
   - WebSphere Application Server for z/OS: "Running the IMS Java IVP on WebSphere Application Server V5 for z/OS" on page 39
   - DB2 UDB for z/OS stored procedure: "Running the IMS Java IVP from DB2 UDB for z/OS" on page 61
   - CICS: "Running the IMS Java IVP on CICS" on page 68
5. Write the PSB, and generate the DBDs, PSB, and ACB for the application.
6. Using the DBDs and PSB as input, write control statements for the DLIModel utility. See “Control Statements for the DLIModel Utility” on page 102.

7. Run the DLIModel utility, which uses the DBDs, PSB, and other input to generate the Java metadata class that the application uses to access the databases. The DLIModel utility is a Java application, so you can run it from the UNIX System Services prompt, or you can run it using the z/OS-provided BPXBATCH utility. See “Running the DLIModel Utility” on page 110.

8. Compile the Java source file of the Java metadata class that is generated to create a Java class file.

9. Provide the Java metadata classes and the DLIModel IMS Java Report, which provides the information about the IMS database, to the Java application developer.

10. Update the IMS system definition with a new APPLCTN macro statement for the Java application.

11. Deploy your application:
   - JMP region: Using the IMS Java IVP and sample application as models, start a JMP region with your specific requirements. See IMS Version 8: Installation Volume 2: System Definition and Tailoring for all of the available options.
   - JBP region: Using the IMS Java IVP as a model, start a JBP region with your specific requirements. See IMS Version 8: Installation Volume 2: System Definition and Tailoring for all of the available options.
   - WebSphere Application Server for z/OS: “Running Your Applications on WebSphere Application Server for z/OS” on page 44
   - DB2 UDB for z/OS stored procedure: “Running Your Stored Procedure from DB2 UDB for z/OS” on page 64
   - CICS: “Running Your Applications on CICS” on page 70

---

**IMS Java Class Library Summary**

Your Java application uses the IMS Java class library, which includes the following packages:

- **com.ibm.ims.base**
  Provides classes for basic IMS Java functions and for problem determination.

- **com.ibm.connector2.ims.db**
  Provides classes for connecting to IMS databases from WebSphere Application Server for z/OS.

- **com.ibm.ims.application**
  Provides classes for processing IMS messages, and performs commits and rollbacks for JMP and JBP applications.

- **com.ibm.ims.db**
  Provides classes for the JDBC driver and for the IMS Java hierarchical database interface.

**Related Reading:** For more information about the IMS Java class library, see the IMS Java API Specification (Javadoc). Go to the IMS Web site at www.ibm.com/ims and link to the IMS Java page.
General Restrictions for Applications

The following restrictions apply to applications that use IMS Java:

- The z/OS JVM restricts the classpath length to 255 characters. Do not create classpaths longer than 255 characters. This restriction does not apply to classpaths in WebSphere Application Server for z/OS.
- IMS Java applications cannot run in an IMS batch environment.
- For IMS Version 8 and later, IMS Java does not support High Performance Java (HPJ).
- IMS does not support local transactions.

Where to Find More Information about IMS Java

The information in this book is only one of the resources available for IMS Java information.

The IMS Java Web site contains current information about IMS Java and links to the resources described in this section. The Web site also has links to presentation materials from recent conferences, downloads, and announcements about IMS Java enhancements. Go to the IMS Web site at www.ibm.com/ims and link to the IMS Java page.

The IMS Java API specification is available on the IMS Java Web site. The specification contains information about the packages described in “IMS Java Class Library Summary” on page 4.


The following Redbooks™ contain information about IMS Java and related technologies:

- **IMS Version 7 Java Update (SG24-6536)**: Contains IMS Version 7 level information about running applications from JMP regions, JBP regions, DB2 stored procedures, and CICS.
- **IMS e-business Connectors: A Guide to IMS Connectivity (SG24-6514)**: Contains a chapter on setting up open database access (ODBA).
- **ABCs of System Programming Volume 9 (SG24-6989)**: Describes UNIX System Services (z/OS UNIX) and how to install, tailor, configure, and use the z/OS Version 1 Release 4 version of z/OS UNIX.
More information about IMS Java
Chapter 2. JMP and JBP Applications

Two IMS dependent regions provide a Java Virtual Machine (JVM) environment for Java applications:

**Java message processing (JMP) regions**

JMP regions are similar to MPP regions, but JMP regions allow the scheduling only of Java message-processing applications. A JMP application is started when there is a message in the queue for the JMP application and IMS schedules the message to be processed. JMP applications are executed through transaction codes submitted by users at terminals and from other applications. Each transaction code represents a transaction that the JMP application processes. A single application can also be started from multiple transaction codes.

JMP applications are very flexible in how they process transactions and where they send the output. JMP applications send any output messages back to the message queues and process the next message with the same transaction code. The program continues to run until there are no more messages with the same transaction code. JMP applications share the following characteristics:

- They are small.
- They can produce output that is needed immediately.
- They can access IMS or DB2 data in a DB/DC environment and DB2 data in a DCCTL environment.

**Java batch processing (JBP) regions**

JBP regions run flexible programs that perform batch-type processing online and can access the IMS message queues for output (similar to non-message-driven BMP applications). JBP applications are started by submitting a job with JCL or from TSO. JBP applications are like BMP applications, except that they cannot read input messages from the IMS message queue. For example, there is no IN= parameter in the startup procedure. JBP applications can access IMS or DB2 UDB for z/OS data in a DB/DC or DBCTL environment and DB2 UDB for z/OS data in a DCCTL environment.

**Important:** JMP and JBP regions are not necessary if your application runs in WebSphere Application Server, DB2 UDB for z/OS, or CICS. JMP or JBP regions are needed only if your application is going to run in an IMS dependent region.

Figure 1 on page 8 shows a Java application that is running in a JMP or JBP region. JDBC or IMS Java hierarchical interface calls are passed to the IMS Java layer, which converts the calls to DL/I calls.
JMP and JBP regions can run applications written in Java, object-oriented COBOL, or a combination of the two. See "Enterprise COBOL Interoperability with JMP and JBP Applications" on page 29.

JMP and JBP applications can access DB2 UDB for z/OS databases in addition to IMS databases. See "Configuring JMP and JBP Regions for DB2 UDB for z/OS Database Access" on page 15.

This chapter uses the sample applications that are shipped with IMS Java to show how to write and deploy IMS Java applications that run in JMP and JBP regions.

The following topics provide additional information:

- "Running the IMS Java IVP in a JMP Region"
- "Running the IMS Java IVP in a JBP Region" on page 11
- "Running the IMS Java Sample Application from a JMP Region" on page 13
- "Configuring JMP and JBP Regions for DB2 UDB for z/OS Database Access" on page 15
- "Developing JMP Applications" on page 16
- "Developing JBP Applications" on page 27
- "Enterprise COBOL Interoperability with JMP and JBP Applications" on page 29
- "Accessing DB2 UDB for z/OS Databases from JMP or JBP Applications" on page 32
- "Program Switching in JMP and JBP Applications" on page 33

Running the IMS Java IVP in a JMP Region

To verify that IMS Java is properly installed and that the JMP region is properly configured, run the IMS Java IVP. Details about the PROCLIB members and procedure parameters are in IMS Version 8: Installation Volume 2: System Definition and Tailoring.

This topic refers to the user-defined IMS Java installation directory as pathprefix. For information on defining this directory, see "Installing IMS Java" on page 1.

Prerequisites:

- Ensure that the standard IMS IVPs have been run. These IVPs prepare the DBD for the IVP database, named DFSIVD2, and load the IVP database. They also prepare the IMS Java application PSB (named DFSIVP37), build ACBs, prepare the
Running the IMS Java IVP in a JMP Region

MFS format (named DFSIVF37), and prepare other IMS control blocks required by the IMS Java IVPs. For details about how to run the IMS IVPs, see [IMS Version 8: Installation Volume 1: Installation Verification]

• "Installing IMS Java" on page 1

To run the IMS Java IVP in a JMP region:

1. Ensure that the samples.tar file has been expanded. This topic assumes that its contents are in the directory pathprefix/usr/lpp/ims/imsjava81. For more information on expanding the samples.tar file, see "Installing IMS Java" on page 1.

2. Edit the sample JVM member DFSJVMMS, which is in IMS.SDFSISR:
   a. For -Dibm.jvm.trusted.middleware.class.path=, change “ImsjavaPath” to pathprefix/usr/lpp/ims/imsjava81
   b. For -Dibm.jvm.shareable.application.class.path=, change “SamplePath” to pathprefix/usr/lpp/ims/imsjava81

3. Edit the sample JVM member DFSJVMEV, which is in IMS.SDFSSMPL:
   a. Change “JavaHome” to the SDK directory. For example:
      /usr/lpp/java/J1.3
   b. Change “imsjavaPath” to pathprefix/usr/lpp/ims/imsjava81

4. Create two HFS files: one for the JMP output and one for errors. The following sample job creates the files JVM.out and JVM.err:

```bash
//name JOB parameters
//TCHMOD PROC TPARM=
//BPX EXEC PGM=BPXBATCH,PARM="&TPARM"
//SYSPRINT DD SYSOUT=*
//STDOUT DD PATH='pathprefix/tchmod.out',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//STDERR DD PATH='pathprefix/tchmod.err',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//PEND
//STEP1 EXEC TCHMOD,TPARM='sh touch path/JVM.out'
//*
//STEP2 EXEC TCHMOD,
// TPARM='sh chmod 777 path/JVM.out'
//*
//STEP3 EXEC TCHMOD,TPARM='sh touch path/JVM.err'
//*
//STEP4 EXEC TCHMOD,TPARM='sh chmod 777 path/JVM.err'
```

5. Edit the DFSJMP procedure, which is in IMS.PROCLIB:
   a. Set the JAVAOUT and JAVAERR DD statements to point to the JVM.out and JVM.err files. For example:
      //JAVAOUT DD PATH='/path/JVM.out'
      //JAVAERR DD PATH='/path/JVM.err'
   b. Set the STEPLIB DD statement to point to the SDFSJLIB data set. This data set contains the DFSCLIB member.
   c. Set the following parameters:
      • JVMOPMAS= IMS.SDFSSMPL data set member DFSJVMMS (master JVM options)
      • JVMOPWKR= IMS.SDFSSMPL data set member DFSJVMWK (worker JVM options)
      • ENVIRON= IMS.SDFSSMPL data set member DFSJVMEV (LIBPATH options)
      • XPLINK=Y if you use SDK 1.4.1
Running the IMS Java IVP in a JMP Region

d. Set any other parameters that are required by your installation. For complete information about the available parameters and DD statements for the DFSJMP procedure, see the *IMS Version 8: Installation Volume 2: System Definition and Tailoring*.

6. Run the JMP procedure.
The JMP region is started.

7. From an IMS terminal, invoke the formatted screen for the transaction by issuing the following command:

   /format IVTCM

   An input screen, as shown in Figure 2, is displayed.

   **************************************************
   * IMS INSTALLATION VERIFICATION PROCEDURE *
   **************************************************

   TRANSACTION TYPE : CONVERSATIONAL
   DATE : 12/11/04

   PROCESS CODE (*1) :

   LAST NAME :

   FIRST NAME :

   EXTENSION NUMBER :

   INTERNAL ZIP CODE :

   (*1) PROCESS CODE

   ADD

   DELETE

   UPDATE

   DISPLAY

   END

   SEGMENT# :

   Figure 2. IVP Input Screen for IMS Java JMP

8. In the PROCESS CODE field, type: DISPLAY. In the LAST NAME field, type: LAST1

   If the IVP was successful, the output that is shown in Figure 3 on page 11 is displayed.
9. Optionally, move the JVM.out and JVM.err files from HFS to a partitioned data set member by submitting the following job:

//name JOB
//MV2PSD EXEC PGM=IKJEFT01
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//O1 DD DISP=SHR,DSN=hlq.dataset(JVMOUT)
//I1 DD PATH='pathPrefix/JVM.out'
//O2 DD DISP=SHR,DSN=hlq.dataset(JVMERR)
//I2 DD PATH='pathPrefix/JVM.err'
//SYSTSIN DD *
OCOPY INDD(I1) OUTDD(O1)
OCOPY INDD(I2) OUTDD(O2)
OCOPY INDD(I3) OUTDD(O3)
/

You can also use this application as a phonebook sample. From the input screen, you can enter the process codes that are listed on the right side of the screen.

---

Running the IMS Java IVP in a JBP Region

To verify that IMS Java is properly installed and that the JBP region is properly configured, run the IMS Java IVP. Details about the PROCLIB members and procedure parameters are in [IMS Version 8: Installation Volume 2: System Definition and Tailoring](#).

**Prerequisites:**

- Ensure that the standard IMS IVPs have been run. These IVPs prepare the DBD for the IVP database, named DFSIVD2, and load the IVP database. They also prepare the IMS Java application PSB (named DFSIVP67) and prepare other IMS control blocks required by the IMS Java IVPs. For details about how to run the IMS IVPs, see [IMS Version 8: Installation Volume 1: Installation Verification](#).
Running the IMS Java IVP in a JBP Region

To run the IMS Java IVP in a JBP region:

1. Ensure that the samples.tar file has been expanded. This topic assumes that its contents are in the directory \texttt{pathprefix/usr/lpp/ims/imsjava81}. For more information on expanding the samples.tar file, see “Installing IMS Java” on page 1.

2. Edit the sample JVM member DFSJVMMS, which is in IMS.SDFSSMPL:
   a. For -Dibm.jvmtrusted.middleware.class.path=, change “ImsjavaPath” to \texttt{pathprefix/usr/lpp/ims/imsjava81}
   b. For -Dibm.jvm.shareable.application.class.path=, change “SamplePath” to \texttt{pathprefix/usr/lpp/ims/imsjava81}

3. Edit the sample JVM member DFSJVMEV, which is in IMS.SDFSSMPL:
   a. Change “JavaHome” to the SDK directory. For example: \texttt{/usr/lpp/java/J1.3}
   b. Change “imsjavaPath” to \texttt{pathprefix/usr/lpp/ims/imsjava81}

4. Create two HFS files: one for the JBP output and one for errors. The following sample job creates the files JVM.out and JVM.err:

   ```
   //name JOB parameters
   //TCHMOD PROC TPARM=
   //BPX EXEC PGM=BPXBATCH,PARM='&TPARM'
   //SYSPRINT DD SYSOUT=* 
   //STDOUT DD PATH='path/tchmod.out',
   //   PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU 
   //STDERR DD PATH='path/tchmod.err',
   //   PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU 
   //PEND 
   //STEP1 EXEC TCHMOD,TPARM='sh touch path/JVM.out'
   //   PEND 
   //STEP2 EXEC TCHMOD,TPARM='sh chmod 777 path/JVM.out'
   //   PEND 
   //STEP3 EXEC TCHMOD,TPARM='sh touch path/JVM.err'
   //   PEND 
   //STEP4 EXEC TCHMOD,TPARM='sh chmod 777 path/JVM.err'
   ```

5. Edit the DFSJBP procedure, which is in IMS.PROCLIB:
   a. Set the JAVAOUT and JAVAERR DD statements to point to the JVM.out and JVM.err files. For example:
      ```
      //JAVAOUT DD PATH='path/JVM.out'
      //JAVAERR DD PATH='path/JVM.err'
      ```
   b. Set the STEPLIB DD statement to point to the SDFSJLIB data set. This data set contains the DFSCLIB member.
   c. Set the following parameters:
      ```
      • JVMOPMAS= IMS.SDFSSMPL data set member DFSJVMMS (master JVM options)
      • JVMOPWKR= IMS.SDFSSMPL data set member DFSJVMWK (worker JVM options)
      • ENVIRON= IMS.SDFSSMPL data set member DFSJVMEV (LIBPATH options)
      • XPLINK=Y if you use SDK 1.4.1
      ```
   d. Set the following EXEC statement parameters to the following:
      ```
      PSB=DFSIVP67 and MBR=DFSJBP.
      ```
Running the IMS Java IVP in a JBP Region

6. Run the JBP procedure.

The JBP region is started, the IVP runs, and output is sent to the JVM.out file.

7. Optionally, move the JVM.out and JVM.err files from HFS to a partitioned data set member by submitting the following job:

```plaintext
//name JOB
//MV2PSD EXEC PGM=IKJEFT01
//SYSPRINT DD SYSOUT=* 
//SYSTSPRT DD SYSOUT=* 
//O1 DD DISP=SHR,DSN=hlq.dataset(JVMOUT) 
//I1 DD PATH='pathPrefix/JVM.out' 
//O2 DD DISP=SHR,DSN=hlq.dataset(JVMERR) 
//I2 DD PATH='pathPrefix/JVM.err' 
//SYSTSIN DD *
OCOPY INDD(I1) OUTDD(O1)
OCOPY INDD(I2) OUTDD(O2)
OCOPY INDD(I3) OUTDD(O3)
/*
```

8. Check the JVMOUT data set or JVM.out file.

If the IVP was successful, the following information is in the output file:

```
LastName = LAST1
FirstName = FIRST1
Extension = 8-111-1111
ZipCode = D01/R01
```

Running the IMS Java Sample Application from a JMP Region

The IMS Java sample application can run in a JMP region. The sample application processes the sample dealership database based on the six available command codes. For an MFS-formatted sample, you can use the IVP application as a phonebook sample. From the input screen of the IVP, you can enter the process codes that are listed on the right side of the screen.

The source files for the sample application are in the HFS directory `pathprefix/usr/lpp/ims/imsjava81/samples/dealership/ims`.

**Prerequisites:**

- “Running the IMS Java IVP in a JBP Region” on page 8
- Appendix A, “Preparing to Run the Dealership Samples,” on page 129

To run the IMS Java sample application from a JMP region:

1. Edit the sample JVM member DFSJVMAP, which is in IMS.SDFSXMPL, by adding the following line:

   ```plaintext
   AUTPSB11=samples/dealership/ims/IMSAuto
   ```

2. Following the directions provided in the sample JVM members, edit the following three sample JVM members, which are in IMS.SDFSSMPL: DFSJVMAP, DFSJVMM, and DFSJVMEV.

3. Create two HFS files: one for the JMP output and one for errors. The following sample job creates the files JVM.out and JVM.err:

   ```plaintext
   //name JOB parameters
   //TCHMOD PROC TPARM=
   //BPX EXEC PGM=BPX BATCH,PARM="TPARM"
   ```
Running the Sample Application from a JMP Region

```
//SYSPRINT DD SYSOUT=*,
//STDOUT DD PATH='/path/tchmod.out',
PATHOPTS=(OWRONLY,CREAT,TRUNC),PATHMODE=SIRWXU
//STDERR DD PATH='/path/tchmod.err',
PATHOPTS=(OWRONLY,CREAT,TRUNC),PATHMODE=SIRWXU
PEND
//STEP1 EXEC TCHMOD,TPARM='sh touch /path/JVM.out'
//STEP2 EXEC TCHMOD, TPARM='sh chmod 777 /path/JVM.out'
//STEP3 EXEC TCHMOD,TPARM='sh touch /path/JVM.err'
//STEP4 EXEC TCHMOD,TPARM='sh chmod 777 /path/JVM.err'
```

4. Edit the DFSJMP procedure, which is in IMS.PROCLIB:
   a. Set the JAVAOUT and JAVAERR DD statements to point to the files that are
      created in step 3 on page 13. For example:
      ```
      //JAVAOUT DD PATH='/path/JVM.out'
      //JAVAERR DD PATH='/path/JVM.err'
      ```
   b. Set the STEPLIB DD statement to point to the SDFSJLIB data set. This data
      set contains the DFSCLIB member.
   c. Set the following parameters:
      - JVMOPMAS= IMS.SDFSSMPL data set member DFSJVMMS (master
        JVM options)
      - JVMOPWK= IMS.SDFSSMPL data set member DFSJVMWK (worker
        JVM options)
      - ENVIRON= IMS.SDFSSMPL data set member DFSJVMEV (LIBPATH
        options)
      - XPLINK=Y if you use SDK 1.4.1
   d. Set any other parameters that are required by your installation. For complete
      information about the available parameters and DD statements for the
      DFSJMP procedure, see the IMS Version 8: Installation Volume 2: System
      Definition and Tailoring

5. Run the JMP procedure.
   The JMP region is started.

6. From an IMS terminal, execute the application by issuing one of the following
   six commands and parameters:
   - LISTMODELS. For example:
     ```
     LISTMODELS
     ```
   - FINDCAR. For example:
     ```
     FINDCAR FORD V234567890123456789
     ```
   - MODELDETAILS. For example:
     ```
     MODELDETAILS VOLVO S40 2002
     ```
   - RECORDSALE. For example:
     ```
     RECORDSALE 1235 S999302042002LAST9 V987654321123456782VOLVO S40
     ```
   - ACCEPTORDER. For example:
     ```
     ACCEPTORDER 123457LAST9 FIRST9 05-18-20011:23:34
     ```
   - CANCELORDER. For example:
     ```
     CANCELORDER 1234571235
     ```
This topic describes how to set up a JMP or JBP region to access DB2 UDB for z/OS databases. It does not describe how to set up DB2 UDB for z/OS for access from IMS. For information about setting up DB2 UDB for z/OS for Java application access, see DB2 Universal Database for OS/390 and z/OS: Application Programming Guide and Reference for Java. Note that you must create a DB2 plan for each PSB (usually each Java application) that is used to access DB2 UDB for z/OS.

JMP and JBP applications can access DB2 UDB for z/OS databases. For JMP or JBP applications to have DB2 UDB for z/OS access, you must attach DB2 UDB for z/OS to IMS using the DB2 Recoverable Resource Manager Services attachment facility (RRSAF). Unlike other dependent regions, JMP and JBP regions do not use the External Subsystem Attach Facility (ESAF).

DB2 UDB for z/OS provides different JDBC drivers:

- JDBC/SQLJ driver for DB2 for OS/390 and z/OS with JDBC 2.0 support (called the DB2 JDBC/SQLJ 2.0 driver), which allows access to DB2 UDB for z/OS databases only when IMS is on the same z/OS image as DB2 UDB for z/OS. This is a type 2 JDBC driver.
- JDBC/SQLJ driver for DB2 for OS/390 and z/OS with JDBC 1.2 support (called the DB2 JDBC/SQLJ 1.2 driver), which allows access to DB2 UDB for z/OS databases only when IMS is on the same z/OS image as DB2 UDB for z/OS. This is a type 2 JDBC driver.
- DB2 Universal JDBC driver, which allows access to DB2 UDB for z/OS databases from IMSs that are on different z/OS images from DB2 UDB for z/OS when you use the Universal Driver type 4 connectivity. You can also use the type 2 implementation of this driver for access to DB2 UDB for z/OS databases when IMS is on the same z/OS image as DB2 UDB for z/OS.

All of these drivers are referred to in this topic as DB2 JDBC drivers.

For type 2 JDBC drivers, you must use the default connection URL in the application program. For example, jdbc:db2os390: or db2:default:connection.

For type 4 JDBC drivers, you can use a specific connection URL in the application program.

With RRSAF, the dependent region builds an attachment thread to DB2 UDB for z/OS using RRS. RRS coordinates the commits of the updates that the application program makes to both IMS and DB2 UDB for z/OS resources. IMS is a participant, not the coordinator, of these updates and commits.

To attach a DB2 UDB for z/OS subsystem to IMS using RRSAF for JMP or JBP access to DB2 UDB for z/OS databases:

1. Create an IMS.PROCLIB member for information about the DB2 UDB for z/OS subsystem. The member name must follow the same naming conventions you follow when you attach DB2 UDB for z/OS with ESAF.

In the IMS.PROCLIB member, define the following three parameters for the DB2 subsystem that JMP and JBP applications need access to:

SST=DB2, SSN=db2name, COORD=RRS
2. In the trusted middleware class path of the DFSJVMMS IMS.SDFSSMPL member, add the following paths:
   • Path to the ZIP file of the DB2 JDBC driver
   • Path to the ZIP file and ZIP file name of the DB2 JDBC driver
   For example:
   `-Dibm.jvm.trusted.middleware.class.path=>
     /usr/lpp/db2/db2710/classes:
     /usr/lpp/db2/db2710/classes/db2j2classes.zip`
3. In the DFSJVMMEV IMS.SDFSSMPL member, add the path to the SO file of the DB2 JDBC driver to the LIBPATH= environment variable. For example:
   `LIBPATH=/usr/lpp/db2/db2710/lib`
4. Add the following parameters to the IMS control region EXEC statement:
   `SSM=name
   RRS=Y`
5. In the DFSJMP or DFSJBP procedure of the region that has access to DB2 UDB for z/OS, add the DFSDB2AF DD statement to point to the DB2 UDB for z/OS libraries, which must be APF-authorized.

**Related Reading:** For details about the IMS.PROCLIB member and procedure parameters, see the external subsystem information of [IMS Version 8: Installation Volume 2: System Definition and Tailoring](http://www.ibm.com/ims) For information about the DB2 JDBC drivers, see [DB2 Universal Database for OS/390 and z/OS: Application Programming Guide and Reference for Java](http://www.ibm.com/ims).

### Developing JMP Applications

JMP applications access the IMS message queue to receive messages to process and to send output messages. Therefore, you must define input and output message classes by subclassing the IMSFieldMessage class. The IMS Java class libraries provide the capability to process IMSFieldMessage objects. JMP applications commit or roll back the processing of each message by calling `IMSTransaction.getTransaction().commit()` or `IMSTransaction.getTransaction().rollback()`.

**Related Reading:** For details about the classes you use to develop a JMP application, see the IMS Java API Specification, which is available on the IMS Java Web site. Go to [http://www.ibm.com/ims](http://www.ibm.com/ims) and link to the IMS Java page.

The following topics provide additional information:

- "Subclassing the IMSFieldMessage Class to Define Input Messages"
- “Subclassing the IMSFieldMessage Class to Define Output Messages” on page 17
- “Implementing the main Method” on page 18
- “JMP Programming Models” on page 19
- “Additional Message Handling Considerations for JMP Applications” on page 21

### Subclassing the IMSFieldMessage Class to Define Input Messages

Figure 4 on page 17 gives an example of subclassing the IMSFieldMessage class. This class defines an input message that accepts a 2-byte type code of a car model to query a car dealership database for available car models.
This example code subclasses the IMSFieldMessage class to make the fields in the message available to the program and creates an array of DLITypeInfo objects for the fields in the message. For the DLITypeInfo class, the code identifies first the field name, then the data type, the position, and finally the length of the individual fields within the array. This allows the application to use the access functions within the IMSFieldMessage class hierarchy to automatically convert the data from its format in the message to a Java type that the application can process. In addition to the message-specific fields it defines, the IMSFieldMessage class provides access functions that allow it to determine the transaction code and the length of the message.

```java
package dealership.application;
import com.ibm.ims.db.*;
import com.ibm.ims.base.*;
import com.ibm.ims.application.*;

/* Subclasses IMSFieldMessage to define application's input messages */
public class InputMessage extends IMSFieldMessage {

    /* Creates array of DLITypeInfo objects for the fields in message */
    final static DLITypeInfo[] fieldInfo={
        new DLITypeInfo("ModelTypeCode", DLITypeInfo.CHAR, 1, 2)
    );

    public InputMessage() {
        super(fieldInfo, 2, false);
    }
}
```

*Figure 4. Subclass IMSFieldMessage: Input Message Sample Code*

**Subclassing the IMSFieldMessage Class to Define Output Messages**

*Figure 5 on page 18* gives a sample of subclassing IMSFieldMessage to define an output message that displays the available car models from a type code query.

This sample code creates an array of DLITypeInfo objects and then passes that array, the byte array length, and the boolean value false, which indicates a non-SPA message, to the IMSFieldMessage constructor. For each DLITypeInfo object, you must first identify the field data type, then the field name, the field offset in the byte array, and finally the length of the byte array.
Developing JMP Applications

```java
package dealership.application;
import com.ibm.ims.db.*;
import com.ibm.ims.base.*;
import com.ibm.ims.application.*;

/*Subclasses IMSFieldMessage to define application's output messages */
public class ModelOutput extends IMSFieldMessage {
  /* Creates array of DLITypeInfo objects for the fields in message */
  final static DLITypeInfo[] fieldInfo=
  {new DLITypeInfo("Type",        DLITypeInfo.CHAR, 1, 2),
   new DLITypeInfo("Make",        DLITypeInfo.CHAR, 3, 10),
   new DLITypeInfo("Model",       DLITypeInfo.CHAR, 13, 10),
   new DLITypeInfo("Year",        DLITypeInfo.DOUBLE, 23, 4),
   new DLITypeInfo("CityMiles",   DLITypeInfo.CHAR, 27, 4),
   new DLITypeInfo("HighwayMiles",DLITypeInfo.CHAR, 31, 4),
   new DLITypeInfo("Horsepower",  DLITypeInfo.CHAR, 35, 4)
  };

  public ModelOutput() {
    super(fieldInfo, 38,false);
  }
}
```

Figure 5. Subclass IMSFieldMessage: Output Message Sample Code

## Implementing the main Method

The main method (public static void main(String[] args)) is the entry point into all JMP and JBP applications.

The sample code shown in Figure 6 on page 19 demonstrates how to perform the following actions:

1. Query the database for a specific model that matches the input model type code. This method is not implemented yet and is explained more fully in Chapter 6, “JDBC Access to IMS Data,” on page 73.
2. Return detailed information about that specific model as output if it is available at the dealership.
3. Return an error message if the model is not available at the dealership.
Developing JMP Applications

Note: The IMSMessageQueue.getUniqueMessage method returns true if a message
was read from the queue and false if one was not. Also, the
IMSTransaction.getTransaction().commit method must be called before receiving
subsequent messages from the queue.

JMP Programming Models

JMP applications get input messages from the IMS message queue, access IMS
databases, commit transactions, and can send output messages.

JMP applications are started when IMS receives a message with a transaction code
for the JMP application and schedules the message. JMP applications end when
there are no more messages with that transaction code to process.

JMP Application Without Rollback

A transaction begins when the application gets an input message and ends when
the application commits the transaction. To get an input message, the application
calls the getUniqueMessage method. The application must commit or rollback any
database processing. The application must issue a commit call immediately before
calling subsequent getUniqueMessage methods.

```java
public static void main(String args[]) {
    conn = DriverManager.getConnection(...); //Establish DB connection
```
Developing JMP Applications

```java
while(MessageQueue.getUniqueMessage(...)){ //Get input message, which
    //starts transaction
    results=statement.executeQuery(...); //Perform DB processing
    ... MessageQueue.insertMessage(...); //Send output messages
    ... IMSTransaction.getTransaction().rollback(); //Roll back DB processing
        //and output messages
    results=statement.executeQuery(...); //Perform more DB processing
        //(optional)
    ... MessageQueue.insertMessage(...); //Send more output messages
        //(optional)
    ... IMSTransaction.getTransaction().commit(); //Commit and end transaction
}
conn.close(); //Close DB connection
return;
```

**JMP Application that Uses Rollback**

A JMP application can roll back database processing and output messages any number of times during a transaction. A rollback call backs out all database processing and output messages to the most recent commit. The transaction must end with a commit call when the program issues a rollback call, even if no further database or message processing occurs after the rollback call.

```java
public static void main(String args[])
{
    conn = DriverManager.getConnection(...); //Establish DB connection
    while(MessageQueue.getUniqueMessage(...)){ //Get input message, which
        //starts transaction
        results=statement.executeQuery(...); //Perform DB processing
        ... MessageQueue.insertMessage(...); //Send output messages
        ... IMSTransaction.getTransaction().rollback(); //Roll back DB processing
            //and output messages
        results=statement.executeQuery(...); //Perform more DB processing
            //(optional)
        ... MessageQueue.insertMessage(...); //Send more output messages
            //(optional)
        ... IMSTransaction.getTransaction().commit(); //Commit and end transaction
    }
    conn.close(); //Close DB connection
    return;
}
```

**JMP Application that Accesses IMS or DB2 UDB for z/OS Data**

When a JMP application accesses only IMS data, it needs to open a database connection only once to process multiple transactions, as shown in "JMP Application Without Rollback" on page 19. However, a JMP application that accesses DB2 UDB for z/OS data must open and close a database connection for each message that is processed. The following model is valid for DB2 UDB for z/OS database access, IMS database access, or both DB2 UDB for z/OS and IMS database access.

**Related Reading:** For more information about accessing DB2 data from a JMP application, see "Accessing DB2 UDB for z/OS Databases from JMP or JBP Applications" on page 32.

```java
public static void main(String args[])
{
    while(MessageQueue.getUniqueMessage(...)){ //Get input message, which
        //starts transaction
```
Developing JMP Applications

```java
conn = DriverManager.getConnection(...); //Establish DB connection
results=statement.executeQuery(...); //Perform DB processing
... MessageQueue.insertMessage(...); //Send output messages
... conn.close(); //Close DB connection
IMSTransaction.getTransaction().commit(); //Commit & end transaction
}

return;
```

Additional Message Handling Considerations for JMP Applications

JMP applications access the IMS message queue in addition to IMS or DB2 UDB for z/OS databases. This topic provides information about specific programming considerations for the IMS message queue.

In this topic:
- "Conversational Transactions"
- "Handling Multi-Segment Messages" on page 23
- "Coding and Accessing Messages with Repeating Structures" on page 24
- "Flexible Reading of Multiple Input Messages" on page 25

Conversational Transactions

A conversational program runs in a JMP region and processes conversational transactions that are made up of several steps. It does not process the entire transaction at the same time. A conversational program divides processing into a connected series of terminal-to-program-to-terminal interactions. Use conversational processing when one transaction contains several parts.

A nonconversational program receives a message from a terminal, processes the request, and sends a message back to the terminal. A conversational program receives a message from a terminal and replies to the terminal, but it saves the data from the transaction in a scratch pad area (SPA). Then, when the person at the terminal enters more data, the program has the data it saved from the last message in the SPA, so it can continue processing the request without the person at the terminal having to enter the data again. The application package classes enable applications to be built using IMS Java.

Related Reading: For more information about conversational and nonconversational transaction processing, see *IMS Version 8: Administration Guide: Transaction Manager*.

**Defining a SPA Message in a Conversational Program:** To define a SPA message in a conversational program:

1. Define the SPA message (including the boolean as a SPA parameter). By default, all messages going to (input) and from (output) a Java application are transmitted as EBCDIC character data. To use a different type of encoding, you must call the `IMSFieldMessage` class inherited method `setDefaultEncoding` and provide the new encoding type. This encoding can be any Java-supported encoding type. In Figure 7 on page 22, the default encoding is specified as UTF-8.
2. Read the SPA message before reading the application messages:

```java
try {
    // Get the SPA data
    msgReceived = msgQ.getUniqueMessage(spaMessage);
}
catch (IMSException e) {
    if (e.getStatusCode() != JavaToDLI.MESSAGE_QUEUED_PRIOR_TO_LAST_START)
        throw e;
}
if (!msgReceived)
    outputMessage.setString("Message","UNABLE TO READ SPA");
else if (!msgQ.getNextMessage(inputMessage))
    // No input message received
    outputMessage.setString("Message","NO INPUT MESSAGE");
else if ((spaMessage.getShort("SessionNumber")==0)
    && (inputMessage.getString("ProcessCode").trim().equals("END"))
    && (inputMessage.getString("LastName").trim().equals("")))
    // New Conversation. User has to specify last name.
    outputMessage.setString("Message","LAST NAME WAS NOT SPECIFIED");
else {
    // Set spa data fields
    spaMessage.setString("LastName", inputMessage.getString("LastName"));
    spaMessage.setString("FirstName", inputMessage.getString("FirstName"));
    spaMessage.setString("Extension", inputMessage.getString("Extension");
    spaMessage.incrementSessionNumber();
    msgQ.insertMessage(spaMessage);
}
```

Figure 8. Reading a SPA Message

3. Write the SPA message before sending any output messages:

```java
public SPAMessage() {
    super(fieldInfo, 66, true);
   setDefaultEncoding("UTF-8");
}
```

Figure 9. Writing a SPA Message
4. End the conversation by using the version of the insertMessage method that contains a boolean isLast argument set to true:

```java
msgQ.insertMessage(spaMessage, true);
```

**Conversational Transaction Sequence of Events:** When the message is a conversational transaction, the following sequence of events occurs:

1. IMS removes the transaction code and places it at the beginning of a message segment. The message segment is equal in length to the SPA that was defined for this transaction during system definition. This is the first segment of the input message that is made available to the program. The second through the nth segments from the terminal, minus the transaction code, become the remainder of the message that is presented to the application program.

2. After the conversational program prepares its reply, it inserts the SPA to IMS. The program then inserts the actual text of the reply as segments of an output message.

3. IMS saves the SPA and routes the message to the input LTERM (logical terminal).

4. If the SPA insert specifies that another program is to continue the same conversation, the total reply (including the SPA) is retained on the message queue as input to the next program. This program then receives the message in a similar form.

5. A conversational program must be scheduled for each input exchange. The other processing continues while the operator at the input terminal examines the reply and prepares new input messages.

6. To terminate a conversation, the program places blanks in the transaction code field of the SPA and inserts the SPA to IMS. In IMS Java this happens when you call IMSMessageQueue.insertMessage with the boolean parameter isLast set to true.

7. The conversation can also be terminated if the transaction code in the SPA is replaced by any nonconversational program's transaction code, and the SPA is inserted to IMS. After the next terminal input, IMS routes that message to the other program's queue in the normal way.

**Handling Multi-Segment Messages**

Message-driven applications can have multi-segment input messages. That is, more than one message needs to be read from the message queue in order to retrieve the entire message. When this occurs, you must provide a mapping for each message that is to be read from the queue and use the appropriate methods available from the IMSMessageQueue class.

The following code defines two input messages that comprise a multi-segment message:

```java
public class InputMessage1 extends IMSFieldMessage {
    final static DLITypeInfo[] segmentInfo = {
        new DLITypeInfo("Field1", DLITypeInfo.CHAR, 1, 10),
        new DLITypeInfo("Field2", DLITypeInfo.INTEGER, 11, 4)
    };

    public InputMessage1() {
        super(segmentInfo, 14, false);
    }
}

public class InputMessage2 extends IMSFieldMessage {
```
Developing JMP Applications

```
final static DLITypeInfo[] segmentInfo = {
    new DLITypeInfo("Field1", DLITypeInfo.CHAR, 1, 10),
    new DLITypeInfo("Field2", DLITypeInfo.CHAR, 11, 8)  
};

public InputMessage2() {
    super(segmentInfo, 18, false);
}
```

The following code shows how the message queue is used to retrieve both messages:

```
//Create a message queue
IMSMessageQueue messageQueue = new IMSMessageQueue();
//Create the first input message
InputMessage1 input1 = new InputMessage1();
//Create the second input message
InputMessage2 input2 = new InputMessage2();

try {
    //Read the first message from the queue
    messageQueue.getUniqueMessage(input1);

    //Read the second message from the queue
    messageQueue.getNextMessage(input2);

    ...
} catch (IMSException e) {
    ...
}
```

### Coding and Accessing Messages with Repeating Structures

Messages with repeating structures can be defined by using the `DLITypeInfoList` class. With the `DLITypeInfoList` class, you can specify a repeating list of fields and the maximum number of times the list can be repeated. These repeating structures can contain repeating structures.

**Figure 10** is a sample output message that contains a set of Make, Model, and Color fields, with a count field to identify how many occurrences were stored:

```
public class ModelOutput extends IMSFieldMessage {
    static DLITypeInfo[] modelTypeInfo = {
        new DLITypeInfo("Make", DLITypeInfo.CHAR, 1, 20),
        new DLITypeInfo("Model", DLITypeInfo.CHAR, 21, 20),
        new DLITypeInfo("Color", DLITypeInfo.CHAR, 41, 20),
    };
    static DLITypeInfo[] modelTypeInfoList = {
        new DLITypeInfo("ModelCount", DLITypeInfo.INTEGER, 1, 4),
        new DLITypeInfoList("Models", modelTypeInfo, 5, 60, 100),
    };
    public ModelOutput() {
        super(modelOutputTypeInfo, 6004, false);
    }
}
```

**Figure 10. Sample Output Message with Repeating Structures**

To access the nested structures that are defined in a `DLITypeInfoList` object, use a dotted notation to specify the fields and the index of the field within a repeating structure. This dotted notation can use either the field names or field indexes. For example, the “Color” field in the fourth “Models” definition in the `ModelOutput` object is accessed as “Models.4.Color” within the `ModelOutput` message. The following code sets the fourth “Color” in the `ModelOutput` message to “Red.”
The following code uses field indexes instead of field names to make the same change to the `ModelOutput` message:

```java
ModelOutput output = new ModelOutput();
output.setString("2.4.3", "Red");
```

**Flexible Reading of Multiple Input Messages**

There are times when an application needs to process multiple input messages that require different input data types. For example, the car dealership sample application supports requests to list models, show model details, find cars, cancel orders, and record sales. Each of these requests requires different input data. The following steps explain how to define the messages to support these requests, and how to access the messages from the application.

1. Define the primary input message. The primary input message is the message that you pass to the `IMSMessageQueue.getUniqueMessage` method to retrieve all of your input messages. Your primary input message must have an I/O area that is large enough to contain any of the input requests that your application might receive. It must also contain at least one field in common with all of your input messages. This common field allows you to determine the input request. In the example in Figure 11, the common field is `CommandCode`, and the maximum length of each message is 64 (the number passed to the `IMSFieldMessage` constructor):

   ```java
   public class InputMessage extends IMSFieldMessage {
      final static DLITypeInfo[] fieldInfo = {
            new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
      };

      public InputMessage(DLITypeInfo[] fieldInfo) {
         super(fieldInfo, 64, false);
      }
   }
   ```

   **Figure 11. Defining the Primary Input Message**

2. Define separate input messages for each request. Each of these input messages contains the same `CommandCode` field as its first field. Each of these input messages also uses an `IMSFieldMessage` constructor that takes an `IMSFieldMessage` object and a `DLITypeInfo` array. The `IMSFieldMessage` constructor allows you to remap the contents of the primary input message using the same type of information with each request; therefore, you do not copy the I/O area of the message, only a reference to this area. Figure 12 on page 26 illustrates code that creates the input messages for the requests `ShowModelDetails`, `FindACar`, and `CancelOrder`.

```
```
Developing JMP Applications

```java
public class ShowModelDetailsInput extends IMSFieldMessage {
    final static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
        new DLITypeInfo("ModelTypeCode", DLITypeInfo.CHAR, 21, 2),
    };
    public ShowModelDetailsInput(InputMessage inputMessage) {
        super(inputMessage, fieldInfo);
    }
}

public class FindACarInput extends IMSFieldMessage {
    final static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
        new DLITypeInfo("Make", DLITypeInfo.CHAR, 21, 10),
        new DLITypeInfo("Model", DLITypeInfo.CHAR, 31, 10),
        new DLITypeInfo("Year", DLITypeInfo.CHAR, 41, 4),
        new DLITypeInfo("LowPrice", DLITypeInfo.PACKEDDECIMAL, 45, 5),
        new DLITypeInfo("HighPrice", DLITypeInfo.PACKEDDECIMAL, 50, 5),
        new DLITypeInfo("Color", DLITypeInfo.CHAR, 55, 10),
    };
    public FindACarInput(InputMessage inputMessage) {
        super(inputMessage, fieldInfo);
    }
}

public class CancelOrderInput extends IMSFieldMessage {
    final static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
        new DLITypeInfo("OrderNumber", DLITypeInfo.CHAR, 21, 6),
        new DLITypeInfo("DealerNumber", DLITypeInfo.CHAR, 21, 6),
    };
    public CancelOrderInput(InputMessage inputMessage) {
        super(inputMessage, fieldInfo);
    }
}
```

Figure 12. Defining Separate Input Messages for Each Request

Note the following details about Figure 11 on page 25 and Figure 12:

- The CommandCode field is defined within every class at lines A, C, E, and G. This field must be defined in every message that reads the command code. If you do not define the field, you must adjust the offsets of the following fields to account for the existence of the CommandCode in the byte array. For example, you can delete the DLITypeInfo entry for CommandCode in the CancelOrderInput class, but the OrderNumber field must still start at offset 21.
- The length of the base class InputMessage must be large enough to contain any of its subclasses. In this example, the InputMessage class is 65 bytes because the fields of the FindACarInput method require it B.
- Each InputMessage subclass must provide a constructor to create itself from an InputMessage object, as in lines D, F, and H. This constructor uses a new constructor in the IMSFieldMessage class, called a copy constructor.
Given this design, an application can provide message-reading logic similar to that shown in Figure 13.

```java
while (getUniqueMessage(inputMessage)) {
    string commandCode=inputMsg.getString("CommandCode").trim();
    if (commandCode.equals("ShowModelDetails")) {
        showModelDetails(new ShowModelDetailsInput(inputMessage));
    } else if(commandCode.equals("FindACar")) {
        findACar(new FindACarInput(inputMessage));
    } else {
        //process an error
    }
}
```

Figure 13. Message-Reading Logic

Developing JBP Applications

JBP applications do not access the IMS message queue, and therefore you do not need to subclass the IMSFieldMessage class.

**Related Reading:** For details about the classes you use to develop a JBP application, see the IMS Java API Specification, which is available on the IMS Java Web site. Go to http://www.ibm.com/ims and link to the IMS Java page.

The following topics provide additional information:
- [JBP Programming Models](#)

**JBP Programming Models**

JBP applications are similar to JMP applications, except that JBP applications do not receive input messages from the IMS message queue. The program should periodically issue commit calls, except for applications that have the PSB PROCOPT=GO parameter.

Unlike BMP applications, JBP applications must be non-message-driven applications.

**JBP Application without Rollback**

A JBP application connects to a database, performs database processing, periodically commits, and disconnects from the database at the end of the program. The program must issue a final commit before ending.

```java
public static void main(String args[]) {
    conn = DriverManager.getConnection(...);  //Establish DB connection
    repeat {
        repeat {
            results=statement.executeQuery(...); //Perform DB processing
            ...  
            MessageQueue.insertMessage(...);  //Send output messages
        }
    }
```
Developing JBP Applications

```java
IMSTransaction.getTransaction().commit(); //Periodic commits divide work

conn.close(); //Close DB connection
return;
}
```

**JBP Application using Rollback**

Similarly to JMP applications, JBP applications can also roll back database processing and output messages. A final commit call is required before the application can end, even if no further database processing occurs or output messages are sent after the last rollback call.

```java
public static void main(String args[]) {
    conn = DriverManager.getConnection(...); //Establish DB connection
    repeat {
        repeat {
            results = statement.executeQuery(...); //Perform DB processing
            ...
            MessageQueue.insertMessage(...); //Send output messages
            ...
            IMSTransaction.getTransaction().rollback(); //Roll out DB processing and output messages
            results = statement.executeQuery(...); //Perform more DB processing (optional)
            ...
            MessageQueue.insertMessage(...); //Send more output messages (optional)
            ...
        }
        IMSTransaction.getTransaction().commit(); //Periodic commits divide work
    }
    conn.close(); //Close DB connection
    return;
}
```

**JBP Application that Accesses DB2 UDB for z/OS or IMS Data**

Like a JBP application that accesses IMS data, a JBP application that accesses DB2 UDB for z/OS data connects to a database, performs database processing, periodically commits, and disconnects from the database at the end of the application. However, the application must also issue a final commit after closing the database connection.

The following model is valid for DB2 UDB for z/OS database access, IMS database access, or both DB2 UDB for z/OS and IMS database access.

**Related Reading:** For more information about accessing DB2 UDB for z/OS data from a JBP application, see "Configuring JMP and JBP Regions for DB2 UDB for z/OS Database Access" on page 15.

```java
public void doBegin() ... { //Application logic runs
    conn = DriverManager.getConnection(...); //Establish method
    repeat {
        repeat {
            results = statement.executeQuery(...); //Perform DB processing
            ...
        }
    }
    ...
```
Enterprise COBOL Interoperability with JMP and JBP Applications

IMS Enterprise COBOL for z/OS and OS/390 Version 3 Release 2 supports interoperation between COBOL and Java languages when running in a JMP or JBP region. With this support, you can:

- Call an object-oriented (OO) COBOL application from an IMS Java application by building the front-end application, which processes messages, in Java, and the back end, which processes databases, in OO COBOL.
- Build an OO COBOL application containing a main routine that can invoke Java routines.

Restriction: COBOL applications that run in an IMS Java dependent region must use the AIB interface, which requires that all PCBs in a PSB definition have a name.

You can access COBOL code in a JMP or JBP region because Enterprise COBOL provides object-oriented language syntax that enables you to:

- Define classes with methods and data implemented in COBOL
- Create instances of Java and COBOL classes
- Invoke methods on Java and COBOL objects
- Write classes that inherit from Java classes or other COBOL classes
- Define and invoke overloaded methods

In Enterprise COBOL programs, you can call the services provided by the JNI to obtain Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language.

In Enterprise COBOL classes, you can code CALL statements that interface with procedural COBOL programs. Therefore, COBOL class definition syntax can be especially useful for writing wrapper classes for procedural COBOL logic, enabling existing COBOL code to be accessed from Java.

Java code can create instances of COBOL classes, invoke methods of these classes, and can extend COBOL classes.

Related Reading: For details building applications that use Enterprise COBOL and that run in an IMS Java dependent region, see Enterprise COBOL for z/OS and OS/390: Programming Guide.

The following topics provide additional information:

- “Enterprise COBOL as a Back-End Application in a JMP or JBP Region” on page 30
**Enterprise COBOL as a Front-End Application in a JMP or JBP Region**

When you define an OO COBOL class and compile it with the Enterprise COBOL compiler, the compiler generates a Java class definition with native methods and the object code that implements the native methods. After compiling the class, you can create an instance and invoke the methods of the class from a Java program that runs in a JMP or JBP region. For example, you can define an OO COBOL class with the appropriate DL/I call in COBOL to access an IMS database.

When Java is the front-end language, you must perform all message-queue and message-synchronization processing in Java.

For example, you must call both the IMSMessageQueue.getUniqueMessage method (to read messages from the queue) and the IMSTransaction.getTransaction().commit() method (to commit changes) before reading subsequent messages from the message queue or exiting the application.

In the back-end application, you can access IMS databases by either using Java or calling a COBOL routine.

You can use the COBOL STOP RUN statement in the COBOL part of an application that runs in an JMP or JBP region. However, this statement terminates all COBOL and Java routines, including the JVM, and returns control immediately to IMS with both the program and transaction left in a stopped state.

**Important:** Do not mix the languages that are used to read messages from the message queue or to commit resources. The IMS Java library tracks the calls that are made in Java to ensure that the syncpoint rules are followed, but it does not track calls made in COBOL.

For example, you can define an OO COBOL class with the appropriate DL/I call in COBOL to access an IMS database. To make the implementation of this class available to an IMS Java program:

1. Compile the COBOL class with the Enterprise COBOL compiler to generate a Java source file, which contains the class definition, and an object module, which contains the implementation of the native methods.
2. Compile the generated Java source file with the Java compiler to create the application class file.
3. Link the object module into a dynamic link library (DLL) in the HFS file (.so).
4. Update the application class path (ibm.jvm.application.class.path) for the JMP or JBP region to allow access to the Java class file.
5. Update the library path for the JMP or JBP region to allow access to the DLL.

**Enterprise COBOL as a Front-End Application in a JMP or JBP Region**

The object-oriented syntax of Enterprise COBOL enables you to build COBOL applications with a `main` method, which can be run directly in a JMP or JBP region. The JMP or JBP region locates, instantiates, and invokes this `main` method in the same way it does for the `main` method of a Java application.
You can write an application for an JMP or JBP region entirely with OO COBOL, but a more likely use for a front-end COBOL application is to call a Java routine from a COBOL application.

When running within the JVM of an JMP or JBP region, Enterprise COBOL run-time support automatically locates and uses this JVM to invoke methods on Java classes.

A front-end OO COBOL application with a main routine that runs in a JMP or JBP region has the same requirements as a Java program that runs in a JMP or JBP region.

The COBOL application must commit resources before reading subsequent messages or exiting the application. A COBOL GU call does not implicitly commit resources when the program is running in a JMP or JBP region as it does when the program is running in an MPP region.

Use D/I calls for message processing (GU and GN) and transaction synchronization (CHKP). A CHKP call in a JMP or JBP region does not automatically retrieve a message from the message queue.

You can use the COBOL STOP RUN statement in the COBOL part of an application that runs in a JMP or JBP region. However, this statement terminates all COBOL and Java routines, including the JVM, and returns control immediately to IMS with both the program and transaction left in a stopped state.

**Performance Consideration for OO COBOL in a JMP or JBP Region**

COBOL code in a JMP or JBP dependent region affects performance. Because COBOL class methods are implemented in native code, the JVM cannot be reset after a transaction that uses COBOL routines runs.

IBM's Persistent Reusable Java Virtual Machine is specifically designed to treat applications that invoke native code as untrusted. After a transaction runs that contains COBOL routines, IMS ends the current JVM and creates a fresh JVM before scheduling the next transaction. Only classes in the trusted middleware class path ibm.jvm.middleware.class.path can call native routines without affecting JVM reset.

**Related Reading:** For more information about the Persistent Reusable Java Virtual Machine, see *IBM Developer Kit for OS/390, Java 2 Technology Edition: New IBM Technology featuring Persistent Reusable Java Virtual Machines*.

**Recommendation against Accessing Databases with Both Java and COBOL**

IBM recommends that you do not access the same DB PCB from both Java and COBOL. The COBOL and Java parts of an application share a single database pointer (or cursor). If the same DB PCB is accessed by both Java and COBOL, database positioning as a result of calls in one language affect the database positioning for calls in the other language.

For example, if you build a SQL SELECT clause and use JDBC to query and retrieve results, the IMS Java class library constructs the appropriate request to IMS to establish the correct position in the database. If you then call a COBOL routine, which builds an SSA and runs a GU request to IMS against the same DB PCB, the GU request will likely change the position in the database for that DB PCB. If the
position is changed, subsequent JDBC requests using the same SQL SELECT clause to retrieve more records will be wrong because the database position has changed.

If you must access the same DB PCB from multiple languages, establish database positioning again when returning from an inter-language call before accessing more records in the database.

Note: Although IBM advises caution for language interoperability, the behavior described in this section is not related to the programming languages themselves. Two parts of the same application that both access the same DB PCB can have the same behavior described in this section even if both parts are written in the same language.

Accessing DB2 UDB for z/OS Databases from JMP or JBP Applications

A JMP or JBP application can access DB2 UDB for z/OS databases by using the DB2 JDBC/SQILJ 2.0 driver or the DB2 JDBC/SQILJ 1.2 driver. The JMP or JBP region that the application is running in must also be defined with DB2 UDB for z/OS attached by the DB2 Recoverable Resource Manager Services attachment facility (RRSAF).

Related Reading: For information about attaching DB2 UDB for z/OS to IMS for JMP or JBP application access to DB2 UDB for z/OS databases, see "Configuring JMP and JBP Regions for DB2 UDB for z/OS Database Access" on page 15.

Accessing DB2 UDB for z/OS data from a JMP or JBP application is similar to accessing IMS data. When writing a JMP or JBP application that accesses DB2 UDB for z/OS data, consider both the differences from IMS database access and the differences from accessing DB2 UDB for z/OS data in other environments:

- You can have only one active DB2 UDB for z/OS connection open at any time.
- For type 2 JDBC drivers, you must use the default connection URL in the application program. For example, jdbc:db2os390: or db2:default:connection.
- For type 4 JDBC drivers, you can use a specific connection URL in the application program.
- To commit or roll back work, you must use the IMSTransaction.getTransaction().commit() method or the IMSTransaction.getTransaction().rollback() method. For JMP applications, theIMSTransaction.getTransaction().commit() method commits all work: SQL calls and connection closures. For JBP applications, the IMSTransaction.getTransaction().commit() method commits SQL calls.
- Because RRS is the coordinator, you cannot use the Connection.setAutoCommit or Connection.commit method of the DB2 JDBC driver.
- You must always call IMSTransaction.getTransaction().commit() after closing a connection to DB2 UDB for z/OS to commit the connection closure.
- You cannot use COBOL to access DB2 UDB for z/OS in a JMP or JBP region.

Related Reading: For a JMP programming model, see "JMP Application that Accesses IMS or DB2 UDB for z/OS Data" on page 20. For a JBP programming model, see "JBP Application that Accesses DB2 UDB for z/OS or IMS Data" on page 28.
Program Switching in JMP and JBP Applications

IMS Java provides an API for immediate program switching in JMP and JBP applications and for deferred program switching in conversational JMP applications.

For more information about program switches, see the IMS Version 8: Application Programming: Design Guide.

Immediate Program Switching for JMP and JBP Applications

The setModifiableAlternatePCB(String) method of the com.ibm.ims.application.IMSMessageQueue class sets the name of the alternate PCB for the program switch. The setModifiableAlternatePCB(String) method calls the DL/I CHNG call.

To make a program switch in a JMP or JBP application:
1. Call the setModifiableAlternatePCB(String) method to set the name of the alternate PCB.
2. Call the insertMessage(IMSFieldMessage) method to send the message to the alternate PCB.

For more information about these methods, see the IMS Java API Specification.

Deferred Program Switching for Conversational JMP Applications

You can make a deferred program switch in a conversational JMP application. A deferred program switch changes the transaction code in the SPA before the SPA is returned to IMS. When an application makes a deferred program switch, the application replies to the terminal and passes the conversation to another conversational application.

The setTransactionID(String) method of the com.ibm.ims.application.IMSFieldMessage class specifies the transaction code in the SPA.

To make a deferred program switch:
1. Call the insertMessage(IMSFieldMessage) method to send the output message to the terminal.
2. Call the setTransactionID(String) method to set the name of the transaction code in the SPA.
3. Call the insertMessage(IMSFieldMessage) method to send the SPA to IMS.

For more information about these methods, see the IMS Java API Specification.
Chapter 3. WebSphere Application Server for z/OS Applications

You can write applications that run on WebSphere Application Server for z/OS and access IMS databases when WebSphere Application Server for z/OS and IMS are on the same LPAR (logical partition).

To deploy an application on WebSphere Application Server for z/OS, you must install the IMS JDBC resource adaptor (the IMS Java class libraries) on WebSphere Application Server for z/OS, and configure both IMS open database access (ODBA) and the database resource adapter (DRA).

Figure 14 shows an Enterprise JavaBean (EJB) that is accessing IMS data. JDBC or IMS Java hierarchical interface calls are passed to the IMS Java layer, which converts the calls to DL/I calls. The IMS Java layer passes these calls to ODBA, which uses the DRA to access the DL/I region in IMS.

Figure 14. WebSphere Application Server for z/OS EJB Using IMS Java

The following topics provide additional information:

- “System Requirements for WebSphere Application Server for z/OS” on page 36
- “Restrictions for WebSphere Application Server for z/OS” on page 36
- “Configuring WebSphere Application Server V5 for z/OS for IMS Java” on page 36
- “Running the IMS Java IVP on WebSphere Application Server V5 for z/OS” on page 39
- “Running the IMS Java Sample Application on WebSphere Application Server V5 for z/OS” on page 42
- “Running Your Applications on WebSphere Application Server for z/OS” on page 44
- “WebSphere Application Server V4.0.1 for z/OS Setup” on page 49
- “Developing Enterprise Applications that Access IMS DB” on page 54
System Requirements for WebSphere Application Server for z/OS

In addition to the software listed in "IMS Java System Requirements" on page 1, the following software is required:

• Either:
  – WebSphere Application Server V4.0.1 for z/OS and OS/390 with additional
    WebSphere Application Server z/OS Connection Management support.
  – WebSphere Application Server V5.0 for z/OS or later. If you have WebSphere
    Application Server V5.0.2 for z/OS, you must install either 5.0.2.1 or APAR
    PQ81944.

The following z/OS components are required:

• RRS (resource recovery services) for OS/390

The following IMS components are required:

• Open database access (ODBA)
• Database resource adapter (DRA)

Restrictions for WebSphere Application Server for z/OS

The following restrictions apply to WebSphere Application Server for z/OS EJBs that access IMS databases:

• IMS Java does not support container-managed signon or component-managed
  signon.
• IMS Java does not support shared connections.
• The java.sql.Connection object must be acquired, used, and closed within a
  transaction boundary.
• A global transaction must exist before you create a Connection object from a
  JDBC connection. Either specify container-demarcated transactions in the EJB
  deployment descriptor or explicitly begin a global transaction by calling
  the javax.transaction.UserTransaction API before creating a JDBC connection.

Configuring WebSphere Application Server V5 for z/OS for IMS Java

This section assumes that you are familiar with WebSphere Application Server V5 for z/OS and its administrative console.

Prerequisite: "Installing IMS Java" on page 1

To configure WebSphere Application Server V5 for z/OS:

1. "Configuring WebSphere Application Server for z/OS to Access IMS"
2. "Installing the IMS JDBC Resource Adapter" on page 37
3. "Installing the Custom Service" on page 38

Next: "Running the IMS Java IVP on WebSphere Application Server V5 for z/OS" on page 39

Configuring WebSphere Application Server for z/OS to Access IMS

To use JDBC to access IMS DB from WebSphere Application Server for z/OS, you first must configure WebSphere Application Server for z/OS to access IMS databases using ODBA. ODBA uses the database resource adapter (DRA) to access IMS databases.
To configure WebSphere Application Server for z/OS to access IMS databases:

1. If not already done, create a DRA startup table. The DRA startup table module name must have the following naming convention:
   - Bytes 1-3: “DFS”
   - Bytes 4-7: 1- to 4-byte ID
   - Byte 8: “0”

   Recommendation: The 1- to 4-byte ID should be the IMS system ID.

2. If not already done, link the DRA startup table into a load library.

3. Update the JCL for WebSphere Application Server for z/OS by adding to the STEPLIB the following data sets:
   - The load library that contains the DRA startup table and the ODBA run-time code
   - The SDFSJLIB data set. This data set contains the DFSCLIB member.

4. Note the DRA name, which is defined by the MBR parameter. You will need to know bytes 4-7, which are usually the IMS system ID, when you install the data source.

Next: “Installing the IMS JDBC Resource Adapter”

Installing the IMS JDBC Resource Adapter

After you configure WebSphere Application Server for z/OS to have access to IMS databases, you must install the IMS JDBC resource adapter on WebSphere Application Server for z/OS.

Prerequisite: “Configuring WebSphere Application Server for z/OS to Access IMS” on page 36

To install the IMS JDBC resource adapter:

1. From the WebSphere Application Server for z/OS administrative console, click Resources, and then click Resource Adapters.
   A list of resource adapters is displayed.

2. Click Install RAR.
   A dialog for installing the resource adapter is displayed.

3. Select Server path and type the path to the imsjava81.rar file: 
   pathprefix/usr/lpp/ims/imsjava81/imsjava81.rar

4. Click Next.
   A configuration dialog is displayed.

5. Type the following information:
   - Name: a name for the resource adapter
   - Classpath: the path to imsjava.jar, including the file name: 
     pathprefix/usr/lpp/ims/imsjava81/imsjava.jar

6. Click OK.
   The IMS JDBC resource adapter is listed.

7. Click Save.
   The Save page is displayed.
8. Under **Save to Master Configuration**, click **Save** to ensure that the changes have been made.

**Next: Installing the Custom Service**

**Installing the Custom Service**

**Prerequisite:** [“Installing the IMS JDBC Resource Adapter” on page 37](#)

When WebSphere Application Server for z/OS is started, the custom service initializes the ODBA environment. When the server is stopped, the custom service terminates the ODBA environment. After a server is started, every application that is running in the server uses the initialized ODBA environment.

To install the custom service:

1. Modify the WebSphere Application Server for z/OS server.policy file, which is in the properties directory of the WebSphere Application Server installation directory, by adding the following code:

   ```
   grant codeBase "file:/pathprefix/usr/lpp/ims/imsjava81/-" {
     //Allows the IMS JDBC resource adapter and the custom service to read and write environment properties
     permission java.util.PropertyPermission "*", "read, write";

     //Allows the IMS JDCB resource adapter and the custom service to use the JavTDLI load library during runtime.
     permission java.lang.RuntimePermission "loadLibrary.JavTDLI";
   };
   ```

2. In the left frame of the WebSphere Application Server for z/OS administrative console, click **Servers**, and then click **Application Servers**. A list of application servers is displayed.

3. Click the name of the server on which you want to deploy your enterprise application.

4. Under Additional Properties, click **Custom Services**. A list of custom services is displayed.

5. Click **New**. A configuration dialog is displayed.

6. Select the **Startup** check box. If you do not select the **Startup** check box, the custom service is not invoked when you start the server.

7. Type the following information:
   - **Classname**: com.ibm.connector2.ims.db.IMSSdbcCustomService
   - **Display Name**: a name for the custom service
   - **Classpath**: the path to the directory that contains imsjava.jar and libJavTDLI.so: `pathprefix/usr/lpp/ims/imsjava81`

8. Click **OK**. The custom service is listed.

9. Click **Save**. The Save page is displayed.

10. Under **Save to Master Configuration**, click **Save** to ensure that the changes have been made.

11. Restart the server in order for the custom service to take effect.
Running the IMS Java IVP on WebSphere Application Server V5 for z/OS

Prerequisites:

- “Configuring WebSphere Application Server V5 for z/OS for IMS Java” on page 36
- Ensure that the standard IMS IVPs have been run. The IMS IVPs prepare the DBD for the IVP database, named IVPDB2, and load the IVP database. They also prepare the IMS Java application PSB (named DFSIVP37), build ACBs, and prepare other IMS control blocks that are required by the IMS Java IVPs. For details about how to run the IMS IVPs, see IMS Version 8: Installation Volume 1: Installation Verification.
- Ensure that the samples.tar file has been expanded. This topic assumes that its contents are in the directory pathprefix/usr/lpp/ims/imsjava81. For more information on expanding the samples.tar file, see “Installing IMS Java” on page 1.

To run the IMS Java IVP for WebSphere Application Server for z/OS:

1. “Installing the Data Source for the IMS Java IVP”
2. “Installing the IMS Java IVP” on page 40
3. “Testing the IMS Java IVP” on page 41

Next: “Running the IMS Java Sample Application on WebSphere Application Server V5 for z/OS” on page 42 or “Running Your Applications on WebSphere Application Server for z/OS” on page 44.

Installing the Data Source for the IMS Java IVP

The DataSource facility is a factory for connections to a physical data source, or database. A data source is registered with a naming service based on the Java Naming and Directory (JNDI) API. DataSource objects have properties that pertain to the actual data source that an application needs to access.

Requirement: You must use the DataSource facility, which replaces the DriverManager facility, because the DriverManager facility is not supported by the J2EE Connection Architecture Specification.

To install the data source for the IVP:

1. In the left frame of the WebSphere Application Server for z/OS administrative console, click Resources, and then click Resource Adapters. A list of resource adapters is displayed.
2. Click the name of the IMS JDBC resource adapter that you chose when you installed the adapter. A configuration dialog is displayed.
4. Click New. A configuration dialog is displayed.
5. Type the following information:
   - **Name**: name for the data source
   - **JNDI Name**: jdbc/IMSIVP

6. Click OK.
   The data source is listed in the J2C Connection Factories.

7. Click the name of the data source that you installed in step 5.

   Six properties are listed in a table.

9. In the **DRAName** row, click the dash symbol in the **Value** column.
10. In the **Value** field, type bytes 4-7 of the DRA startup table module name
     usually the IMS system ID. For more information about the DRA startup table,
     see "Configuring WebSphere Application Server for z/OS to Access IMS" on
     page 36.

11. Click OK.
   The properties table displays the DRA name that you just entered.

12. In the **DatabaseViewName** row, click the dash symbol in the **Value** column.
13. In the **Value** field, type samples.ivp.DFSIVP37DatabaseView

14. Click OK.
   The properties table displays the host name that you just entered.

15. Click Save.
   The Save page is displayed.

16. Under **Save to Master Configuration**, click Save.

17. Restart the server to ensure that the changes have been made.

Next: "Installing the IMS Java IVP"

### Installing the IMS Java IVP

**Prerequisite:** "Installing the Data Source for the IMS Java IVP" on page 39

This section describes how to deploy the IMS Java IVP on WebSphere Application Server for z/OS.

To install the IMS Java IVP:
1. From the WebSphere Application Server for z/OS administrative console, click Applications, and then click Install New Application.
   A dialog for installing the application is displayed.

2. Select **Server path** and type the path to IMSJavaIVP.ear:
   `pathprefix/usr/lpp/ims/imsjava81/samples/ivp/was/imsjavaIVP.ear`

3. Click Next.

4. Accept the defaults and click Next.
   The Install New Application wizard is started. Step 1, "Provide options to perform the installation," is displayed.

5. Clear the **Create MBeans for Resources** check box.

6. Click Next.
   Step 2, "Provide JNDI Names for Beans," is displayed.

7. In the **JNDI Name** field, type: `samples.ivp.was.WASIVPSessionHome`

8. Click Next.
   Step 3, "Map resource references to resources," is displayed.
9. In the **JNDI Name** field, type: jdbc/IMSIVP
10. Click **Next**.
   Accept the defaults and click **Next**.
12. Step 5, “Map modules to application servers,” is displayed.
   Accept the defaults and click **Next**.
   Step 6, “Ensure all unprotected 2.0 methods have the correct level of protection,” is displayed.
13. Make any necessary changes and click **Next**.
   The options that you specified are displayed in Step 7, “Summary,” of the Install New Application wizard.
14. Verify that the options are correct, and then click **Finish**.
   A message is displayed that indicates first that the application is being installed, and then that the installation was successful.
15. Click **Save to Master Configuration**.
   The Save page is displayed.
16. Under **Save to Master Configuration**, click **Save**.
17. Restart the server to ensure that the changes have been made.

**Next:** "Testing the IMS Java IVP"

### Testing the IMS Java IVP

**Prerequisite:** "Installing the IMS Java IVP" on page 40

This topic describes how to test the IVP on WebSphere Application Server for z/OS.

To test the IMS Java IVP:

1. From the WebSphere Application Server for z/OS administrative console, click **Applications**, and then click **Enterprise Applications**.
   The application IMSJava IVP is listed with a red X, which indicates that the application is stopped.
2. Select **IMSJava IVP**.
3. Click **Start**.
   The application IMSJava IVP is listed with a green arrow, which indicates that the application is started.
4. Open a Web browser.
5. Type the Web address:
   http://host_IP_address:port/IMSjdbcIVPWeb/WASIVP.html
   An input Web page opens.
6. In the **Last Name** field, type: LAST1
   If WebSphere Application Server for z/OS is configured properly, the following output is displayed:
   Last Name: LAST1 First Name: FIRST1 Extension: 8-111-1111 Zip code: D01/R01
Running the IMS Java Sample Application on WebSphere Application Server V5 for z/OS

Prerequisite: “Running the IMS Java IVP on WebSphere Application Server V5 for z/OS” on page 39

To run the IMS Java sample application on WebSphere Application Server for z/OS:
1. “Installing the Data Source for the IMS Java Samples”
2. “Installing the IMS Java Sample Application” on page 43
3. “Testing the IMS Java Sample Application” on page 44

Next: “Running Your Applications on WebSphere Application Server for z/OS” on page 44

Installing the Data Source for the IMS Java Samples

The DataSource facility is a factory for connections to a physical data source, or database. A data source is registered with a naming service based on the Java Naming and Directory (JNDI) API. DataSource objects have properties that pertain to the actual data source that an application needs to access.

Requirement: You must use the DataSource facility, which replaces the DriverManager facility, because the DriverManager facility is not supported by the J2EE Connection Architecture Specification.

To install the data source for the IMS Java samples:
1. In the left frame of the WebSphere Application Server for z/OS administrative console, click Resources, and then click Resource Adapters.
   A list of resource adapters is displayed.
2. Click the name of IMS JDBC resource adapter that you chose when you installed the adapter.
   A configuration dialog is displayed.
4. Click New.
   A configuration dialog is displayed.
5. Type the following information:
   Name: name for the data source
   JNDI Name: jdbc/DealershipSample
6. Click OK.
   The data source is listed in the J2C Connection Factories.
7. Click the name of the data source that you installed in step 5
   Six properties are listed in a table.
9. In the DRAName row, click the dash symbol in the Value column.
10. In the Value field, type bytes 4-7 of the DRA startup table module name (usually the IMS system ID). For more information about the DRA startup table, see “Configuring WebSphere Application Server for z/OS to Access IMS” on page 36.
11. Click OK.
   The properties table displays the DRA name that you just entered.
12. In the **DatabaseViewName** row, click the dash symbol in the **Value** column.
13. In the **Value** field, type: `samples.dealership.AUTPSB11DatabaseView`
14. Click **OK**.
   The properties table displays the host name that you just entered.
15. Optionally, set the trace level for the applications.
16. Click **Save**.
   The Save page is displayed.
17. Under **Save to Master Configuration**, click **Save**.

**Next:** "Installing the IMS Java Sample Application"

### Installing the IMS Java Sample Application

**Prerequisite:** "Installing the Data Source for the IMS Java Samples" on page 42

This topic describes how to install the IMS Java sample application on WebSphere Application Server for z/OS.

To install the sample applications:

1. From the WebSphere Application Server for z/OS administrative console, click **Applications**, and then click **Install New Application**.
   A dialog for installing a new application is displayed.
2. Select **Server path** and type:
   `pathprefix/usr/lpp/ims/imsjava81/samples/dealership/was/imsjavaDealershipEAR.ear`
3. Click **Next**.
4. Accept the defaults and click **Next**.
   The Install New Application wizard is started. Step 1, “Provide options to perform the installation,” is displayed.
5. Clear the **Create MBeans for Resources** check box.
6. Click **Next**.
   Step 2, “Provide JNDI Names for Beans,” is displayed.
7. In the **JNDI Name** field, type: `samples.dealership.was.DealershipSessionHome`
8. Click **Next**.
   Step 3, “Map resource references to resources,” is displayed.
9. In the **JNDI Name** field for the IMSDealershipWeb module, type: `jdbc/DealershipSample`
10. Click **Next**.
    Step 4, “Map virtual hosts for web modules,” is displayed.
11. Accept the defaults and click **Next**.
    Step 5, “Map modules to application servers,” is displayed.
12. Accept the defaults and click **Next**.
    Step 6, “Ensure all unprotected 2.0 methods have the correct level of protection,” is displayed.
13. Make any necessary changes and click **Next**.
    The options that you specified are displayed in Step 7, “Summary,” of the Install New Application wizard.
14. Verify that the options are correct, and then click **Finish**.
A message is displayed that indicates first that the application is being installed, and then that the installation was successful.

15. Click **Save to Master Configuration**.

The Save page is displayed.

16. Under **Save to Master Configuration**, click **Save**.

17. Restart the server to ensure that the changes have been made.

Next: [“Testing the IMS Java Sample Application”](#)

### Testing the IMS Java Sample Application

**Prerequisite:** [“Installing the IMS Java Sample Application” on page 43](#)

This section describes how to test the dealership sample application on WebSphere Application Server for z/OS.

To test the dealership sample:
1. From the WebSphere Application Server for z/OS administrative console, click **Applications**, and then click **Enterprise Applications**.
   - The application that you installed is listed with a red X, which indicates that the application is stopped.
2. Select **IMSDealershipEAR**.
3. Click **Start**.
   - The application is listed with a green arrow, which indicates that the application is started.
4. Open a Web browser.
5. Type the Web address of the application:
   - `http://host_IP_address:port/IMSDealershipWeb/dealership.html`
   - An input Web page opens that is titled **Find a car in stock**.
6. Verify that **Car Make** and **VIN Number** fields contain the following information:
   - **Car Make**: FORD
   - **VIN Number**: V234567890123456789V
7. Click **Submit**.
   - If WebSphere Application Server for z/OS is configured properly, the output is displayed. A message indicating that the query was successful is displayed.

### Running Your Applications on WebSphere Application Server for z/OS

**Prerequisite:** [“Running the IMS Java IVP on WebSphere Application Server V5 for z/OS” on page 39](#)

To deploy your applications on WebSphere Application Server for z/OS:
1. [“Setting the WebSphere Application Server for z/OS Classpath” on page 45](#)
2. [“Installing the Data Source for Your Application” on page 45](#)
3. [“Installing Your Application” on page 46](#)
4. [“Enabling J2EE Tracing with WebSphere Application Server for z/OS” on page 47](#)
Setting the WebSphere Application Server for z/OS Classpath

Your application can include the IMS Java metadata class (DLIDatabaseView subclass) or the metadata class can be stored elsewhere.

If your application does not include the metadata class, you must set the WebSphere Application Server for z/OS classpath to the location of the IMS Java metadata class that is used by the application.

One way to set the classpath is to add these files to the IMS JDBC resource adapter classpath.

To add the required files to the IMS JDBC resource adapter classpath:
1. From the WebSphere Application Server for z/OS administrative console, click Resources, and then click Resource Adapters.
   A list of resource adapters is displayed.
2. Click the name of the IMS JDBC resource adapter.
   A configuration dialog is displayed.
3. In the Classpath field, add the path to the required files. Include the file name for JAR files. Do not delete imsjava.jar.
4. Click OK.

Installing the Data Source for Your Application

The DataSource facility is a factory for connections to a physical data source, or database. A data source is registered with a naming service based on the Java Naming and Directory (JNDI) API. DataSource objects have properties that pertain to the actual data source that an application needs to access.

Requirement: You must use the DataSource facility, which replaces the DriverManager facility, because the DriverManager facility is not supported by the J2EE Connection Architecture Specification.

To install the data source for your application:
1. In the left frame of the WebSphere Application Server for z/OS administrative console, click Resources, and then click Resource Adapters.
   A list of resource adapters is displayed.
2. Click the name of IMS JDBC resource adapter that you chose when you installed the adapter.
   A configuration dialog is displayed.
4. Click New.
   A configuration dialog is displayed.
5. Type the following information:
   Name: name for the data source
   JNDI Name: path to the data source.
6. Click OK.
   The data source is listed in the J2C Connection Factories.
7. Click the name of the data source that you installed in step 5.
   Six properties are listed in a table.
9. In the DRAName row, click the dash symbol in the Value column.
10. In the Value field, type bytes 4-7 of the DRA startup table module name (usually the IMS system ID). For more information about the DRA startup table, see “Configuring WebSphere Application Server for z/OS to Access IMS” on page 36.

11. Click OK.
   The properties table displays the DRA name that you just entered.

12. In the DatabaseViewName row, click the dash symbol in the Value column.

13. Optional: In the Value field, type the fully-qualified DLIDatabaseView subclass name.
    If you do set the subclass name, you must either create a data source for every PSB an application accesses, or you must override the DLIDatabaseView subclass name in the DataSource object by calling the setDatabaseView method and providing the fully-qualified name of the subclass.
    If you do not set the subclass name, you need to create a data source only for each IMS. In the application, define the DLIDatabaseView subclass name in the DataSource object by calling the setDatabaseView method and providing the fully-qualified name of the subclass.

14. Click OK.
   The properties table displays the host name that you just entered.

15. Optionally, set the trace level for the applications. See “Enabling J2EE Tracing with WebSphere Application Server for z/OS” on page 47.

16. Click Save.
   The Save page is displayed.

17. Under Save to Master Configuration, click Save.

18. Restart the server to ensure that the changes have been made.

Next: “Installing Your Application”

### Installing Your Application

**Prerequisite:** “Installing the Data Source for Your Application” on page 45

This section describes how to deploy an application on WebSphere Application Server for z/OS.

To install your application:

1. From the WebSphere Application Server for z/OS administrative console, click Applications, and then click Install New Application.
   A dialog for installing a new application is displayed.

2. Type the path to the EAR file.

3. Click Next.

4. Accept the defaults and click Next.
   The Install New Application wizard is started. Step 1, “Provide options to perform the installation,” is displayed.

5. Clear the Create MBeans for Resources check box.

6. Click Next.
   Step 2, “Provide JNDI Names for Beans,” is displayed.

7. In the JNDI Name field, type the path to the EJB home interface.

8. Click Next.
Step 3, "Map resource references to resources," is displayed.

9. Type the JNDI name for the data source that you created in "Installing the Data Source for Your Application" on page 45.

10. Click Next.

Step 4, "Map modules to application servers," is displayed.

11. Accept the defaults and click Next.

Step 5, "Correct use of System Identity," is displayed.

12. Verify that no role is selected and click Next.

Step 6, "Ensure all unprotected 2.0 methods have the correct level of protection," is displayed.

13. Make any necessary changes and click Next.

The options that you specified are displayed in Step 7, "Summary," of the Install New Application wizard.

14. Verify that the options are correct, and then click Finish.

A message is displayed that indicates first that the application is being installed, and then that the installation was successful.

15. Click Save to Master Configuration.

The Save page is displayed.

16. Under Save to Master Configuration, click Save.

17. Restart the server to ensure that the changes have been made.

### Enabling J2EE Tracing with WebSphere Application Server for z/OS

You can trace the IMS library classes by using the WebSphere Application Server for z/OS tracing service.

To enable tracing if you have not yet specified the level of tracing:

1. "Specifying the Level of Tracing"
2. "Specifying the Application Server and the Package to Trace" on page 48

To enable tracing if you have already specified the level of tracing:

1. "Specifying at Runtime the Application Server and the Package to Trace" on page 48

You can also trace the IMS library classes or your applications using the com.ibm.ims.base.XMLTrace class. The XMLTrace class is an IMS Java-provided class that represents the trace as an XML document. You can trace different levels of the code depending on the trace level. For more information, see the IMS Java API Specification.

### Specifying the Level of Tracing

To use the WebSphere Application Server for z/OS tracing service, you must first specify the level of tracing.

To specify the level of tracing:

1. In the left frame of the WebSphere Application Server for z/OS administrative console, click Resources, and then click Resource Adapters.
   
   A list of resource adapters is displayed.

2. Click IMS JDBC resource adapter.

   A configuration dialog is displayed.

A list of connection factories is displayed.

4. Click the name of the J2C connection factory for which you want to enable tracing.
   A configuration dialog is displayed.

   Properties are listed in a table.

6. In the Trace Level row, click the number in the Value column.

7. In the Value field, type the trace level.

8. Click OK.
   The properties table displays the trace level that you just entered.

9. Click Save.
   The Save page is displayed.

10. Under Save to Master Configuration, click Save to ensure that the changes are made.

**Specifying the Application Server and the Package to Trace**

After you specify the level of tracing, specify the application server and package to trace and then restart the server.

To specify the application server and the package to trace:

1. In the left frame of the WebSphere Application Server for z/OS administrative console, click Servers, and then click Application Servers.
   A list of application servers is displayed.

2. Click the name of the server on which you want to enable tracing.

   A configuration dialog for Diagnostic Trace Service is displayed.

4. Select the Enable Trace check box.

5. In the Trace Specification field after any other traces that are listed, type:
   com.ibm.connector2.ims.db.*=all=enabled

6. Click Apply.

7. Click Save.
   The Save page is displayed.

8. Under Save to Master Configuration, click Save to ensure that the changes are made.

9. Restart the server.

**Specifying at Runtime the Application Server and the Package to Trace**

You can turn tracing on and off by specifying at runtime the server and package to trace. You do not need to restart your server each time.

To specify the application server and the package to trace at runtime:

1. In the left frame of the WebSphere Application Server for z/OS administrative console, click Servers, and then click Application Servers.
   A list of application servers is displayed.

2. Click the name of the server on which you want to enable tracing.

   A configuration dialog for Diagnostic Trace Service is displayed.

4. Click the Runtime tab.
5. In the **Trace Specification** field after any other traces that are listed, type:
   
   ```
   com.ibm.connector2.ims.db.*=all=enabled
   ```

6. Click **Apply**.

---

### WebSphere Application Server V4.0.1 for z/OS Setup

This section assumes you are familiar with IBM WebSphere Application Server V4.0.1 for z/OS and OS/390, its Administration application and its Application Assembly tool.

**Related Reading:**

- For detailed information about how to use the Administration application, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management User Interface*.
- For detailed information about assembling and deploying an EJB onto a J2EE server, see *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications*.

---

### WebSphere Application Server V4.0.1 for z/OS Configuration

To deploy an EJB on a WebSphere Application Server V4.0.1 for z/OS J2EE server, you must perform the following steps:

1. Configure the WebSphere Application Server for z/OS J2EE server to access IMS databases.
2. Configure the WebSphere Application Server for z/OS J2EE server to locate the IMS JDBC Resource Adapter.
3. Deploy an instance of the IMS JDBC Resource Adapter.
4. Deploy an Enterprise Archive (EAR) that contains an Enterprise Java Bean (EJB).

### Configuring the WebSphere Application Server for z/OS J2EE Server for IMS Access

To run an EJB on a WebSphere Application Server for z/OS J2EE server, you must configure WebSphere Application Server for z/OS to access IMS databases using ODBA. ODBA uses the database resource adapter (DRA) to access IMS databases. More details about the steps in this section are in *IMS Version 8: Installation Volume 2: System Definition and Tailoring*.

To configure WebSphere Application Server for z/OS to access IMS databases:

1. If not already done, create a DRA startup table. The DRA startup table module name must have the following naming convention:
   - Bytes 1-3: “DFS”
   - Bytes 4-7: 1- to 4-byte ID
   - Byte 8: “0”

   **Recommendation:** The 1- to 4-byte ID should be the IMS system ID.

2. Link the DRA startup table into a load library.

3. Update the JCL for WebSphere Application Server for z/OS by adding to the STEPLIB the following:
   - The load library that contains the DRA startup table and the ODBA run-time code
   - The data set that contains the DFSCLIB member.
4. Note the DRA name (defined by the MBR parameter). Bytes 4-7 (usually the IMS system ID) are needed when installing the data source.

Configuring the WebSphere Application Server for z/OS J2EE Server to Locate the IMS JDBC Resource Adapter

You must configure the WebSphere Application Server for z/OS J2EE server for the IMS JDBC resource adapter. The procedure is described in this section.

The WebSphere Application Server for z/OS and OS/390 Administration application groups a series of changes into a *conversation*. All the changes described in a conversation are validity checked and activated (installed) in the server at the same time. Activating a conversation sets the J2EE server properties for the IMS JDBC Resource Adapter. Expanding the custom service file and entering its directory in the file jvm.properties allows WebSphere Application Server for z/OS to initialize and terminate ODBA.

To configure the WebSphere Application Server for z/OS J2EE server to locate the IMS JDBC Resource Adapter:

1. Create a conversation and define the IMS JDBC Resource Adapter as a J2EE resource:
   a. Open the WebSphere Application Server for z/OS and OS/390 Administration application.
   b. Add a conversation.
   c. Navigate to the sysplex that will run the EJBs.
   d. Modify the sysplex properties by selecting Connection Management.
   e. Modify the J2EE properties by defining the CLASSPATH and LIBPATH environment variables:
      • For the CLASSPATH environment variable, type the full name of the directory that contains the file imsjava.jar, including the file name. If you use the default IMS installation directory, type:
        `/usr/lpp/ims/imsjava81/imsjava.jar`
      • For the LIBPATH environment variable, enter the full name of where IMS Java is installed. If you use the default IMS installation directory, type:
        `/usr/lpp/ims/imsjava81`
   f. Save and activate the conversation.

2. In the installation directory, expand the file imsjava81.rar. If you used the default directory to install IMS, the file is in /usr/lpp/ims/imsjava81. To expand the file, type the following command:
   `jar -xvf imsjava81.rar`
   The expansion puts the following files (in UTF-8 format) in the same directory as the imsjava81.rar file:
   • IMSJdbcCustomService.xml
   • howto.html
   **Related Reading:** For more information about the IMS JDBC resource adapter configuration, see the howto.html file. Because it is encoded in UTF-8 format, you cannot read it in the OS/390 environment. To read the file, expand imsjava81.rar on your desktop.

3. Edit the jvm.properties file for WebSphere J2EE server regions that will access IMS databases to identify the location of the file IMSJdbcCustomService.xml.
To edit the jvm.properties file, add the directory where the file IMSJdbcCustomService.xml is to the preconfigured custom services in the jvm.properties file.

If you use the default IMS installation directory, enter:
com.ibm.websphere.preconfiguredCustomServices=/usr/lpp/ims/imsjava81/IMSJdbcCustomService.xml

**Important:** If you do not edit the jvm.properties file correctly, PSB allocation fails when using the IMS JDBC Resource Adapter.

### Deploying an Instance of the IMS JDBC Resource Adapter

Configuring and deploying an instance of the IMS JDBC resource adapter creates a data source for an EJB. Deploy a new instance for each database or database resource adapter (DRA) startup table an EJB accesses. See “Two Strategies for Deploying Instances of IMS JDBC Resource Adapter” for more information on when to deploy a new instance.

**Note:** If you are setting up WebSphere Application Server for z/OS for the first time and want to verify your installation, use the procedure in “WebSphere Application Server V4.0.1 for z/OS Installation Verification” on page 52.

To deploy an instance of the IMS JDBC resource adapter:

1. Open the WebSphere Application Server for z/OS and OS/390 Administration application.
2. Add a conversation.
3. Navigate to the sysplex that will run the EJBs.
   - The following folders are displayed:
     J2EEServers
     Servers
     Systems
     J2EE Resources
     Logical Resource Mapping
4. Add a J2EE resource. The resource name is your choice. The type is IMSJdbcDataSource.
5. Add a J2EE resource instance. The name is your choice.
6. Enter the DRA startup table name and optionally the DLIDatabaseView subclass name. To decide whether to add a DLIDatabaseView subclass name, see “Two Strategies for Deploying Instances of IMS JDBC Resource Adapter.”
7. Save and activate the conversation.

**Note:** These instructions use two conversations to deploy the IMS JDBC Resource Adapter and the Enterprise Archive (EAR). However, you can also deploy an EAR in the same conversation that deploys the IMS JDBC Resource Adapter instance.

**Two Strategies for Deploying Instances of IMS JDBC Resource Adapter:**

When configuring an instance of the IMS JDBC resource adapter, you can optionally set the DLIDatabaseView subclass name.

If you do set the subclass name, you must either create a new instance of the IMS JDBC resource adapter for every PSB an EJB accesses, or you must override the DLIDatabaseView subclass name (set in step 6) in the DataSource object by calling the setDatabaseView method and providing the fully-qualified name of the subclass.
If you do not set the subclass name, you only need to deploy an instance of the IMS JDBC resource adapter for each DRA startup table. In the EJB, define the DLIDatabaseView subclass name (set in step 6 on page 51) in the DataSource object by calling the setDatabaseView method and providing the fully-qualified name of the subclass.

Deploying an EJB onto a WebSphere Application Server for z/OS J2EE Server

After you develop the EJB, deploy it on the WebSphere Application Server for z/OS J2EE server.

To deploy your application onto the WebSphere Application Server for z/OS J2EE server:

1. Package the EJB into an Enterprise Archive (EAR) using a development tool such as WebSphere Studio Application Developer Integrated Edition.
2. Import the EAR into WebSphere for z/OS and OS/390 Application Assembly tool.
3. Create a resolved EAR suitable for deploying on a WebSphere Application Server for z/OS J2EE server.
4. Open the WebSphere Application Server for z/OS and OS/390 Administration application.
5. Add a conversation.
6. Navigate to the J2EEServers folder of the sysplex that will run the EJB.
   The following folders are displayed:
   J2EEServers
   Servers
   Systems
   J2EE Resources
   Logical Resource Mapping
7. Expand the J2EEServers folder and choose the server to install the EJB on.
8. Install the EJB.
   a. Specify the fully-qualified directory name of the EAR and the FTP server of the sysplex where the EJB will run.
   b. Set the JNDI name and path.
   c. Associate the J2EE resource defined in “Deploying an Instance of the IMS JDBC Resource Adapter” on page 51.
9. Save and activate the conversation.

WebSphere Application Server V4.0.1 for z/OS Installation Verification

After you have configured WebSphere Application Server for z/OS to access IMS (“Configuring the WebSphere Application Server for z/OS J2EE Server for IMS Access” on page 49) and to locate the IMS JDBC Resource Adapter (“Configuring the WebSphere Application Server for z/OS J2EE Server to Locate the IMS JDBC Resource Adapter” on page 50), you can verify that your systems are configured correctly by running the installation verification program (IVP).

Note: You can also use this procedure to install the dealership sample EAR, which is also in samples.tar. The sample file name is imsjavaDealership.ear and the DLIDatabaseView subclass name is samples.dealership.AUTPSB11DatabaseView.

1. Ensure that the standard IMS IVPs have been run.
   These IVPs prepare the DBD for the IVP database, named IVPDB2, and load
the IVP database. They also prepare the IMS Java application PSB (named DFSIVP37), build ACBs, and prepare other IMS control blocks required by the IMS Java IVPs. For details of how to run the IMS IVP procedures, see IMS Version 8: Installation Volume 1: Installation Verification.

2. Ensure that the samples.tar file has been expanded. This topic assumes that its contents are in the directory pathprefix/usr/lpp/ims/imsjava81. For more information on expanding the samples.tar file, see “Installing IMS Java” on page 1.

3. Transfer the IVP EAR file in binary mode to your workstation:
   a. Expand the samples.tar file. For instructions, see the Readme file in the samples directory.
   b. Use FTP to transfer the file imsjavaIVP.ear to your workstation.

4. Deploy an instance of the IMS JDBC Resource Adapter:
   a. Open the WebSphere Application Server for z/OS and OS/390 Administration tool.
   b. Add a conversation.
   c. Navigate to the sysplex that will run the EJBs.
      The following folders are displayed:
      J2EEServers
      Servers
      Systems
      J2EE Resources
      Logical Resource Mapping
   d. Add a J2EE resource. The resource name is your choice. The type is IMSJdbcDataSource.
   e. Add a J2EE resource instance with the following information:
      • J2EE Resource Instance Name: your choice, such as IMSJavaIVPDataSource
      • System Name: name of system that will run the server
      • DLIDatabaseView subclass name: samples.ivp.DFSIVP37DatabaseView
      • DRA Startup Table: name of your DRA table
   f. Save and activate the conversation.

5. Assemble the IVP EAR:
   a. Import the EAR into WebSphere for z/OS and OS/390 Application Assembly tool.
   b. Create a resolved EAR suitable for deploying on a WebSphere Application Server for z/OS J2EE server.

6. Deploy the IVP EAR, which contains the IVP JAR file and the IVP Web Archive (WAR):
   a. Open the WebSphere Application Server for z/OS and OS/390 Administration application.
   b. Add a conversation.
   c. Navigate to the J2EEServers folder of the sysplex that will run the EJB.
      The following folders are displayed:
      J2EEServers
      Servers
      Systems
      J2EE Resources
      Logical Resource Mapping
   d. Expand the J2EEServers folder and choose the server to install the EJB on.
WebSphere Application Server for z/OS Setup

e. Install the EJB with the following information:
   - EAR file name: imsjavaIVP.ear
   - System name: FTP server for the sysplex that the IVP EJB will run on
   - JNDI Path: Clear the text field
   - JNDI Name: samples.ivp.was.WASIVPSessionHome
   - Associate the J2EE resource defined in step [4 on page 53

f. Save and activate the conversation.

7. Update the HTTP server to access the IVP Web application:
   a. Update the file webcontainer.conf to contain the context root specification for
      the IVP:
      host.default_host.contextroot=/,IMSJdbcIVPWeb,/IMSJdbcIVPWeb/*
   b. Update the file httpd.conf to contain the service entry for IVP:
      Service /IMSJdbcIVPWeb/*

8. Run the IVP to verify the installation:
   a. Open a Web browser.
   b. Enter the following Web address, prefaced by the specific host IP address:
      http://host_IP_address:port/IMSJdbcIVPWeb/WASIVP.html

Developing Enterprise Applications that Access IMS DB

Enterprise applications that access IMS DB can be servlets or EJBs. The EJBs can
be bean-managed or container-managed. This topic describes the programming
models for these different types of enterprise applications. These programming
models apply to enterprise applications that run on WebSphere Application Server
for z/OS.

In this topic:
- "Bean-Managed EJBs"
- "Container-Managed EJBs" on page 55
- "Servlets" on page 56
- "Deployment Descriptor Requirements for IMS Java" on page 56

Bean-Managed EJBs

In bean-managed EJBs, you programmatically define the transaction boundaries. To
define an EJB as bean-managed, set the transaction-type property, which is in the
ejb-jar.xml file of the EJB jar file, to Bean. You must manage the scope of the
transaction by using the javax.transaction.UserTransaction interface. This topic
describes how to use this interface.

The programming model applies to applications that run on WebSphere Application
Server for z/OS. With the javax.transaction.UserTransaction interface, you can
define when the scope of the transaction begins and ends, and when the
transaction commits or rolls back. The EJB container supplies the EJB with a
javax.ejb.SessionContext object that allows the
javax.transaction.UserTransaction interface to perform the required operations to
manage the transaction.

```java
try {
    // Use the javax.ejb.SessionContext set by the EJB container to instantiate
    // a new UserTransaction
    javax.transaction.UserTransaction userTransaction =
        sessionContext.getUserTransaction();
```
// Begin the scope of this transaction
userTransaction.begin();

// Perform JNDI lookup to obtain the data source (the IVP datasource for
// example) and cast
javax.sql.DataSource dataSource = (javax.sql.DataSource)
    initialContext.lookup("java:comp/env/jdbc/IMSIVP");

// Get a connection to the data source
java.sql.Connection connection = dataSource.getConnection();

// Create an SQL statement using the connection
java.sql.Statement statement = connection.createStatement();

// Acquire a result set by executing the query using the statement
java.sql.ResultSet results = statement.executeQuery(...);

// Commit and complete the scope of this transaction
userTransaction.commit();

// Close the connection
connection.close();

} catch (Throwable t) {

    // If an exception occurs, roll back the transaction
    userTransaction.rollback();

    // Close the connection
    connection.close();

}

---

**Container-Managed EJBs**

In container-managed EJBs, the container manages the transaction demarcation. The demarcation is defined in the ejb-jar.xml file of the EJB. To define an EJB as container-managed, set the transaction-type property, which is in the ejb-jar.xml file of the EJB jar file, to Container. Because the container manages the transaction demarcation, this programming model does not have any transaction logic.

try {

    // Perform JNDI lookup to obtain the data source (the IVP data source
    // for example) and cast
    javax.sql.DataSource dataSource = (javax.sql.DataSource)
        initialContext.lookup("java:comp/env/jdbc/IMSIVP");

    // Get a connection to the data source
    java.sql.Connection connection = dataSource.getConnection();

    // Create an SQL statement using the connection
    java.sql.Statement statement = connection.createStatement();

    // Acquire a result set by executing the query using the statement
    java.sql.ResultSet results = statement.executeQuery(...);

    // Close the connection
    connection.close();

} catch (Throwable t) {

    // Close the connection
    connection.close();

}
Servlets

Similarly to the bean-managed EJBs, the servlet programming model uses the UserTransaction interface to begin, commit, or rollback the transaction. Because the servlet resides outside of the EJB container and cannot use an EJBContext object, the initial context requires an additional JNDI lookup to locate and instantiate the UserTransaction interface.

```java
try {
    // Establish an initial context to manage the environment properties and JNDI names
    javax.naming.InitialContext initialContext = new InitialContext();

    // Locate and instantiate a UserTransaction object that is associated with
    // the initial context using JNDI
    javax.transaction.UserTransaction userTransaction = (UserTransaction)
        initialContext.lookup("java:comp/UserTransaction");

    // Begin the scope of this transaction
    userTransaction.begin();

    // Perform JNDI lookup to obtain the data source (the IVP data source
    // for example) and cast
    javax.sql.DataSource dataSource = (javax.sql.DataSource)
        initialContext.lookup("java:comp/env/jdbc/IMSIVP");

    // Get a connection to the datasource
    java.sql.Connection connection = dataSource.getConnection();

    // Create an SQL statement using the connection
    java.sql.Statement statement = connection.createStatement();

    // Acquire a result set by executing the query using the statement
    java.sql.ResultSet results = statement.executeQuery(...);

    // Commit and complete the scope of this transaction
    userTransaction.commit();

    // Close the connection
    connection.close();
} catch (Throwable t) {
    // If an exception occurs, roll back the transaction
    userTransaction.rollback();

    // Close the connection
    connection.close();
}
```

Deployment Descriptor Requirements for IMS Java

The deployment descriptor for an EJB or servlet has certain requirements for IMS Java. In an EJB, the deployment descriptor is the file ejb-jar.xml. In a servlet, the deployment descriptor is the file web.xml.

You must have a resource-ref element in the deployment descriptor. The resource-ref element describes external resources. In the resource-ref element, you must have the following elements:

- `<res-type>javax.sql.DataSource</res-type>`
- `<res-sharing-scope>Unshareable</res-sharing-scope>`

The `<res-type>javax.sql.DataSource</res-type>` element specifies the type of data source. The `<res-sharing-scope>Unshareable</res-sharing-scope>` element specifies that the connections are not shareable.
The following example is a resource-ref element from an EJB deployment descriptor:

```
<resource-ref>
  <res-ref-name>jdbc/DealershipSample</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Unshareable</res-sharing-scope>
</resource-ref>
```
Chapter 4. DB2 UDB for z/OS Stored Procedures

You can write DB2 UDB for z/OS Java stored procedures that access IMS databases.

To deploy a Java stored procedure on DB2 UDB for z/OS, you must configure IMS Java, ODBA, and DRA.

Figure 15 shows a DB2 UDB for z/OS stored procedure using IMS Java, ODBA, and DRA to access IMS databases.

The following topics provide additional information:
- “Configuring DB2 UDB for z/OS for IMS Java”
- “Running the IMS Java IVP from DB2 UDB for z/OS” on page 61
- “Running the IMS Java Sample Application on DB2 UDB for z/OS” on page 63
- “Running Your Stored Procedure from DB2 UDB for z/OS” on page 64
- “Developing DB2 UDB for z/OS Stored Procedures that Access IMS DB” on page 66

Configuring DB2 UDB for z/OS for IMS Java

Access to IMS databases from DB2 UDB for z/OS stored procedures requires IBM DB2 Universal Database for z/OS and OS/390 Version 7 with APARs PQ46673 and PQ50443. You also must have the DB2 for OS/390 and z/OS SQLJ/JDBC driver with APAR PQ48383 installed or the DB2 Universal JDBC Driver.

Prerequisite: “Installing IMS Java” on page 1

To configure DB2 UDB for z/OS for IMS Java:
1. Create a data set with the following attributes. This data set is the JAVAENV DD statement data set.
   - Organization: PS
2. In the data set that you created in step 1 on page 59, add the ENVAR keyword with following parameters:

- **JAVA_HOME=**
  The HFS directory of the JVM.

- **DB2_HOME=**
  The HFS directory of the JDBC driver for DB2 UDB for z/OS.

- **CLASSPATH=**
  The HFS directories of the client application Java class files. You do not specify the CLASSPATH= if you specify the client application Java class files in the stored procedure definition.

- **LIBPATH=**
  The HFS directory of the file libJavTDLI.so.

- **TMSUFFIX=**
  The HFS directories of the IMS Java class libraries:

  TMPREFIX=
  pathprefix
  /usr/lpp/ims/imsjava81/imsjava.jar

Figure 16 shows a sample JAVAENV data set.

```
ENVAR("CLASSPATH=/usr/lpp/ims/imsjava81/samples.jar",
  "DB2_HOME=/usr/lpp/db2/db27",
  "JAVA_HOME=/usr/lpp/J1.3",
  "LIBPATH=/usr/lpp/ims/imsjava81",
  "TMSUFFIX=/usr/lpp/ims/imsjava81/imsjava.jar")
```

Figure 16. Sample JAVAENV Data Set

3. Set DB2 UDB for z/OS environment variables in UNIX System Services by issuing the following commands:

- `export SQLJ_HOME=` location of the DB2 SQLJ driver (for example /usr/lpp/db2/db2710)
- `export JDBC_HOME=` location of the DB2 JDBC driver (for example /usr/lpp/db2/db2710)
- `export JAVA_HOME=` location of SDK1.3 (for example /usr/lpp/java/J1.3)
- `export DB2SQLJPROPERTIES=/path/db2sqljjdbc.properties` (you will create this file later)
- `export CLASSPATH=$DB2_HOME/classes/db2sqljdbc.zip`
- `export CLASSPATH=$CLASSPATH:$JDBC_HOME/classes/db2sqljruntime.zip`
- `export CLASSPATH=$CLASSPATH:$JDBC_HOME/classes/db2sqljdbc.zip`
- `export CLASSPATH=$CLASSPATH:pathprefix/usr/1pp/ims/imsjava81/imsjava.jar`
- `export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib:
- export LD_LIBRARY_PATH=$SQLJ_HOME/lib:$JDBC_HOME/lib
- export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAVA_HOME/lib
- export PATH=$SQLJ_HOME/bin:$PATH
- export STEPLIB=yourDB2HLQ.DSNEXIT:yourDB2HLQ.DSNLOAD:
- export STEPLIB=yourDB2HLQ.DSNLOAD2:yourDB2HLQ.DSNLINK:$STEPLIB

Next: “Running the IMS Java IVP from DB2 UDB for z/OS” on page 61
Running the IMS Java IVP from DB2 UDB for z/OS

To verify that DB2 UDB for z/OS is configured correctly and that IMS Java is installed correctly, run the IMS Java installation verification program (IVP) on DB2 UDB for z/OS.

The IMS Java IVP for DB2 UDB for z/OS is two programs:
- The Java application DB2IvpClient, which runs under UNIX System Services.
- The stored procedure DB2IvpStoredProcedure, which runs in a WLM-managed address space.

Prerequisites:
- “Configuring DB2 UDB for z/OS for IMS Java” on page 59
- Ensure that the standard IMS IVPs have been run. These IVPs prepare the DBD for the IVP database, named IVPDB2, and load the IVP database. They also prepare the IMS Java application PSB (named DFSIVP37), build ACBs, and prepare other IMS control blocks required by the IMS Java IVPs. For details of how to run the IMS IVPs, see IMS Version 8: Installation Volume 1: Installation Verification.

To run the IMS Java IVP on DB2 UDB for z/OS:
1. Ensure that the samples.tar file has been expanded. This topic assumes that its contents are in the directory pathprefix/usr/lpp/ims/imsjava81. For more information on expanding the samples.tar file, see “Installing IMS Java” on page 1.
2. In the JAVAENV data set, modify the CLASSPATH= parameter to pathprefix/usr/lpp/ims/imsjava81/samples.jar.
3. Edit the IMS-provided V71AWLM procedure as follows (if IMS.SDFSRESL does not contain the DRA startup table, add that data set to the DFSRESLB DD statement):
   ```
   //V71AWLM PROC RGN=0M,APPLENV=,
   //   DB2SSN=,NUMTCB=,
   //* Define the V71AWLM procedure parameters here on in the service policy.
   //IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
   //   PARM='&DB2SSN,&NUMTCB,&APPLENV'
   //STEPLIB DD DISP=SHR,DSN=CEE.SCEERUN
   //DB2 Library
   //   DD DISP=SHR,DSN=yourDB2HLQ.SDSNLLOAD
   //   DD DISP=SHR,DSN=yourDB2HLQ.SDSNLOAD2
   //   DD DISP=SHR,DSN=yourDB2HLQ.RUNLIB.LOAD
   //* DBRM Library
   //   DD DISP=SHR,DSN=yourHLQ.SDNSDBRM
   //DFSRESLB DD DISP=SHR,DSN=IMS.SDFSRESL
   //JAVAENV DD DISP=SHR,DSN=dat set with ENVAR settings
   //JSPDEBUG DD SYSOUT=**
   //CEEDUMP DD SYSOUT=**
   //SYSPRINT DD SYSOUT=**
   //SYSOUT DD SYSOUT=**
   ```
4. Create a new service policy for WLM. You can define the V71AWLM procedure parameters in the service policy or you can modify the procedure itself.
5. Define the procedure V71AWLM to RACF®.
6. Start DB2 UDB for z/OS, the WLM-managed address space, and IMS DB.
7. Define the stored procedure to DB2 UDB for z/OS by running the following job (Your_WLM_Environment_Name must match the APPLENV= parameter of the V71ALWM procedure):
IMS Java IVP for DB2 UDB for z/OS

```
/**CREATIVP JOB , name',
  MSGCLASS=H,TIME=3,
  USER=SYSADM, PASSWORD=XXXXXXXX,
  MSGLEVEL=1
/**CREATJSP EXEC PGM=IKJEFT01, DYNAMNBR=20
/**STEPLIB DD DISP=SHR, DSN=DB2HLQ.DSNEXIT
/** DD DISP=SHR, DSN=DB2HLQ.SDSNLOAD
/**SYSTSIN DD SYSOUT=* 
/**STPLIB DD DISP=SHR, DSN=DB2HLQ.DSNEXIT 
/** DD DISP=SHR, DSN=DB2HLQ.SDSNLOAD
/**SYSTSPRT DD SYSOUT=* 
/**SSTSPRT DD SYSOUT=* 
/**SYSTRM DD SYSOUT=* 
/**SYSTSSIN DD * 
/**CREATE PROCEDURE IVPStoredProc(VARCHAR(50) IN, VARCHAR(800) OUT) 
/**FENCED 
/**NO SQL
/**DYNAMIC RESULT SET 0
/**EXTERNAL NAME 'samples.ivp.db2.IVPStoredProcedure.execute'
/**PARMSTYLE JAVA
/**COLLID DSNJDBC
/**WLM_ENVIRONMENT Your_WLM_Environment_Name; 
/**GRANT EXECUTE ON PROCEDURE IVP StoredProc TO PUBLIC;
8. Create a DB2 plan that runs the client program by running the following job:
/**BNDIVPCL JOB , 'YOUR NAME',
/** MSGCLASS=H,TIME=3,
/** USER=SYSADM, PASSWORD=XXXXXXXX,
/** MSGLEVEL=1
/**BINDCLNT EXEC PGM=IKJEFT01, DYNAMNBR=20
/**STEPLIB DD DISP=SHR, DSN=DB2HLQ.DSNEXIT 
/** DD DISP=SHR, DSN=DB2HLQ.SDSNLOAD
/**DDRLIB DD DISP=SHR, DSN=DB2HLQ.SDSNDBRM 
/**SYSTSIN DD SYSOUT=* 
/**SYSTRM DD SYSOUT=* 
/**SYSUDUMP DD SYSOUT=* 
/**CREATE PLAN(DB2IVPCL) KEEPDYNAMIC(YES) 
/**PKLIST(DB2IVPCL) PARM('RC0') 
/**RUN PROGRAM(DB2IVPCL) PLAN(DB2IVPCL) 
/**LIB('DB2HLQ.RUNLIB.LOAD') 
/**END 
/**GRANT EXECUTE ON PLAN DB2IVPCL TO PUBLIC; 
*/ 
9. In UNIX System Services, in the directory that you specified by the export 
DB2SQLJPRTERRITIES command, create the file db2sqljjdbc.properties that 
contains the following:
```
10. Run the client application by issuing following command from UNIX System Services:

```
java samples.ivp.db2.DB2IvpClient IMSID
```

If the program ran successfully, it displays a first name, last name, extension, and zip code.

---

### Running the IMS Java Sample Application on DB2 UDB for z/OS

IMS Java provides a sample dealership application in addition to the IVP.

The IMS Java sample application for DB2 UDB for z/OS is two programs:
- The Java application DB2AutoClient, which runs under UNIX System Services
- The stored procedure DB2Auto, which runs in a WLM-managed address space.

**Prerequisites:**

- Appendix A, “Preparing to Run the Dealership Samples,” on page 129
- “Running the IMS Java IVP from DB2 UDB for z/OS” on page 61

To run the IMS Java sample dealership application:

1. Ensure that the DB2 UDB for z/OS environment is configured and running as required by the IVP. If the DB2 UDB for z/OS environment is not configured and running for the IVP, perform steps 2 through 6 in “Running the IMS Java IVP from DB2 UDB for z/OS” on page 61 before continuing.

2. Define the stored procedure to DB2 UDB for z/OS by running the following job (Your_WLM_Environment_Name must match the APPLENV= parameter of the V71AWLM procedure): 

```sql
//CREATDLR JOB 'name',
//  MSGCLASS=H,TIME=3,
//  USER=SYSADM,PASSWORD=XXXXXXX,
//  MSGLEVEL=(1)
//CREATJSP EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB DD DISP=SHR,DSN=DB2HLQ.DSNEXIT
// DD DISP=SHR,DSN=DB2HLQ.SDSNLOAD
//SYSTSPRT DD SYSOUT**
//SYSTSIN DD *
//DSN SYSTEM(DB2_Subsystem_Name)
//  RUN PROGRAM(DSNTIAD) PLAN(DSNTIA71) -
//  LIB('DB2HLQ.RUNLIB.LOAD') -
//  PARM('RC0')
//SYSPRINT DD SYSOUT**
//SYSUDUMP DD SYSOUT**
//SYSDUMP DD *
CREATE PROCEDURE AutoListModels(VARCHAR (100) IN, VARCHAR (100) OUT,
  VARCHAR (100) OUT, VARCHAR(100) OUT,
  VARCHAR (100) OUT, VARCHAR(100) OUT)
FENCED
NO SQL
LANGUAGE JAVA
DYNAMIC RESULT SET 0
EXTERNAL NAME 'samples.dealership.db2.DB2Auto.listModels'
PARAMETER STYLE JAVA
COLLID DSNJDBC
WLM ENVIRONMENT Your_WLM_Environment_Name;
GRANT EXECUTE ON PROCEDURE AutoListModels TO PUBLIC;
```
3. Create a DB2 plan that runs the client program by running the following job:

```
//BNDCLRLC JOB ,"name",
// MSGCLASS=H,TIME=3,
// USER=SYSADM, PASSWORD=XXXXXXXX,
// MSGLEVEL=(1)
//BINDCLNT EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB DD DISP=SHR,DSN=DB2HLQ.DSNEXIT
// DDBRMLIB DD DISP=SHR,DSN=DB2HLQ.DSNDNBR
//SYSTSPRT DD SYSOUT**
//SYSPRINT DD SYSOUT**
//SYSUDUMP DD SYSOUT**
//SYSTSIN DD *
DSN SYSTEM(DB2ID)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC1) ISOLATION(UR) -
  ACTION(REPLACE) VALIDATE(BIND)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC2) ISOLATION(CS) -
  ACTION(REPLACE) VALIDATE(BIND)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC3) ISOLATION(RS) -
  ACTION(REPLACE) VALIDATE(BIND)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC4) ISOLATION(UR) -
  ACTION(REPLACE) VALIDATE(BIND)
  BIND PLAN(DB2DLRCL) KEEPDYNAMIC(YES) ACTION(REPLACE) -
  PKLIST(DSNJDBC.DSNJDBC1, -
  DSNJDBC.DSNJDBC2, -
  DSNJDBC.DSNJDBC3, -
  DSNJDBC.DSNJDBC4)
RUN PROGRAM(DSNTEP2) PLAN(DSNTEP71) -
LIB('DB2HLQ.RUNLIB.LOAD')
END
//SYSTSIN DD *
GRANT EXECUTE ON PLAN DB2DLRCL TO PUBLIC;
/*
*/
```

4. In UNIX System Services, edit the file db2sqljdbc.properties by changing the DB2SQLJPLANNAME= parameter to DB2DLRCL:

```
DB2SQLJSSID=yourDB2ID
DB2SQLJPLANNAME=DB2DLRCL
DB2SQLJATTACHTYPE=RRSAF
DB2SQLJDBRMLIB=DB2HLQ.DSNDNBR
```

5. Run the client application by issuing following command from UNIX System Services:

```
java samples.dealership.db2.DB2AutoClient IMSID
```

The sample application displays information about models of cars.

---

**Running Your Stored Procedure from DB2 UDB for z/OS**

**Prerequisite:** "Running the IMS Java IVP from DB2 UDB for z/OS" on page 61

To run your Java application that accesses IMS DB on DB2 UDB for z/OS:

1. In the JAVAENV data set, modify the CLASSPATH= parameter to point to your application files. If your application files are in JAR files, include the JAR file names. If the application files are not in JAR files, do not include the file names.

2. Edit the IMS-provided V71AWLM procedure as follows (if IMS.SDFSRESL does not contain the DRA startup table, add that data set to the DFSRESLB DD statement):

```
//V71AWLM PROC RGN=0M,APPLENV=,
// DB2SSN=,NUMTCB=
//* Define the V71AWLM procedure parameters here on in the service policy.
//IEFPROC EXEC PGM=SNX9WLM,REGION=&RGN,TIME=NOLIMIT,
```
3. Create a new service policy for WLM. You can define the V71AWLM procedure parameters in the service policy or you can modify the procedure itself.

4. Define the procedure V71AWLM to RACF.

5. Start DB2 UDB for z/OS, the WLM-managed address space, and IMS DB.

6. Define the stored procedure to DB2 UDB for z/OS by running the following job (Your_WLM_Environment_Name must match the APPLENV= parameter of the V71AWLM procedure):

```
//name JOB,'name',
// MSGCLASS=H,TIME=3,
// USER=user, PASSWORD=xxxxxxxx,
// MSGLEVEL=(1)
//CREATJSP EXEC PGM=IKJEFT01, DYNAMNBR=20
//STEPLIB DD DISP=SHR, DSN=DB2HLQ.DSNEXIT
// DD DISP=SHR, DSN=DB2HLQ.DSNLOAD
//SYSTSIN DD *
//DSN SYSTEM(DB2_Subsystem_Name)
//RUN PROGRAM(DSNTIAD) PLAN(DSNTIA71) -
// LIB('DB2HLQ.RUNLIB.LOAD') -
//PARM('RC0')
//SYSPRINT DD SYSOUT**
//SYSDUMP DD SYSOUT**
//SYSIN DD *
//CREATE PROCEDURE StoredProcName (... IN, ... OUT)
//FENCED
//NO SQL
//LANGUAGE JAVA
//DYNAMIC RESULT SET 0
//EXTERNAL NAME 'package.StoredProcedure.targetMethod'
//PARAMETER STYLE JAVA
//COLLID DSNJDBC
//WLM ENVIRONMENT Your_WLM_Environment_Name;
//GRANT EXECUTE ON PROCEDURE StoredProcName TO PUBLIC;
```

7. Create a DB2 plan that runs the client program by running the following job:

```
//name JOB,'name',
// MSGCLASS=H,TIME=3,
// USER=user, PASSWORD=xxxxxxxx,
// MSGLEVEL=(1)
//BINDCLNT EXEC PGM=IKJEFT01, DYNAMNBR=20
//STEPLIB DD DISP=SHR, DSN=DB2HLQ.DSNEXIT
// DD DISP=SHR, DSN=DB2HLQ.DSNLOAD
//DBRM DD DISP=SHR, DSN=DB2HLQ.DSNDBG
//SYSTSIN DD *
//DSN SYSTEM(DB2ID)
//BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC1) ISOLATION (UR) -
// ACTION (REPLACE) VALIDATE (BIND)
Running Your Stored Procedure

```sql
BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC2) ISOLATION(CS) - 
ACTION(REPLACE) VALIDATE(BIND)
BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC3) ISOLATION(RS) - 
ACTION(REPLACE) VALIDATE(BIND)
BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC4) ISOLATION(RR) - 
ACTION(REPLACE) VALIDATE(BIND)
BIND PLAN(plan_name) KEEPDYNAMIC(YES) ACTION(REPLACE) - 
PKLIST(DSNJDBC.DSNJDBC1, - 
DSNJDBC.DSNJDBC2, - 
DSNJDBC.DSNJDBC3, - 
DSNJDBC.DSNJDBC4)
RUN PROGRAM(DSNTEP2) PLAN(DSNTEP71) - 
LIB('DB2HLQ.RUNLIB.LOAD')
END

//SYSIN DD *
GRANT EXECUTE ON PLAN plan_name TO PUBLIC;
/*
//
8. In UNIX System Services, in the directory that you specified by the export
DB2SQLJPPROPERTIES command, create the file db2sqljjdbc.properties that contains
the following:
DB2SQLJSSID=yourDB2ID
DB2SQLJPLANNAME=plan_name
DB2SQLJATTACHTYPE=RRSAF
DB2SQLJDBRMLIB=DB2HLQ.SDSNDBRM
9. Run the client application.
```

Developing DB2 UDB for z/OS Stored Procedures that Access IMS DB

The stored procedure must perform the following tasks in the order listed. An example is given for each step:

1. Load the IMS JDBC driver:
   ```java
   Class.forName("com.ibm.ims.db.DLIDriver");
   ```

2. Create an IMS JDBC connection:
   ```java
   connection = DriverManager.getConnection
   ("jdbc:dli:package.DatabaseViewName/DRAname");
   ```

3. Create a statement:
   ```java
   Statement statement = connection.createStatement();
   ```

4. Query the IMS database:
   ```java
   ResultSet results = statement.executeQuery(query);
   ```

5. Move the results to the output parameters:
   ```java
   parmOut[...] = ...;
   ```

6. Close the connection:
   ```java
   connection.close();
   ```
Chapter 5. CICS Applications

Java applications that run on CICS Transaction Server for z/OS can access IMS databases by using IMS Java.

Java applications use the IMS Java class libraries to access IMS. Other than the IMS Java layer, access to IMS from a Java application is the same as for a non-Java application.

Figure 17 shows a JCICS application accessing an IMS database using ODBA and IMS Java.

```
Figure 17. CICS Application Using IMS Java
```

The following topics provide additional information:

- "Configuring CICS for IMS Java"
- “Running the IMS Java IVP on CICS” on page 68
- “Running the IMS Java Sample Application on CICS” on page 69
- “Running Your Applications on CICS” on page 70
- “Developing CICS Applications that Access IMS DB” on page 71

Configuring CICS for IMS Java

To run Java applications that access IMS databases in a CICS environment, you must have CICS Transaction Server for z/OS Version 2 or later installed.

Prerequisite: "Installing IMS Java” on page 1

To configure CICS for IMS Java:

1. Build the CICSPSB DLL:
   a. Modify the Makefile, which is in the pathprefix/usr/lpp/ims/imsjava81/cics directory, by changing both occurrences of your pdse.loadlib to the data set that will store the CICSPSB module.
   b. Set the CICS library to the _CX_LSYSLIB environment variable by issuing the following command from the UNIX System Services prompt:
      ```
      export _CX_LSYSLIB=CICS_library_location
      ```
Configuring CICS for IMS Java

c. Run the Makefile by issuing the following command from the UNIX System Services prompt:
   
   make

d. In the data set that you specified in the Makefile, create two data set members named CICSPSB and DFSCLIB.
e. Add the data set that you specified in the Makefile to CICS STEPLIB concatenation.

2. Modify the CICS environment member DFHJVMPR, which is the JVM profile:
   a. Add a TMPREFIX variable, which adds libraries to the beginning of the middleware path.
b. To the TMPREFIX variable, add the path to the IMS Java class libraries as follows:
   
   TMPREFIX=pathprefix/usr/lpp/ims/imsjava81/imsjava.jar

c. Update the LIBPATH variable so that it contains the path to the file libJavTDLI.so as follows:
   
   LIBPATH=pathprefix/usr/lpp/ims/imsjava81

Next: Running the IMS Java IVP on CICS

Related Reading: For detailed information about CICS system definition, see the CICS Transaction Server for z/OS: CICS System Definition Guide.

Running the IMS Java IVP on CICS

After you configure CICS to run Java applications that access IMS databases, verify that IMS Java is installed correctly and that CICS is configured correctly by running the IMS Java installation verification program, which is named CICSIVP.

Prerequisites:

- "Configuring CICS for IMS Java" on page 67
- Ensure that the standard IMS IVPs have been run. These IVPs prepare the DBD for the IVP database, named IVPDB2, and load the IVP database. They also prepare the IMS Java Application PSB (named DFSIVP37), build ACBs, and prepare other IMS control blocks required by the IMS Java IVPs. For details of how to run the IMS IVP procedures, see IMS Version 8: Installation Volume 1: Installation Verification.

To run the IMS Java IVP on CICS:

1. Ensure that the samples.tar file has been expanded. This topic assumes that its contents are in the directory pathprefix/usr/lpp/ims/imsjava81. For more information on expanding the samples.tar file, see "Installing IMS Java" on page 1.

2. Create an HFS file named dfjjvmpr.props that contains the following class path:
   
   ibm.jvm.shareable.application.class.path=/pathprefix/usr/lpp/ims/imsjava81/samples.jar

   If you need to debug your application, you can also add the JVM debug options in this file.

3. In the DFHJVMPR member of the DFHJVM data set, add:
   
   JVMPROPS=path/dfjjvmpr.props
   STDOUT=path
   STDERR=path
4. Start IMS DB and CICS.
5. Turn off the uppercase translation feature of CICS by entering CEOT NOUCTRAN.
   By default, everything you type on the CICS terminal is converted to uppercase. However, the samples.jar file and path contain lowercase letters that must remain in lowercase.
6. Define a program that can run the CICSIVP application (JVM class):
   a. From the CICS terminal, enter: CEDA DEFINE PROGRAM
   b. In the list of program attributes, type the following:
      
      | PROGRAM  | => cicsivp |
      | Group    | => ivp |
      | COncurrency | => Threadsafe |
      | JVM      | => Yes |
      | JVMClass  | => samples.ivp.cics.CICSIVP |
   c. Press F3 to return to the main CICS terminal.
7. Define a transaction that can run the program:
   a. From the CICS terminal, enter: CEDA DEFINE TRANSACTION
   b. In the list of transaction attributes, type the following:
      
      | TRANSaction | => civp |
      | Group       | => ivp |
      | PROGram     | => cicsivp |
   c. Press F3 to return to the main CICS terminal.
8. Install the program that you defined in step 6:
   a. From the CICS terminal, enter: CEDA INSTALL
   b. In the list of program attributes, type the following:
      
      | PROGram  | => cicsivp |
      | Group    | => ivp |
   c. Press F3 to return to the main CICS terminal.
9. Install the transaction that you defined in step 7:
   a. From the CICS terminal, enter: CEDA INSTALL
   b. In the list of transaction attributes, type the following:
      
      | TRANSaction | => civp |
      | Group       | => ivp |
   c. Press F3 to return to the main CICS terminal.
10. Run the transaction by entering: civp
    If the transaction ran successfully, the program correctly returns a first name, last name, zip code, and extension.

---

## Running the IMS Java Sample Application on CICS

IMS Java provides the dealership sample application to run on CICS. The dealership sample files for CICS are located in `pathprefix/usr/lpp/ims/imsjava81/samples/dealership/cics`.

**Prerequisites:**

- [Running the IMS Java IVP on CICS](#) on page 68
- [Appendix A, “Preparing to Run the Dealership Samples,”](#) on page 129

To run the IMS Java dealership sample on CICS:

1. Modify the HFS `dfjjvmpr.props` file to set the `ibm.jvm.shareable.application.class.path=` parameter to the path of the...
IMS Java Sample Application on CICS

application. The location of the dfjjvmpr.props file is specified by the
JVMPROPS variable in the CICS JVM profile:
ibm.jvm.shareable.application.class.path=
/pathprefix/usr/lpp/ims/imsjava81/samples/samples.jar

2. Start IMS DB and CICS.

3. Turn off the uppercase translation feature of CICS by entering: CEOT NOUCTRAN

4. Define a program that can run the IMS Java sample application (JVM class):
   a. From the CICS terminal, enter: CEDA DEFINE PROGRAM
   b. In the list of program attributes, type the following:
      PROGram ==> cicsauto
      Group ==> imsj
      COncurrency ==> Th Leadsafe
      JVM ==> Yes
      JVMClass ==> samples.ivp.cics.CICSAuto
   c. Press F3 to return to the main CICS terminal.

5. Define a transaction that can run the program:
   a. From the CICS terminal, enter: CEDA DEFINE TRANSACTION
   b. In the list of transaction attributes, type the following:
      TRANSaction ==> cicssamp
      Group ==> imsj
      PROGram ==> cicsauto
   c. Press F3 to return to the main CICS terminal.

6. Install the program that you defined in step 4:
   a. From the CICS terminal, enter: CEDA INSTALL
   b. In the list of program attributes, type the following:
      PROGram ==> cicsauto
      Group ==> imsj
   c. Press F3 to return to the main CICS terminal.

7. Install the transaction that you defined in step 5:
   a. From the CICS terminal, enter: CEDA INSTALL
   b. In the list of transaction attributes, type the following:
      TRANSaction ==> cicssamp
      Group ==> ivp
   c. Press F3 to return to the main CICS terminal.

8. Run the transaction by entering: cicssamp
   The sample application displays information about models of cars.

Running Your Applications on CICS

Prerequisite: "Running the IMS Java IVP on CICS" on page 68

To run your Java application that accesses IMS DB from CICS:

1. Modify the HFS dfjjvmpr.props file to set the
   ibm.jvm.shareable.application.class.path= parameter to the path of the
   application. The location of the dfjjvmpr.props file is specified by the
   JVMPROPS variable in the CICS JVM profile.

2. Start IMS DB and CICS.

3. Turn off the uppercase translation feature of CICS by entering: CEOT NOUCTRAN

4. Define a program that can run the Java application (JVM class).

5. Define a transaction that can run the program.
6. Install the program that you defined in step 4.

7. Install the transaction that you defined in step 5.

### Developing CICS Applications that Access IMS DB

The following programming model outlines the supported structure for JCICS applications that use IMS Java. The model is not complete, but it shows the normal flow of the application for both the JDBC and SSA access methods.

In a CICS environment, only one PSB can be allocated at a time. Therefore, an application can have only one active JDBC connection at a time. The application must close the JDBC connection before it opens another JDBC connection.

```java
public static void main(CommAreaHolder cah) { //Receives control
    conn = DriverManager.getConnection(...); //Establish DB connection

    repeat {
        results = statement.executeQuery(...); //Perform DB processing

        //send output to terminal
    }

    conn.close(); //Close DB connection
    return;
}
```
Developing CICS Applications
Chapter 6. JDBC Access to IMS Data

JDBC is the SQL-based standard interface for data access in the Java 2 SDK Standard Edition and Enterprise Edition. IMS Java's implementation of JDBC supports a selected subset of the full facilities of the JDBC 2.1 API.

IMS Java supports a subset of SQL keywords. Some keywords have specific IMS usage requirements. For this usage information, see "Supported SQL Keywords" on page 76.

Recommendation: Use JDBC to access IMS data instead of the IMS Java hierarchical database interface.

This chapter uses the sample dealership applications that are shipped with IMS Java to describe how to use JDBC to access an IMS database.

The following topics provide additional information:
- "Comparison of Hierarchical and Relational Databases"
- "Supported SQL Keywords" on page 76
- "Supported SQL Aggregate Functions" on page 83
- "Supported JDBC Interfaces" on page 84
- "JDBC Prepared Statements for SQL" on page 86
- "Supported JDBC Data Types" on page 86
- "General Mappings from COBOL Copybook Types to IMS Java and Java Data Types" on page 88
- "JDBC Recommendations for IMS Databases" on page 90
- "Java Metadata Classes for IMS Databases" on page 90
- "Sample Application that Uses JDBC" on page 93

Comparison of Hierarchical and Relational Databases

A hierarchical database segment definition defines the fields for a set of segment instances similar to the way a relational table defines columns for a set of rows in a table. In this way, segments relate to relational tables, and fields in a segment relate to columns in a relational table.

The name of an IMS segment becomes the table name in an SQL query, and the name of a field becomes the column name in the SQL query.

A fundamental difference between segments in a hierarchical database and tables in a relational database is that, in a hierarchical database, segments are implicitly joined with each other. In a relational database, you explicitly join two tables. A segment instance in a hierarchical database is already joined with its parent segment and its child segments, which are all along the same hierarchical path. In a relational database, this relationship between tables is captured by foreign and primary keys.

This section compares the dealership sample database, which is shipped with IMS Java, to a relational representation of the database.

The dealership sample database contains five segment types, which are shown in Figure 18 on page 74. The root segment is the Dealer segment. Under the Dealer
segment is its child segment, the Model segment. Under the Model segment are its children: the segments Order, Sales, and Stock. See Figure 31 on page 91 for the database description (DBD) of the dealership sample database.

![Diagram of Dealership Database](image)

**Figure 18. Sample Dealership Database**

The Dealer segment identifies a dealer selling cars, and the segment contains a dealer name and a unique dealer number in the fields DLRNAME and DLRNO.

Dealers carry car types, each of which has a corresponding Model segment. A Model segment contains a type code in the field MODTYPE.

There is an Order segment for each car that is ordered for the dealership. A Stock segment is created for each car that is available for sale in the dealer's inventory. When the car is sold, a Sales segment is created.

**Figure 19 on page 75** shows a relational representation of the IMS database record shown in Figure 18.

**Important:** This figure is only to help you understand how to use JDBC calls in a hierarchical environment. IMS Java does not change the structure of IMS data in any way.
If a segment does not have a unique key, which is similar to a primary key in relational databases, view the corresponding relational table as having a generated primary key added to its column (field) list. An example of a generated primary key is in the Model table (segment) of Figure 19. Similar to referential integrity in relational databases, you cannot insert, for example, an Order (child) segment to the database without it being a child of a specific Model (parent) segment.

Also note that the field (column) names have been renamed. You can rename segments and fields to more meaningful names using the DLIModel utility.

An occurrence of a segment in a hierarchical database corresponds to a row (or tuple) of a table in a relational database. Figure 20 on page 76 shows three dealership database records. The Dealer segment occurrences have dependent Model segment occurrences. The relational representation of these segment occurrences is shown in Figure 21 on page 76.
The following example shows the SELECT statement of an SQL call. Model is a segment name that is used as a table name in the query:

```
SELECT *
FROM Model
```

In the following example, ModelTypeCode is the name of a field contained in the Model segment and it is used in the SQL query as a column name:

```
SELECT * FROM Model
WHERE ModelTypeCode = '062579'
```

In both of the preceding examples, Model and ModelTypeCode are alias names that you assign by using the DLIModel utility. These names will likely not be the same 8-character names used in the database description (DBD) for IMS. Alias names act as references to the 8-character names that are described in the DBD.

### Supported SQL Keywords

The following portable SQL keywords are currently supported by IMS Java. IMS-specific usage for frequently-used keywords is described in this section. None of the keywords is case-sensitive. These keywords are a subset of all SQL keywords, which are listed in Appendix B, “SQL Keywords,” on page 133.
Important: Because the IMS Java SQL parser supports portable SQL, you cannot use any SQL keywords as Java aliases for PCBs, fields, or segments. When you define Java aliases, do not use an SQL keyword. If a PCB, segment, or field has the same name as an SQL keyword, you must explicitly define a different Java alias for it. If you use an SQL keyword as an alias for a PCB, segment, or field, your application will receive an error when it attempts an SQL query. For a complete list of SQL keywords, see Appendix B, “SQL Keywords,” on page 133.

The following topics provide additional usage information about SQL keywords:

- “SELECT Statement Usage”
- “INSERT Statement Usage” on page 80
- “DELETE Statement Usage” on page 81
- “UPDATE Statement Usage” on page 81
- “FROM Clause Usage” on page 81
- “WHERE Clause Usage” on page 82

## SELECT Statement Usage

A SELECT statement is a query used as a top-level SQL statement. A SELECT statement can be executed against a Statement or PreparedStatement object, which returns the results as a ResultSet object.

Figure 22 on page 78 shows sample code that uses the results of a SELECT query to update the modelOutput object with the model information. This example requires an inputMessage object with the ModelTypeCode field information.
public boolean getModelDetails(InputMessage inputMessage, 
   ModelOutput modelOutput) throws IMSException {

   // Parse the input message for ModelTypeCode
   String queryString = "SELECT * FROM DealershipDB.Model WHERE ModelTypeCode = " + "+" + inputMessage.getString("ModelTypeCode").trim() + ";";
   // Create a statement and execute it to get a ResultSet
   try {
      Statement statement = connection.createStatement();
      ResultSet results = statement.executeQuery(queryString);
      // Send back the result of the query
      // Note: because "ModelTypeCode" is unique - only 1 row 
      // is returned
      if (results.next()) {
         modelOutput.setString("ModelTypeCode", 
            results.getString("Type").trim());
         modelOutput.setString("Make", 
            results.getString("CarMake").trim());
         modelOutput.setString("Model", 
            results.getString("CarModel").trim());
         modelOutput.setString("Year", 
            results.getString("CarYear").trim());
         modelOutput.setString("CityMiles", 
            results.getString("EPACityMileage").trim());
         modelOutput.setString("HighwayMiles", results.getString( 
            "EPAHighwayMileage").trim());
         modelOutput.setString("Price", 
            results.getString("Price").trim());
         modelOutput.setString("Horsepower", 
            results.getString("Horsepower").trim());
         return true;
      }
      else {
         reply("Unknown Type");
         return false;
      }
   } 
   catch (SQLException e) {
      reply("Query Failed:" + e.toString());
      return false;
   }

Figure 22. Example of SELECT Statement Query Results

Notice that the PCB reference name, DealershipDB, qualifies the Model segment name in the query string. You qualify the segment name with the PCB name because a PSB can contain multiple PCBs, and the PCBs can have segments with the same name. When you use the PCB name to indicate the exact segment to access, you avoid the ambiguity checking and improve the performance of your application.

Note: The method trim() is used because IMS character fields are padded with blanks if they are not long enough. The method trims off the extra blanks.

Figure 22 illustrates the use of a Statement object to execute an SQL query. You can also use a PreparedStatement object to execute an SQL query. A PreparedStatement object has two advantages over a Statement object:

- The SQL can be parsed one time for many executions of the query.
- You can build the query and use substitute values with each execution.
Selecting Multiple Segments

By using IMS Java to write IMS applications, you can avoid the long process of coding segment search arguments (SSAs) for every segment in the path that leads to the segment being queried. Instead, you can use the IMS Java JDBC driver for SQL queries to retrieve results from any segment in the path that leads to the segment being queried.

The primary difference between SQL queries to relational databases and SQL queries to IMS using IMS Java is that the hierarchical structure of an IMS database eliminates the need for the join that is required for tables in relational databases.

For example, Figure 23 is a query to a relational database for the address of a dealership that sells a particular car model (AnyCarModel):

```sql
SELECT Dealer.Address
FROM DealershipDB.Dealer, DealershipDB.Model
WHERE Model.CarMake = 'AnyCarModel'
    AND Dealer.DealerName = Model.CarrierName
```

*Figure 23. Sample Relational Database Query*

In a relational database query, you must query two independent tables (Dealer and Model) and indicate how they are joined using a WHERE clause. This query is not valid against an IMS database.

In an IMS Java application, you can write the query in Figure 24 to access the same data in a hierarchical database using a WHERE clause:

```sql
SELECT Dealer.Address
FROM DealershipDB.Model
WHERE Model.CarMake = 'AnyCarModel'
```

*Figure 24. Sample Hierarchical Database Query*

In a hierarchical database, all data in segments along the hierarchical path from the root segment to the target segment are implicitly included in the query results, and therefore they do not need to be explicitly stated. In Figure 24, the information about the Dealer segment is included in the result set because it is along the hierarchical path to the Model segment.

**Requirement:** This implicit inclusion of segments is called a path call. For a path call to be made, the PROCOPT parameter in the PCB or SENSEG statement of the PSB source must include ‘P’. If P is not included in the PROCOPT parameter and you issue a query that requires a path call to be made, an SQLException object is generated.

Segment-Qualified Fields

SQL dictates that whenever a field is common between two tables in an SQL query, the desired field must be table-qualified to resolve the ambiguity. Similarly, whenever a field name is common in any two segments along a hierarchical path, the field must be segment-qualified. For example, if a PCB has two segments, segment ROOT and segment CHILD, and both possess a field named id, any query that references the id field must be segment-qualified.

The following example is incorrect because the id field is not segment-qualified:
SELECT id
FROM PCBName.CHILD
WHERE id='10'

The following example is correct because the id field is segment-qualified:
SELECT CHILD.id
FROM PCBName.CHILD
WHERE ROOT.id='10'

Recommendations:

- For performance reasons, always qualify fields by prefixing the field names with a segment. This improves performance because IMS Java does not need to search through all the segments to locate the field and check for ambiguity.
- Although you do not need to provide the PCB reference name on the query unless the query is ambiguous without it, you should always provide the PCB reference name to remove ambiguity and to eliminate the need for checking.

Summary of SELECT Statement Usage

When using the SELECT statement in SQL calls to IMS databases:

- Qualify fields by prefixing them with segment names.
- Select fields that are in any segment from the root segment down to the segment in the FROM clause.

INSERT Statement Usage

An INSERT statement inserts a segment instance with the specified data under any number of parent segments that match the criteria specified in the WHERE clause. All field names must be specified in the statement, unless you set a default value in the IMS Java metadata class with the DLIModel utility control statements. For more information about the DLIModel control statements, see the [IMS Version 8: Utilities Reference: System](#).

Figure 25 shows an example of an INSERT statement that inserts a segment occurrence in the database using the DealershipDB PCB:

```sql
INSERT INTO DealershipDB.Sales (DateSold, PurchaserLastName, PurchaserFirstName, PurchaserAddress, SoldBy, StockVINNumber)
VALUES ('07032000', 'Beier', 'Otto', '101 W. 1st Street', 'Springfield, OH', 'S123', '1ABCD23E4G5678901234')
WHERE Dealer.DealerNumber = 'A123'
AND Model.ModelTypeCode = 'K1'
```

Figure 25. Sample INSERT Statement

You can set a default value for any field in a segment by using the FIELD control statement when running the DLIModel utility. For more information, see the description of the Default parameter of the DLIModel utility in [IMS Version 8: Utilities Reference: System](#).

One difference between JDBC queries to relational databases and to IMS is that standard SQL does not have a WHERE clause in an INSERT statement because tuples are being inserted into the table that is specified by the INTO keyword. In an IMS database, you are actually inserting a new instance of the specified segment, so you need to know where in the database this segment occurrence should be placed. With an INSERT statement, the WHERE clause is always necessary, unless
you are inserting a root segment. With a prepared statement, the list of values can include a question mark (?) as the value that can be substituted before the statement is executed. For example:

```sql
INSERT INTO DealershipDB.Model(ModelTypeCode, CarMake, CarModel, CarYear, Price, 
     EPACityMileage, EPAHighwayMileage, Horsepower) 
VALUES (?, ?, ?, ?, ?, ?, ?, ?) 
WHERE Dealer.DealerNumber=?
```

### DELETE Statement Usage
A DELETE statement can delete any number of segment occurrences that match the criteria specified in the WHERE clause. A DELETE statement with a WHERE clause also deletes the child segments of the matching segments. If no WHERE clause is specified, all of the segment occurrences of that type are deleted as are all of their child segment occurrences. Figure 26 shows an example of a DELETE statement:

```sql
DELETE FROM DealershipDB.Order 
WHERE Dealer.DealerNumber = '123' AND OrderNumber = '345'
```

*Figure 26. Sample DELETE Statement*

### UPDATE Statement Usage
An UPDATE statement modifies the value of the fields in any number of segment occurrences.

An UPDATE statement applies its SET operation to each instance of a specified segment with matching criteria in the WHERE clause. If the UPDATE statement does not have a WHERE clause, the SET operation is applied to all instances of the specified segment.

A SET clause contains at least one assignment. In each assignment, the values to the right of the equal sign are computed and assigned to columns to the left of the equal sign. For example, the UPDATE statement in Figure 27 is called to accept an order. When a customer accepts an order, the Order segment’s SerialNo and DeliverDate fields are updated.

```sql
UPDATE DealershipDB.Order 
SET SerialNo = '93234', DeliverDate = '12/11/2004' 
WHERE OrderNumber = '123'
```

*Figure 27. Sample UPDATE Statement*

### FROM Clause Usage
A FROM clause in IMS Java differs from standard SQL in that explicit joins are not required or allowed. Instead, the lowest-level segment in the query (in the SELECT statement and WHERE clause) must be the only segment that is listed in the FROM clause. The lowest-level segment in the FROM clause is equivalent to a join of all the segments, starting with the one that is listed in the FROM clause up the hierarchy to the root segment. For example, the FROM clause FROM DealershipDB.Order is equivalent to the following FROM clause in a relational query:

```sql
FROM DealershipDB.Order, DealershipDB.Model, DealershipDB.Dealer
```
PCB-Qualified SQL Queries

In IMS Java, connections are made to PSBs. Because there are multiple database PCBs in a PSB, there must be a way to specify which PCB (using its alias) in a PSB to use when executing an SQL query on the java.sql.Connection object. To specify which PCB to use, always qualify segments that are referenced in the FROM clause of an SQL statement by prefixing the segment name with the PCB name. You can omit the PCB name only if the PSB contains only one PCB.

Figure 28 shows a PCB-qualified SQL query.

```
SELECT *
FROM DealershipDB.Model
```

**Figure 28. PCB-Qualified SQL Query Example**

**Recommendation:** For clarity and performance reasons, always qualify segments in the FROM clause by using the PCB alias.

**Summary of FROM Clause Usage**

When using the FROM clause in SQL calls to IMS databases:

- Do not join segments in the FROM clause.
- List only one segment in the FROM clause.
- List the lowest-level segment that is used in the SELECT list and WHERE clause.
- Qualify the segment in the FROM clause by using the PCB alias.

**WHERE Clause Usage**

IMS Java converts the WHERE clause in an SQL query to an SSA list when querying a database. SSA rules restrict the type of conditions you can specify in the WHERE clause. This section describes how you must form your WHERE clause so that it can be converted into SSA lists.

The WHERE clause can contain fields only from the segment in the FROM clause or segments that are higher in the hierarchy. The fields in the WHERE clause can be DBD-defined fields. These fields that are in the DBD are marked in the DLIModel IMS Java Report as being either primary key fields or search fields.

You cannot use parentheses in the WHERE clause because SSAs do not support parentheses.

Fields in the WHERE clause can be compared only to values, not to other fields. You can use the following operators between field names and values in the individual qualification statements:

- `<`
- `<=`
- `=`
- `<>`
- `<`
- `!=`

For example, the following WHERE clause will fail because it is trying to compare two fields:

```
WHERE Sales.SoldBy=Sales.PurchaserFirstName
```
The following example is valid because the WHERE clause is comparing a field to a value:
WHERE Sales.SoldBy='Lauren'

When using prepared statements, you can use the question mark (?) character, which is later filled in with a value. For example, the following WHERE clause is valid:
WHERE Sales.SoldBy= ?

You can combine multiple qualification statements with AND and OR operators, but you must follow special rules. Because separate SSAs are created for each segment, list all qualification statements for a segment together and combine qualification statements for different segments with an AND operator.

Qualification statements that are combined with an AND operator make up a qualification set. For a qualification set to be satisfied (true), all qualification statements in the set must be satisfied. For the WHERE clause (and, therefore, the SSA qualification) to be satisfied, at least one qualification set must be satisfied.

The OR operator can be used only between qualification statements that contain fields from the same segment. Because of the way SSA lists are created, you cannot use the OR operator across segments. For example, the following WHERE clause will fail because the Soldby field and DealerName fields are in different segments:
WHERE Sales.SoldBy='Kiran' OR Dealer.DealerName='Bach'

However, the following WHERE clause is valid because the OR operator is between two qualification statements for the same segment:
WHERE Sales.SoldBy='Kyle' OR Sales.PurchaserFirstName='Chris'

**Summary of WHERE Clause Usage**
When using the WHERE clause in SQL calls to IMS databases:
- Use fields that are in any segment from the root segment down to the segment in the FROM clause.
- Qualify fields with segment names.
- Compare fields to values, not other fields.
- Do not use parentheses.
- List all qualification statements for a segment together.
- Combine qualification statements for different segments with an AND operator.
- Do not use the OR operator across segments.

**Supported SQL Aggregate Functions**
IMS Java supports the following SQL aggregate functions and related keywords:
- AS
- ASC
- AVG
- COUNT
- DESC
- GROUP BY
- MAX
- MIN
- ORDER BY
- SUM
SQL Aggregate Functions

**Important:** The field names that are specified in a GROUP BY or ORDER BY clause must match exactly the field name that is specified in the SELECT statement.

The supported SQL aggregate functions accept only a single field name in a segment as the argument (the DISTINCT keyword is not allowed). Table 2 shows the data types of the fields that are accepted by the aggregate functions, along with the resulting data type in the result set.

### Table 2. Supported SQL Aggregate Functions and Their Supported Data Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM and AVG</td>
<td>Byte</td>
<td>Long</td>
</tr>
<tr>
<td></td>
<td>Short</td>
<td>Long</td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>Long</td>
</tr>
<tr>
<td></td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td></td>
<td>Single-precision floating point</td>
<td>Double-precision floating point</td>
</tr>
<tr>
<td></td>
<td>Double-precision floating point</td>
<td>Double-precision floating point</td>
</tr>
<tr>
<td>MIN and MAX</td>
<td>Any type except BIT, BLOB, or BINARY</td>
<td>Same as argument type</td>
</tr>
<tr>
<td>COUNT</td>
<td>Any type</td>
<td>Long</td>
</tr>
</tbody>
</table>

The result set column name from an aggregate function is a combination of the aggregate function name and the field name separated by an underscore character (_). For example, the statement `SELECT MAX(age)` results in a column name `MAX_age`. Use this column name in all subsequent references—for example, `resultSet.getInt("MAX_age")`.

If the aggregate function argument field is segment-qualified, the result-set column name is the combination of the aggregate function name, the segment name, and the field name, separated by underscore characters (_). For example, `SELECT MAX(Employee.age)` results in a column name `MAX_Employee_age`.

You can use the AS keyword to rename the aggregate function column in the result set or any other field in the SELECT statement. You cannot use the AS keyword to rename a segment in the FROM clause. When you use the AS keyword to rename the field, you must use this new name to refer to the field. For example, if you specify `SELECT MAX(age) AS oldest`, a subsequent reference to the aggregate function column is `resultSet.getInt("oldest")`.

The result set type for aggregate functions and ORDER BY and GROUP BY clauses is always TYPE_SCROLL_INSENSITIVE, even if they are defined explicitly as TYPE_FORWARD_ONLY. A TYPE_SCROLL_INSENSITIVE result set is not sensitive to any changes in the database when the result set is open.

### Supported JDBC Interfaces

The following list describes the required interfaces by JDBC 2.1 that are implemented in the database package, and it describes the limitations in the IMS Java implementation of these interfaces.

**java.sql.Connection**

`java.sql.Connection` is an object that represents the connection to the database. A Connection reference is retrieved from the DriverManager.
object that is implemented in the java.sql package. The DriverManager object obtains a Connection reference by querying its list of registered Driver instances until it finds one that supports the universal resource locator (URL) that is passed to the DriverManager.getConnection method.

**Restriction:** IMS does not support the local, connection-based commit scope that is defined in the JDBC model. Therefore, the IMS Java implementation of the methods Connection.commit, Connection.rollback, and Connection.setAutoCommit result in an SQL exception when these methods are called.

Figure 29 shows the sample dealership application code that establishes a connection to the sample database:

```java
connection = DriverManager.getConnection("jdbc:dli:dealership.application.DealerDatabaseView");
```

**Figure 29. Establishing a Connection to the Dealership Database**

**java.sql.DatabaseMetaData**

The DatabaseMetaData interface defines a set of methods for querying information about the database, including capabilities the database might or might not support. The class is provided for tool developers and is normally not used in client programs. Much of the functionality is specific to relational databases and is not implemented for DL/I databases.

**java.sql.Driver**

The Driver interface itself is not usually used in client applications, although an application must dynamically load a particular Driver implementation by name. One of the first lines in an IMS JDBC program for IMS access must be:

```java
Class.forName("com.ibm.ims.db.DLIDriver");
```

This code loads the IMS Java driver and causes the Driver implementation to register itself with the DriverManager object so that the driver can later be found by DriverManager.getConnection. The Driver implementation creates and returns a Connection object to the DriverManager object. The IMS Java implementation of JDBC is not fully JDBC-compliant and the Driver object method jdbcCompliant returns a value of false.

**java.sql.Statement**

A Statement interface is returned from the Connection.createStatement method. The Statement class and its subclass, PreparedStatement, define the interfaces that accept SQL statements and return tables as ResultSet objects. The code to create a Statement object is as follows:

```java
Statement statement = connection.createStatement();
```

**Restriction:** The IMS Java implementation of the Statement interface does not support:

- Named cursors. Therefore, the method Statement.setCursorName throws an SQL exception.
- Aborting a DL/I operation. Therefore, the method Statement.cancel throws an SQL exception.
- Setting a time-out for DL/I operations. Therefore, the methods Statement.setQueryTimeout and Statement.getQueryTimeout throw SQL exceptions.
java.sql.ResultSet

The ResultSet interface defines an iteration mechanism to retrieve the data in the rows of a table, and to convert the data from the type defined in the database to the type required in the application. For example, ResultSet.getString converts an integer or decimal data type to an instance of a Java String. The code to return ResultSet object is as follows:

```
ResultSet results = statement.executeQuery(queryString);
```

Rather than building a complete set of results after a query is run, the IMS Java implementation of ResultSet interface retrieves a new segment occurrence each time the method ResultSet.next is called.

Restriction: The IMS Java implementation of ResultSet does not support:
- Returning data as an ASCII stream. Therefore the method ResultSet.getAsciiStream throws an SQL exception.
- Named cursors. Therefore the method ResultSet.getCursorName throws an SQL exception.
- The method ResultSet.getUnicodeStream, which is deprecated in JDBC 2.1.

java.sql.ResultSetMetaData

The java.sql.ResultSetMetaData interface defines methods to provide information about the types and properties in a ResultSet object. It includes methods such as getColumnCount, isSigned, getPrecision, and getColumnName.

java.sql.PreparedStatement

The PreparedStatement interface extends the Statement interface, adding support for pre-compiling an SQL statement (the SQL statement is provided at construction instead of execution), and for substituting values in the SQL statement (for example, UPDATE Suppliers SET Status = ? WHERE City = ?).

JDBC Prepared Statements for SQL

To improve performance of your IMS Java application, use JDBC prepared statements for the SQL. The PreparedStatement class completes the initial steps in preparing queries only once so that you need to provide the parameters only before each repeated database call.

The PreparedStatement object performs the following actions only once before repeated database calls are made:
1. Parses the SQL.
2. Cross-references the SQL with the IMS Java DLIDatabaseView object.
3. Builds SQL into SSAs before a database call is made.

Supported JDBC Data Types

IMS Java supports the JDBC data types that are listed in Table 3 on page 87. The DLIModel IMS Java Report indicates the JDBC type that is assigned to each field in the DLIDatabaseView subclass. Table 3 on page 87 also lists the supported Java data types for each JDBC type.
Table 3. Supported JDBC Data Types

<table>
<thead>
<tr>
<th>JDBC Data Type</th>
<th>Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>BIT</td>
<td>boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>PACKEDDECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>ZONEDDECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
</tbody>
</table>

Table 4 shows the get methods that are available for accessing different types of JDBC data.

The methods that are marked with “X” are methods that are designed for accessing the given data type. No truncation or data loss occurs when you use those methods. The methods that are marked with “O” are all other legal calls. Data integrity is not be ensured when you use these methods. If the box is does not contain an “X” or an “O”, using that get method for that data type results in an exception.

Table 4. ResultSet.getxxx Methods to Retrieve JDBC Types

<table>
<thead>
<tr>
<th>ResultSet.getxxx Methods</th>
<th>JDBC Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>getByte</td>
<td>X O O O O O O O O O</td>
</tr>
<tr>
<td>getShort</td>
<td>O X O O O O O O O O</td>
</tr>
<tr>
<td>getInt</td>
<td>O O X O O O O O O O</td>
</tr>
<tr>
<td>getLong</td>
<td>O O O X O O O O O O</td>
</tr>
<tr>
<td>getFloat</td>
<td>O O O O X O O O O O</td>
</tr>
<tr>
<td>getDouble</td>
<td>O O O O O X O O O O</td>
</tr>
<tr>
<td>getBoolean</td>
<td>O O O O O X O O O O</td>
</tr>
<tr>
<td>getString</td>
<td>O O O O O O O O X O</td>
</tr>
<tr>
<td>getBigDecimal</td>
<td>O O O O O O O O X X</td>
</tr>
</tbody>
</table>
### Supported JDBC Data Types

**Table 4. ResultSet.getxxx Methods to Retrieve JDBC Types** (continued)

<table>
<thead>
<tr>
<th>ResultSet.getxxx Methods</th>
<th>TINYINT</th>
<th>SMALLINT</th>
<th>INTEGER</th>
<th>BIGINT</th>
<th>FLOAT</th>
<th>DOUBLE</th>
<th>BIT</th>
<th>VARCHAR</th>
<th>PACKEDDECIMAL</th>
<th>ZONEDDECIMAL</th>
<th>BINARY</th>
<th>DATE</th>
<th>TIME</th>
<th>TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBytes</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getDate</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>getTime</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>getTimestamp</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O</td>
</tr>
</tbody>
</table>

**Notes:**

1. PACKEDDECIMAL and ZONEDDECIMAL are IMS Java JDBC types. All other types are standard SQL types defined in SQL92. **Restriction:** PACKEDDECIMAL and ZONEDDECIMAL data types do not support the Sign Leading or Sign Separate modes. For these two data types, sign information is always stored with the Sign Trailing method.

If the field type is either PACKEDDECIMAL or ZONEDDECIMAL, the type qualifier is the PICTURE string that represents the layout of the field. All COBOL PICTURE strings that contain valid combinations of 9s, Ps, Vs, and Ss are supported. Expansion of PICTURE strings is handled automatically. For example, '9(5)' is a valid PICTURE string. For zoned decimal numbers, the decimal point can also be used in the PICTURE string.

If the field contains DATE, TIME, or TIMESTAMP data, the type qualifier specifies the format of the data. For example, a type qualifier of ddMMyyyy indicates that the data is formatted as follows:

11122004 is December 11, 2004

For DATE and TIME types, all formatting options in the java.text.SimpleDateFormat class are supported.

For the TIMESTAMP type, the formatting option 'f' is available for nanoseconds. TIMESTAMP can contain up to nine 'f's and replaces the 'S' options for milliseconds. Instead, 'fff' indicates milliseconds of precision. An example TIMESTAMP format is as follows:

`yyyy-mm-dd hh:mm:ss.fffffffff`

### General Mappings from COBOL Copybook Types to IMS Java and Java Data Types

**Table 5** describes how COBOL copybook types are mapped to DLITypeInfo constants and Java data types.

**Table 5. Mapping from COBOL Formats to DLITypeInfo Constants and Java Data Types**

<table>
<thead>
<tr>
<th>Copybook Format</th>
<th>DLITypeInfo Constant</th>
<th>Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X</td>
<td>CHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>
Table 5. Mapping from COBOL Formats to DLITypeInfo Constants and Java Data Types (continued)

<table>
<thead>
<tr>
<th>Copybook Format</th>
<th>DLITypeInfo Constant</th>
<th>Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC 9 BINARY(^1)</td>
<td>See Table 6 (^2)</td>
<td>See Table 6 (^2)</td>
</tr>
<tr>
<td>COMP-1</td>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>COMP-2</td>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>PIC 9 COMP-3(^3)</td>
<td>PACKEDDECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>PIC 9 DISPLAY(^4)</td>
<td>ZONEDDECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
</tbody>
</table>

**Notes:**

1. Synonyms for BINARY data items are COMP and COMP-4.
2. For BINARY data items, the DLITypeInfo constant and Java type depend on the number of digits in the PICTURE clause. Table 6 describes the type based on PICTURE clause length.
3. PACKED-DECIMAL is a synonym for COMP-3.
4. If the USAGE clause is not specified at either the group or elementary level, it is assumed to be DISPLAY.

Table 6 shows the DLITypeInfo constants and the Java data types based on the PICTURE clause.

Table 6. DLITypeInfo Constants and Java Data Types Based on the PICTURE Clause

<table>
<thead>
<tr>
<th>Digits in PICTURE Clause</th>
<th>Storage Occupied</th>
<th>DLITypeInfo Constant</th>
<th>Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 through 4</td>
<td>2 bytes</td>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>5 through 9</td>
<td>4 bytes</td>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>10 through 18</td>
<td>8 bytes</td>
<td>BIGINT</td>
<td>long</td>
</tr>
</tbody>
</table>

Table 7 shows examples of specific copybook formats mapped to DLITypeInfo constants.

Table 7. Copybook Formats Mapped to DLITypeInfo Constants

<table>
<thead>
<tr>
<th>Copybook Format</th>
<th>DLITypeInfo Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X(25)</td>
<td>CHAR</td>
</tr>
<tr>
<td>PIC S9(04) COMP</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>PIC S9(06) COMP-4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>PIC S9(12) BINARY</td>
<td>BIGINT</td>
</tr>
<tr>
<td>COMP-1</td>
<td>FLOAT</td>
</tr>
<tr>
<td>COMP-2</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>PIC S9(06)V99</td>
<td>ZONEDDECIMAL</td>
</tr>
<tr>
<td>PIC 9(06).99</td>
<td>ZONEDDECIMAL</td>
</tr>
<tr>
<td>PIC S9(06)V99 COMP-3</td>
<td>PACKEDDECIMAL</td>
</tr>
</tbody>
</table>
**JDBC Recommendations for IMS Databases**

Although the JDBC interface to an IMS database closely follows the relational database paradigm, the segments are physically stored in a hierarchical database, which affects the semantics of your JDBC calls to some extent. To avoid unexpected results or potential performance problems, follow these recommendations:

- When you code a SELECT list, generally try to supply predicates in the WHERE clause for all levels down the hierarchy to your target segment.
  
  If you supply a predicate in the WHERE clause for a target segment somewhere down the hierarchy and omit predicates for its parents, IMS must scan all candidate segments at the parent levels in an attempt to match the predicate that you supplied. For example, if you are retrieving a second-level segment and you supply a predicate for that second-level segment, but do not supply one for the root segment, IMS might perform a full database scan, testing every second-level segment under every root against the predicate. This has performance implications, particularly at the root level, and also might result in unexpected segments being retrieved. A similar consideration applies to locating segments for UPDATE clauses.

- When you insert a new segment, generally try to supply predicates in the WHERE clause for all levels down the hierarchy to your target new segment.
  
  If you omit a predicate for any level down to the insert target segment, IMS chooses the first occurrence of a segment at that level that allows it to satisfy remaining predicates, and performs the insert in that path. This might not be what you intended. For example, in a three-level database, if you insert a third-level segment, and supply a predicate for the root but none at the second-level, your new segment will always be inserted under the first second-level segment under the specified root.

- If you delete a segment that is not a bottom-level (leaf) segment in its hierarchy, you also delete the remaining segments in that hierarchical subtree. The entire family of segments of all types that are located hierarchically below your target deleted segment are also usually deleted.

- When you provide predicates to identify a segment, the search is generally faster if the predicate is qualified on a primary or secondary index key field, rather than simply on a search field. Primary and secondary key fields are identified for each segment in the DLIModel IMS Java Report.

**Java Metadata Classes for IMS Databases**

To access a set of IMS databases using JDBC, you must describe to IMS Java the application’s view of the databases. The application view information is in the program specification block (PSB), but you must first convert this information into a form that you can use in your Java application: a subclass of com.ibm.ims.db.DLIDatabaseView. This subclass is called the IMS Java metadata class. When you establish the JDBC database connection, you pass the name of this class to IMS Java.

Create the metadata class for a PSB by providing the application PSB source and related DBD source files to the DLIModel utility so that the utility can generate the IMS Java metadata class. The DLIModel utility is described in Chapter 7, “DLIModel Utility,” on page 97.
The examples used throughout this chapter are based on the sample application. The PSB for the sample dealership application is shown in Figure 30.

```
DLR_PCB1 PCB TYPE=DB,DBDNAME=DEALERDB,PROCOPT=GO,KEYLEN=42
SENSEG NAME=DEALER,PARENT=0
SENSEG NAME=MODEL,PARENT=DEALER
SENSEG NAME=ORDER,PARENT=MODEL
SENSEG NAME=SALES,PARENT=MODEL
SENSEG NAME=STOCK,PARENT=MODEL
PSBGEN PSBNAME=DLR_PSB,MAXQ=200,LANG=JAVA
END
```

Figure 30. Sample PSB for the Sample Dealership Application

The physical DBD that is referenced by the PSB in Figure 30 is shown in Figure 31.

```
DBD NAME=DEALERDB,ACCESS=(HDAM,OSAM),RMNAME=(DFSHDC40.1.10)
SEGM NAME=DEALER,PARENT=0,BYTES=94
  FIELD NAME=(DLRNO,SEQ,U),BYTES=4,START=1,TYPE=C
  FIELD NAME=DLRNAME,BYTES=30,START=5,TYPE=C
SEGM NAME=MODEL,PARENT=DEALER,BYTES=43
  FIELD NAME=(MODTYPE,SEQ,U),BYTES=2,START=1,TYPE=C
  FIELD NAME=MAKE,BYTES=10,START=3,TYPE=C
  FIELD NAME=MODEL,BYTES=10,START=13,TYPE=C
  FIELD NAME=YEAR,BYTES=4,START=23,TYPE=C
  FIELD NAME=MSRP,BYTES=5,START=27,TYPE=P
SEGM NAME=ORDER,PARENT=MODEL,BYTES=127
  FIELD NAME=(ORDNBR,SEQ,U),BYTES=6,START=1,TYPE=C
  FIELD NAME=LASTNME,BYTES=25,START=50,TYPE=C
  FIELD NAME=FIRSTNME,BYTES=25,START=75,TYPE=C
SEGM NAME=SALES,PARENT=MODEL,BYTES=113
  FIELD NAME=(SALDATE,SEQ,U),BYTES=8,START=1,TYPE=C
  FIELD NAME=LASTNME,BYTES=25,START=9,TYPE=C
  FIELD NAME=FIRSTNME,BYTES=25,START=34,TYPE=C
  FIELD NAME=STKVIN,BYTES=20,START=94,TYPE=C
SEGM NAME=STOCK,PARENT=MODEL,BYTES=62
  FIELD NAME=(STKVIN,SEQ,U),BYTES=20,START=1,TYPE=C
  FIELD NAME=COLOR,BYTES=10,START=37,TYPE=C
  FIELD NAME=PRICE,BYTES=5,START=47,TYPE=C
  FIELD NAME=LOT,BYTES=10,START=52,TYPE=C
DBDGEN
FINISH
END
```

Figure 31. DBD for the Sample Dealership Database

The DLIModel utility generates a subclass of DLIDatabaseView from the PSB and DBD. It also produces a report, called the DLIModel IMS Java Report, that provides information about the metadata class. Figure 32 on page 92 shows an example of a DLIModel IMS Java Report.

The report supplements the information in the generated metadata class and the original PSB and DBD source files. Use this information when you write JDBC calls to IMS databases.
Java Metadata Classes for IMS

DLIModel IMS Java Report

Class: DealerDatabaseView in package: com.ibm.ims.tooling generated for PSB: AUTPSB11

PCB: DealershipDB

Segment: Dealer
Field: DealerNumber Type=CHAR Length=4 ++ Primary Key Field ++
Field: DealerName Type=CHAR Length=30
Field: DealerAddress Type=CHAR Length=50
Field: YTDSales Type=PACKEDDECIMAL Type Qualifier=S9(18)

Segment: Model
Field: ModelTypeCode Type=CHAR Length=2 ++ Primary Key Field ++
Field: CarMake Type=CHAR Length=10 (Search Field)
Field: CarModel Type=CHAR Length=10 (Search Field)
Field: CarYear Type=CHAR Length=4 (Search Field)
Field: Price Type=CHAR Length=5 (Search Field)
Field: EPACityMileage Type=CHAR Length=4
Field: EPAHighwayMileage Type=CHAR Length=4
Field: HorsepowerType Type=CHAR Length=4

Segment: Order
Field: OrderNumber Type=CHAR Length=6 ++ Primary Key Field ++
Field: PurchaserLastName Type=CHAR Length=25 (Search Field)
Field: PurchaserFirstName Type=CHAR Length=25 (Search Field)
Field: Options Type=CHAR Length=30
Field: Price Type=ZONEDDECIMAL Type Qualifier=99999
Field: OrderDate Type=CHAR Length=8
Field: SerialNo Type=CHAR Length=8
Field: DeliverDate Type=CHAR Length=8

Segment: Sales
Field: DateSold Type=CHAR Length=8 ++ Primary Key Field ++
Field: PurchaserLastName Type=CHAR Length=25 (Search Field)
Field: PurchaserFirstName Type=CHAR Length=25 (Search Field)
Field: StockVINumber Type=CHAR Length=20 (Search Field)
Field: PurchaserAddress Type=CHAR Length=25 (Search Field)
Field: SoldBy Type=CHAR Start=84 Length=10

Segment: Stock
Field: StockVINumber Type=CHAR Length=20 ++ Primary Key Field ++
Field: Color Type=CHAR Length=10 (Search Field)
Field: Price Type=ZONEDDECIMAL Type Qualifier=99999
Field: Lot Type=CHAR Length=10 (Search Field)
Field: DateIn Type=CHAR Length=8
Field: DateOut Type=CHAR Length=8

Figure 32. Sample DLIModel IMS Java Report for the Dealership Sample Database

The DLIModel IMS Java Report provides you with the following information:

- The name of the metadata class (DealerDatabaseView in this example) to use when you establish a connection to the database.
- The hierarchy of segments for each PCB.
- The fields within each segment, which are specified by the DBD, by any COBOL copybooks, or by control statements. For example, the fields DealerAddress and YTDSales in the Dealer segment are added fields.
- The names of PCBs, segments, and fields to use in your JDBC calls. These names may be alias names that are assigned to the IMS entities. Alias names are intended to be more representative and intuitive identifiers for your Java application to use rather than the 8-character names in the PSB and DBDs. In
the example, the name DealershipDB replaces the PCB name DLR_PCB1 from the PSB. A comparison of the names of the segments and the fields in the report with their names in the DBD shows that they have all been assigned more meaningful names.

- The data types of the fields. The data types of the fields are based on the simple TYPE property of the fields in the DBD and the DLIModel utility control statements. For example, the field YTDSales in the Dealer segment is type PACKEDDECIMAL with a type qualifier (format descriptor) of S9(18).
- The fields in each segment, which are identified as primary or secondary index fields, search fields, or other fields.

Sample Application that Uses JDBC

Because IMS is a hierarchical database, IMS Java does not fully implement the JDBC API. This section describes the IMS Java implementation of JDBC with a sample application.

To use JDBC to read, update, insert, and delete segment instances, an application must:

1. Obtain a connection to the database. Load the DLIDriver and retrieve a Connection object from the DriverManager.
2. Retrieve a Statement or PreparedStatement object from the Connection object and execute it. An example of this step is in Figure 33 on page 94.
3. Iterate the ResultSet object returned from the Statement or PreparedStatement object to retrieve specific field results. An example of this step is in Figure 33 on page 94.

Figure 33 on page 94, which is part of a sample method showModelDetails, obtains a Connection object, retrieves a PreparedStatement object, makes SQL calls to the database, and then iterates the ResultSet object that is returned from the PreparedStatement object.
Sample JDBC Application

```java
public ModelDetailsOutput showModelDetails(ModelDetailsInput input)
throws NamingException, SQLException, IMSException {

    // Extract the key from the input
    String modelKey = input.getModelKey();
    ModelDetailsOutput output = new ModelDetailsOutput();

    // Validate the key
    if (modelKey != null && !modelKey.trim().equals("")) {

        // Build the SQL query.
        String query = "SELECT * FROM Dealer.ModelSegment WHERE " + "ModelSegment.ModelKey = " + input.getModelKey() + ";";

        // Execute the query
        Statement statement = connection.createStatement();
        ResultSet results = statement.executeQuery(query);

        // Store the results in the output object and send it back to the caller of this method.
        if (results.next()) {
            output.setMake(results.getString("Make"));
            output.setModelType(results.getString("ModelType"));
            output.setModel(results.getString("Model"));
            output.setYear(results.getString("Year"));
            output.setPrice(results.getString("MSRP"));
            output.setCount(results.getString("Counter"));
        }
    }

    return output;
}
```

*Figure 33. Example JDBC Application*

Imported Packages for JDBC Access to IMS Databases

To use unqualified class names instead of fully-qualified names in your program, include import statements at the top of the Java file.

Use the following import statement to make IMS database access classes available by their unqualified class names:

```java
import com.ibm.ims.db.*;
```

Use the following import statement to make JDBC classes available by their unqualified class names:

```java
import java.sql.*;
```

Connections to IMS Databases

Provide the name of the DLIDatabaseView subclass when retrieving a JDBC Connection object.

When the following code is executed, DLIDriver, a class in com.ibm.ims.db, registers itself with the JDBC DriverManager object:

```java
Class.forName("com.ibm.ims.db.DLIDriver");
```

When the following code is executed, the JDBC DriverManager object determines which of the registered drivers supports the supplied string:

```java
```
connection = DriverManager.getConnection("jdbc:dli:dealership.application.DealerDatabaseView");

Because the supplied string begins with jdbc:dli:, the JDBC DriverManager object locates the DLIDriver instance and requests that it create a connection.
Sample JDBC Application
Chapter 7. DLIModel Utility

This chapter contains information on the DLIModel utility including introductory and practical usage information.

In this chapter:
- “DLIModel Utility Overview”
- “PSB and DBD Requirements” on page 99
- “DLIModel Utility Restrictions” on page 99
- “Running the DLIModel Utility” on page 110
- “Control Statements for the DLIModel Utility” on page 102
- “Output Types of the DLIModel Utility” on page 100
- “Examples of Using the DLIModel Utility” on page 113

DLIModel Utility Overview

Processing IMS databases with an IMS Java application requires that you describe the database view of your application’s PSB to IMS Java. You must do this by providing the name of a metadata class when establishing the JDBC database connection. Use the DLIModel utility to prepare the metadata class for an application.

You can use the DLIModel utility to:
- Create IMS Java metadata classes to describe a PSB’s view of IMS databases, from PSB and DBD source.
- Incorporate additional field information from XMI input files that describe COBOL copybook members.
- Incorporate additional PCB, segment, and field information, or overrides of existing information, into the generated class from user-prepared input control statements.
- Create a DLIModel Java Report (designed to assist Java application programmers), which describes the IMS Java view of the PSB and its databases.
- Create an XMI description of the PSB and its databases.

The DLIModel utility can process most types of PSBs and databases. For example, IMS Java supports:
- All database organizations except MSDB, HSAM, SHSAM, and GSAM
- All types and implementations of logical relationships
- Secondary indexes except for shared secondary indexes
- Secondary indexes processed as stand-alone databases
- PSBs that specify field-level sensitivity

Figure 34 on page 98 shows the inputs and outputs of the DLIModel utility. The actions of the utility are directed by control statements that you supply. PSB and DBD source members are read from their PDS or PDSE data sets and parsed by the utility to build an in-memory object model of the database structure and the PSB’s view of that structure. Multiple PSBs may be processed in a single run of the utility.
The control statements can specify:

- Which PSBs to process in this run
- Aliases for PSBs, PCBs, segments, and fields
- Data types and format masks for fields
- XMI files that contain XMI descriptions of COBOL copybook members corresponding to segments
- Additional field definitions for fields that were not defined in the DBD or COBOL copybook XMI file
- Information that overrides PSB, DBD, and COBOL copybook XMI information
- Default values for newly inserted segments

When these inputs have all been processed and incorporated into the model, the utility generates various outputs that were requested through control statements. You can request to have an IMS Java metadata class be generated for each PSB processed, together with a corresponding easy-to-read DLIModel Java Report for the Java programmer to use.
You can request an XMI description of the entire in-memory model (one description covers all PSBs and DBDs processed in the run). For details of this XMI output, see "XMI Description of the Databases" on page 101.

You can also request a detailed trace file of the utility execution if one is necessary for problem resolution.

### PSB and DBD Requirements

The following are the requirements to run the DLIModel utility.

- PCBs in the PSB must be named, either through statement labels or the PCBNAME parameter.
- If your application uses JDBC and the field list in a call includes fields from more than one segment in a hierarchic path, then IMS Java employs path calls. In this case you must include P as a processing option (PROCOPT) in the PCB or SENSEG statements, as appropriate.
- If your application uses SSA database access, path calls are under your control, and you must choose PSB processing options depending on your processing, in the usual way.
- This utility will not validate the PSB and DBD source. IBM strongly recommends that you generate DBDs, PSBs, and ACBs, correct all errors, and then run the DLIModel utility.
- This utility follows all inter-DBD references when building its model, and may require access to DBDs that are not directly referenced by PCBs in the PSB. For example, when processing a PSB that references a main database with a number of secondary indexes, DLIModel needs access to the secondary index DBDs even if the PSB does not explicitly name any of these indexes for a secondary processing sequence, or for segment search purposes. Similarly, all DBDs related by logical relationships must be accessible.
- You must maintain the length field in variable length segments on INSERT or UPDATE.
- XMI input files must conform to the COBOL metamodel, which is part of the CAM metamodel of the OMG-accepted version of the UML specification for the Enterprise Application Integration (EAI) standard.

### DLIModel Utility Restrictions

The DLIModel utility has the following restrictions.

- The utility cannot process:
  - MSDB, HSAM, SHSAM, and GSAM databases
  - Shared secondary indexes
  - PROCOPT=K option in a PSB SENSEG
- The DLIModel utility does not use DLITypeInfoList classes in its generated classes. If you want to define repeating groups of fields in segments (other than by explicitly defining each group of fields separately) you will have to create the classes manually or modify the classes generated by DLIModel.
- COBOL copybook XMI files, which supply additional information about field layouts, must describe the physical segments. The files cannot describe the logical database segment layouts.
- The default data type for all fields is CHAR, even if the DBD specifies a different data type. To change the data type of a field, use the FIELD control statement.
Output Types of the DLIModel Utility

The DLIModel utility can generate the following output:

- IMS Java Metadata Classes
- DLIModel Java Report
- XMI Description of the Databases
- DLIModel Trace

IMS Java Metadata Classes

The DLIModel utility produces the necessary metadata classes need to develop IMS Java applications. However, the Java developer needs only to reference the DLIModel Java Report for information about the classes.

DLIModel Java Report

The DLIModel Java Report summarizes the structure of the IMS databases in a way that allows you to create IMS Java applications and to code SQL queries against the databases. With the DLIModel Report, you do not have to interpret the syntax of the IMS Java classes or refer to the DBD or PSB source.

Related Reading: Sample DLIModel Java Reports are shown in each of the four examples in "Examples of Using the DLIModel Utility" on page 113.

PSB Description

Within the DLIModel Java Report, a separate section is produced for each PSB named in a PSB statement of the control data set. The name of the generated class for the PSB is given first, which is either the name defined by the JavaName parameter or, if no JavaName is specified, the 8-character IMS PSB name. The report also gives the IMS PSB name and the package for this class, if one was specified in the OPTIONS control statement.

PCB Description

Within each PSB, sections are listed for each PCB. Each PCB is identified by its IMS Java name, which is either the JavaName, if one was specified in the control data set, or the 8-character IMS PCB name.

Use the PCB name in SQL queries to the database. In the SQL queries, the PCB name is equivalent to a table designator.

Segment Description

Within each PCB, all segments are listed in hierarchical sequence. Segment descriptions are indented to illustrate the hierarchical dependencies. Each segment is identified by its IMS Java name, which is either the JavaName, if one was specified in the control data set, or the 8-character IMS Segment name. Use the IMS Java name for the segment in SQL queries to the database.

Use the IMS Java name for the segment in SQL queries to the database. In the SQL queries, the segment name is equivalent to a table name.

Field Description

Within each segment, fields are listed in the order in which they appear in the database DBD with any additional fields appended. Fields are of the following types:

Field that is physically resident in a DBD

A field that is physically resident in a DBD is identified by its IMS Java name.
name, which is either the user-chosen JavaName, if one was specified, or the 8-character IMS Field name. This is the name that should be used in SQL queries. A DBD field is further annotated as either a ++ Primary Key Field ++ if it is the sequence field of its segment, or a (Search field) if it is a non-sequence field. SQL queries with WHERE clauses qualified on Primary Key Fields will generally produce much faster response times than calls qualified on search fields, but both are allowed.

These fields have their IMS Java type listed, and if necessary, their type qualifier string. They also have their Start position and Bytes listed. It is important to note that the Start and Bytes values describe the field’s properties in the original database segment, not necessarily its Start or Bytes as viewed from Java application. The primary purpose of the Start and Bytes values is to describe how these fields overlap or redefine each other in the database. They are otherwise unnecessary for writing SQL queries in the Java application.

**DBD secondary index search field**

A DBD secondary index search field (a field defined in the DBD with an XDFLD macro) is also identified through its IMS Java name, either user-chosen, or the DBD name. The field is annotated as ++ Secondary Key Field ++, and like a ++Primary Key Field ++ will produce fast responses to queries. However, secondary index search fields are not physically present in their segment and can not be retrieved from the Java ResultSet. In the report, secondary index search fields are followed by a list of their component search fields to assist you in creating a suitable string to use as a search argument in an SQL query.

A secondary index field has no Start or Bytes value in the segment. It is essentially a virtual field and is used for search purposes only.

**Field that is not present in the DBD**

A field that is not present in the DBD is identified (for example, one that has been added by a Field control statement or by an XMI COBOL copybook description) by its Java name, if one is present, or by its 8-character name. Its start position, length, data type, and type qualifier are all listed. It has no key field or search field annotation in the report, indicating that it may not be used in an SQL WHERE clause. However, it may be retrieved from the result set following successful queries.

**XMI Description of the Databases**

An XMI file, written in UTF-8 encoding, is produced by the utility if you specify genXMI=YES in the OPTIONS control statement. It describes all of the PCBs and their referenced DBDs processed in the entire run of the utility.

Samples of the XMI produced (converted from UTF-8 to EBCDIC encoding for viewing in an OS/390 environment) for each of the samples in this chapter are in the samples directories.

The XMI that is produced by the utility is based on a metamodel of IMS database defined in UML. This model is a package with a number of inheritance relationships to the OMG Common Warehouse Metamodel (CWM). However, only the IMS package itself is included and used in the DLIModel utility.

Directory /usr/lpp/ims/imaJava81/dlimodel/samples/model of the distribution media contains:
DLIModel Utility Outputs

- An EBCDIC-encoded XMI definition of the meta-model to view in an OS/390 environment.
- An IBM Rational Rose® model file of the meta-model. This model file is at the 4.5/6.0 Model level, corresponding to Rose 98 or 98i. To view this file, you need a licensed and installed copy of a suitable level of the Rational Rose product.

DLIModel Trace

The DLI Model utility can generate a trace file, named dlimodeltrace, if you need to resolve a problem with the utility. For the utility to generate the trace file, specify GenTrace=YES in the OPTIONS statement. You can also specify the path where the file is written by using the TracePath parameter.

Control Statements for the DLIModel Utility

You must write control statements to specify certain options such as input and output data set names, and what PSBs and PCBs to use. You can also use the control statements to supply information to the utility about PSBs, PCBs, segments, and fields that cannot be extracted from the PSBs, DBDs, or COBOL copybook XMI files.

The control statements are supplied to the utility in a PDS member named in the EXEC statement PARM field of the MVS™ JCL, or in an HFS file named in a command line parameter in the UNIX System Services environment.

Control Data Set Rules

You must include at least the following statements in your control statement data set:

- OPTIONS Statement
- PSB Statement for each PSB to be processed

Optionally, you can include the following statements in the control data set:

- PCB Statement
- SEGM Statement
- FIELD Statement
- XDFLD Statement
- INCLUDE Statement

The following syntax diagram shows how to organize the control statements in the control data set.
If you requested IMS Java metadata class source in the OPTIONS statement, each PSB that you specified results in a separate metadata JAVA file, and a corresponding DLIModel Java Report.

A typical reason to include PCB, SEGM, and FIELD statements is to assign to these entities a customized name (referred to as an alias) that can be used in your Java program. You can choose a name that is more meaningful than the 8-character name given to these entities in the DBD and PSB source. You might also need to assign data types to fields, and to define additional fields that are important to your application but that were not defined in the segment in the DBD.

You do not need to include PCB, SEGM, or FIELD statements in your control statement set if all of the following statements are true of your application:

- It can process PCBs, segments and fields by their 8-character IMS names.
- It needs only fields that are defined in the DBD.
- All fields can be processed as data type CHAR.

**Related Reading:** For examples of control statement sets, refer to the examples in “Examples of Using the DLIModel Utility” on page 113.

The rules for ordering the control statements are as follows:

- The OPTIONS statement must be first and only be present in a top-level control data set.
- PCB statements must follow immediately after the PSB statement to which they belong. They may be in any order (for example, PCB statements need not be in the same order as they appear in the original PSB source).
- FIELD statements must follow immediately after the SEGM for the physical segment to which they belong. However, Field statements may be in any order within their segment group. For example, field statements need not be in the same sequence as they appear in the original DBD source. FIELD statements for existing fields and for new fields may be intermixed or grouped in any sequence.
- INCLUDE statements can be positioned anywhere in a control data set, but not between:
  - PSB statement and any PCB statements that belong to it
  - SEGM statement and any FIELD statements that belong to it

You can nest multiple control data sets by using the INCLUDE statement. Nesting gives you the flexibility to store your control statements across multiple HFS files or PDS members for increased convenience and control.
DLIModel Utility Control Statements

For example, a top-level file could contain the OPTIONS, PSB and PCB statements that specify a desired Java-class generation. Included files might each contain a group of SEGEM and FIELD statements that relate to an individual logical or physical DBD. You can reuse these latter files without change for other PSBs that reference the same databases and segments.

Control Statement Rules

The control statement syntax is very flexible. Each statement consists of an identifier followed by keyword parameters. The identifier may start in any column. Each identifier, keyword, and variable must be separated by at least one white-space character, unless it is already separated by an operator. Keyword parameters can occur in any order.

If your control statements are held in an MVS data set, map the statements to multiple 80-character records, between columns 1 and 72 inclusive. Columns 73 through 80 are ignored, and may be used for sequence numbers if you wish. No continuation characters are required.

If your control statements are held in an HFS file, any line length is acceptable, but you can optionally continue statements across multiple lines as in MVS. If you restrict your line length to less than 73 characters, your control statements can be moved between MVS data sets and HFS files without change.

Identifiers, parameter keywords, and predefined parameter values (such as, YES and NO) may appear in upper or lower case. Other parameter values (for example, user specified path or Java names) are case sensitive.

Control Statement Syntax

OPTIONS Statement

One OPTIONS control statement is required. The statement customizes the DLIModel utility by specifying where to find input, what output to produce, and where to put the output.

The following diagram shows the format of the OPTIONS statement.
DLIModel Utility Control Statements

**PSBds=IMS.qual.dsname**
Required parameter specifies the data set name of the PSB source. If multiple parameters are specified, the utility opens and searches the data sets in the order of the PDSds parameters when it is reading a PSB. This action is similar to data set concatenation in an JCL DD statement.

**DBDds=IMS.qual.dsname**
Required parameter specifies the data set name of the DBD source. If multiple parameters are specified, the utility opens and searches the data sets in the order of the DBDds parameters when it is reading a DBD. This action is similar to data set concatenation in an JCL DD statement.

**Package=packagename**
Optional parameter specifies the package that the generated IMS Java classes are for. A Java package statement is added to each JAVA file produced.

**GenJavaSource=YES | NO**
Optional parameter specifies whether to generate IMS Java class source files and a DLIModel Java report.

**GenXMI=YES | NO**
Optional parameter specifies whether to generate an XMI file (dlimodelxmi.xmi) that describes the database model based on all PSBs and corresponding databases processed by the utility.

**GenTrace=YES | NO**
Optional parameter specifies whether to generate a trace file (named dlimodeltrace) of the utility run.

**OutPath=path**
Optional parameter specifies the HFS directory where the utility writes the output files, unless you specify path names for specific output files. The default is the current directory.

**JavaSourcePath=path**
Optional parameter specifies the HFS directory where the utility writes the IMS Java class files. Overrides OutPath.

**ReportPath=path**
Optional parameter specifies the HFS directory where the utility writes the DLIModel Java report. Overrides OutPath.

**XMLPath=path**
Optional parameter specifies the HFS directory where the utility writes the generated XML. Overrides OutPath.

**TracePath=path**
Optional parameter specifies the HFS directory where the utility writes the trace file. Overrides OutPath.

**FieldOrder=DEFAULT | OFFSET**
Optional parameter specifies the order of the fields of segments in the generated IMS Java class.

**DEFAULT**
Fields are in the same order as in the DBD and followed by any new fields defined by the control statements.
DLIModel Utility Control Statements

OFFSET
Fields are in the order their start positions.

PSB Statement
The PSB statement is required because it defines which PSBs that the utility uses. Multiple PSB statements are allowed, unless the * wildcard form is specified.

The following diagram shows the syntax of the PSB statement.

\[
\text{PSB} \rightarrow \text{PSBName} = \text{name} | \text{nameprefix}* | \star
\]

\text{PSBName= name | nameprefix | *}
Required parameter specifies the PSB to be used by the utility.

\text{name}
Process the PSB with the name \text{name}.

\text{nameprefix*}
Process all PSBs with the \text{nameprefix} in the specified PSB data set input file.

\text{*}
Process all PSBs in the specified data set input file.

\text{JavaName= name}
Optional parameter specifies the name of the generated IMS Java class only if using one PSB. Ignored if using \text{nameprefix} or \text{ALL} for \text{PSBName}.

PCB Statement
The PCB statement is optional (unless the PCB name is an SQL keyword, see Appendix B, “SQL Keywords,” on page 133). It specifies a Java alias for a PCB. All PCB statements for a PSB must follow the PSB statement.

The following diagram shows the syntax of the PCB statement.

\[
\text{PCB} \rightarrow \text{PCBName} = \text{name} | \star
\]

\text{PCBName= name}
Required parameter specifies the eight-character PCB name that you want to assign an alias to.

\text{JavaName= name}
Required parameter specifies the Java alias for the PCB, which will be used in the Java application. Must be unique for each PSB. Must not be an SQL keyword.

SEGM Statement
The SEGM control statement is optional (unless the segment name is an SQL keyword, see Appendix B, “SQL Keywords,” on page 133) and used for physical and logical segments.

For physical segments, the SEGM statement:
- Identifies a physical segment in a DBD
- Supplies a Java alias for the segment.
- Specifies an XMI COBOL copybook file containing additional information about the segment
Groups the FIELD statements that follow the SEGM statement

For logical segments, the SEGM statement:
- Identifies a logical segment in a logical DBD
- Specifies a Java alias for the segment.
- Cannot be followed by any FIELD statements

If the utility cannot find the segment, it issues a warning (instead of an error) and ignores any following FIELD statements. Because the utility only issues an error, you can create control statement files that provide information about many segments and their fields, not all of which are used by the particular PSB being processed.

If an XMI COBOL copybook file was named for a segment, the fields that it defines are merged by name with the fields defined in the DBD.

The following diagram shows the syntax of the SEGM statement.

```
SEGM
DBDName=name
SegmentName=name
JavaName=name
CobolXMI=name
```

**DBDName=**\textit{name}
- Required parameter specifies the eight-character DBD name where the segment is defined.

**SegmentName=**\textit{name}
- Required parameter specifies the segment name in the DBD.

**JavaName=**\textit{name}
- Required parameter specifies the Java alias for the segment, which will be used in the Java application. Must be unique for each DBD. If specified, overrides any value that might have been set from a COBOL XMI file.

**CobolXMI=**\textit{name}
- Optional parameter specifies name of an XMI COBOL copybook file that may provide additional information about the segment and its existing fields, and definitions of new fields. XMI input is only allowed for physical segments.

**FIELD Statement**
The FIELD statement is optional (unless the field name is an SQL keyword, see Appendix B, “SQL Keywords,” on page 133). It specifies information about a field or defines a new field for a segment in a physical DBD. All FIELD statements for a segment must immediately follow the SEGM statement. However, FIELD statements can be in any order and mixed with XDFLD statements.

When adding information for an existing DBD field, you must specify the 8-character DBD name of the field using the Name parameter. You can optionally specify the starting position (Start parameter) and length (Bytes parameter) of the field. If you do, DLIModel checks these values against the DBD and produces an error message if they do not match.

To add information to a non-DBD field that has been inputted from COBOL copybook XMI file, specify a Java name (JavaName parameter) that matches the name of the copybook field. Do not specify a DBD 8-character field name (Name parameter). Not specifying a Name parameter, for example, to add a default value to a COBOL copybook field.
To define a new field in the segment, do not specify a DBD 8-character field name. Instead, specify a unique Java name (JavaName parameter) that does not match any Java field name in the segment. You must also specify a starting position (Start parameter) and a length (Bytes parameter) for the new field. You can include other attributes (for example, data type or default value) for the new field.

To define a new field, you must specify the starting position (Start parameter), the length (Bytes parameter) of the field, and the name (Name or JavaName parameter) of the field.

The following diagram shows the syntax of the FIELD statement.

```
FIELD
  Name=name
  Start=int
  Bytes=name
  JavaName=name
```

- **Name=name**
  Specifies eight-character field name as defined in the DBD. Must be unique within segment. Identifies this control statement as applying to an existing field within the DBD. Do not specify this parameter if you are specifying a new field.

- **Start=int**
  Specifies the starting position of the field in the segment. The first byte in the segment is 1. Required for new fields and optional for existing fields.

- **Bytes=name**
  Specifies the length of the field in the segment. Required for new fields and optional for existing fields.

- **JavaName=name**
  Specifies the Java alias for the field. Must not be an SQL keyword. Java names of Field statements and XDFLD statements must be unique within a segment. Required if defining a new field and optional for existing fields. If a field has been inputted from COBOL copybook XMI file with the same name as the JavaName parameter on a control statement, the control statement is applied to this COBOL copybook field.

- **JavaType=string**
  Optional parameter specifies the Java type of the field. Default is CHAR.
  Allowed types:
  - CHAR
  - FLOAT
  - DOUBLE
  - SMALLINT
  - INTEGER
  - BIGINT
  - ZONEDDECIMAL
  - TIME
  - CARCHAR
  - TINYINT
  - BIT
  - TYPELIST
  - BINARY
  - PACKEDDECIMAL
  - DATE
  - TIMESTAMP
Overrides any value that was set by the XMI COBOL copybook file.

**TypeQualifier=string**

Required parameter for some Java types. Specifies type qualifier for the following Java types:
- PACKEDDECIMAL
- ZONEDDECIMAL
- DATE
- TIME
- TIMESTAMP

**Related Reading:** For more information on determining the syntax of type qualifiers, see "Supported JDBC Data Types" on page 86.

**Default=string**

Optional parameter specifies the default value for the field. The default value is used for new instances of the segment when an application does not define a value for the field. The string must be formatted to match the data type qualifier properties of the field.

**XDFLD Statement**

The XDFLD statement is optional. It specifies a Java alias for an existing secondary index field in a segment. The XDFLD statements must follow the SEGM statement that corresponds to the segment with the secondary indexes in the DBD.

You must identify a secondary index field (which must be an existing field that was defined in the DBD) by the eight-character name because secondary index fields do not have a starting position in the segment. Secondary index fields do not have a data type. Therefore, you must create a single string that contains the concatenated search fields, each correctly encoded for its data type, when using the index field in a JDBC Select. An index field cannot be fetched from the JDBC result set.

The following diagram shows the syntax of the XDFLD statement.

```
XDFLD—Name=name—JavaName=name
```

**Name=name**

Required parameter specifies the eight-character name of the secondary index field as defined in the DBD.

**JavaName=name**

Required parameter specifies the Java alias for the field.

**INCLUDE Statement**

The INCLUDE statement is optional and is only allowed in the top-level control statement data set. It specifies a PDS member or HFS file of additional control statements to be included in the top-level data set. The included data set must be the same type (PDS or HFS) as the top-level data set. You are allowed any number of INCLUDE statements in the top-level data set.

**Important:** Do not put an INCLUDE statement between PSB statements and PCB statements or between SEGM statements and FIELD statements. Putting the INCLUDE statement between these statements breaks the required relationship between them.

The following diagram shows the syntax for the INCLUDE statement.
**DLIModel Utility Control Statements**

```
| INCLUDE Dataset=IMS.qual.dsname(member) path/filename |
```

Required parameter specifies the PDS member of HFS file containing the control statements that are to be included in the top-level data set.

**Comment Statement**

The Comment Statement is optional. It indicates that a line in the PDS member is a comment the same way as you would do in Java code. For example:

```java
// The two slashes indicate that this line is a comment.
```

---

**Running the DLIModel Utility**

You can run the DLIModel utility either as a standard OS/390 job, or from the command prompt under UNIX System Services. For the latter alternative, see “Running the DLIModel Utility From UNIX Systems Services” on page 113.

**Running the DLIModel Utility as a JCL Job**

The DLIMODEL procedure is delivered as member DFSMODEL in the IMS distribution library SDFSISRC. To prepare this procedure, perform the following steps:

1. Copy the member DFSMODEL from its distribution library SDFSISRC to the PROCLIB from where you submit IMS procedures for batch execution.
2. Rename the procedure, if desired. These instructions assume that you have renamed it DLIMODEL.

Because the DLIModel utility is a Java application, the DLIMODEL procedure runs it on OS/390 using BPXBATCH, an OS/390-provided utility that runs any OS/390 UNIX shell command or executable.

The DLIMODEL procedure has two steps:

- Step 1 executes the DLIModel utility (a Java application) by invoking the OS/390 utility BPXBATCH. The data set name of a PDS control data set is provided to the utility through the EXEC statement PARM field. This step contains DD statements for the utility’s standard output streams, STDOUT and STDERR, which are directed to temporary HFS files. Other utility inputs and outputs are read from, or sent to data sets and files with names specified by the control data set, and do not have DD statements.

- Step 2 redirects STDOUT and STDERR to SYSOUT streams where they can be viewed using your usual procedures, for example, using SDSF.

The following is the DLIMODEL procedure, which runs the DLIModel utility:
PROC Statement Parameters

DSNAME
The name of the control data set for the utility.

SOUT Sets the class for all SYSOUT output in the procedure.

Step 1 EXEC Statement Parameters

PGM=BPXBATCH
Runs the BPXBATCH utility.

PARM Runs the utility as a Java application under the UNIX shell.

ABSPATH is the string parameter containing the fully qualified path of the HFS script file named go. This script file contains the java command that executes the DLIModel class file.

Note: The go file is a script file that contains the java command and specifies the required JAR files needed by the application. The command uses the $ symbol. Only edit this file if the $ symbol is not valid in your locale, in which case replace the $ symbol with an appropriate symbol.

DSNAME is a string parameter containing the fully qualified data set name of the top-level control data set, which contains the DLIModel utility control statements (as described in "Control Statements for the DLIModel Utility" on page 102). DSNAME must refer to a PDS or PDSE member. The format is:

qual1.qual2.dsnamedname(member)

The named data set must be F or FB type with an LRECL=80.

PDS specifies that the control statements are in a PDS. The control statements must be in a PDS when you are running the utility as a JCL job.
Running the DLIModel Utility

Step 1 DD Statements

STDENV DD
Contains statements that set the Java environment variables. You should not need to use this DD statement.

STDOUT DD
The destination to which the utility in STEP 1 directs standard output. This includes messages recording the normal execution of the utility. This output is redirected by STEP 2 to the standard MVS SYSOUT destination.

STDERR DD
The destination to which the utility in STEP 1 directs standard error output. This includes error and warning messages related to the execution of the utility. This output is redirected by STEP 2 to the standard MVS SYSOUT destination.

SYSTSIN DD
Provides control cards for the MVS utility IKJEFT01 to copy the temporary HFS output files to the MVS SYSOUT destination.

Step 2 EXEC Statement Parameters

PGM=IKJEFT01
Runs the MVS utility IKJEFT01, which redirects STDOUT and STDERR to MVS SYSOUT.

DYNAMNBR

COND

Step 2 DD Statements

SYSTEPRT DD
IKJEFT01 utility output.

HFSOUT DD
Input from the temporary STDOUT file from step 1.

HFSERR DD
Input from the temporary STDERR file from step 1.

STDOUTL DD
Output destination for the STDOUT stream.

STDERRL DD
Output destination for the STDERR stream.

SYSPRINT DD
IKJEFT01 utility output.

SYSTSIN DD
Must be added to execution JCL. Provides control statement input for the IKJEFT01 utility that redirect HFSOUT and HFSERR streams to the STDOUT and HFSERRL destinations. For example:

OCOPY INDD(HFSOUT) OUTDD(STDOUTL)
OCOPY INDD(HFSERR) OUTDD(STDERRL)
Running the DLIModel Utility

Related Reading: For more information about the OS/390 BPXBATCH utility, see z/OS: UNIX System Services User’s Guide and z/OS: UNIX System Services Command Reference.

The following JCL is an example of a job that runs the DLIMODEL procedure:

```bash
//BPXAUTP6 JOB CLASS=Z,MSGCLASS=E,MSGLEVEL=(1,1),
// TIME=(9),USER=OMVSADM,PASSWORD=xxxxxxx,
// REGION=32M
//TEST EXEC DLIMODEL,DSNAME='qual1.qual2.dsname(CNTRSTMT)',
// ABSPATH='/usr/lpp/ims/imsjava81/dlimodel'
//STEP2.SYSTSIN DD *
OCOPY INDD(HFSOUT) OUTDD(STDOUTL)
OCOPY INDD(HFSERR) OUTDD(STDERRL)
/*
```

In this example, the IKJEFT01 SYSTSIN DD statement is provided with control statements to copy the temporary HFS outputs to SYSOUT destinations.

Running the DLIModel Utility From UNIX Systems Services

In addition to the JCL procedure, you can run the DLIModel utility from a prompt under UNIX System Services. You can use this method if you are more familiar with a UNIX environment than with JCL.

Before running the DLIModel utility from the UNIX System Services prompt, ensure that your PATH environment variable is set so that the SDK 1.3.1 java command is accessible.

To run the DLIModel utility, enter the go command using the following syntax from the /imsjava81/dlimodel directory:

```bash
/go HFSpath/controldataset
qual.qual.dsname(member) PDS
```

**go** Script file that contains the java command to execute the DLIModel class.

**Note:** The go file is a script file that contains the java command and specifies the required JAR files needed by the application. The command uses the $ symbol. Edit this file only if the $ symbol is not valid in your locale, in which case replace the $ symbol with an appropriate symbol.

*path/controldataset*

HFS file of the control data set.

*qual.qual.dsname(member)*

PDS file for control data set.

**PDS**

Specifies that the control data set is a PDS and not an HFS file.

Examples of Using the DLIModel Utility

This section shows examples of how the DLIModel utility uses DBDs, PSBs, and control statements to create IMS Java classes and DLIModel Java reports.

The examples in this section are in the following default directories:
JMP IVP Metadata Sample

You can generate the IMS Java metadata class needed to run the JMP IVP using the input files found in /usr/lpp/ims/imsjava81/dlimodel/samples/ivpJMP/. After you run the utility, you can compare the generated file with the IMS Java-provided file in /usr/lpp/ims/imsjava81/samples/ivp/.

The IMS Java metadata source file DFSIVP37DatabaseView.java, can also be used, after it is compiled, for the WebSphere Application Server IVP, DB2 stored procedures IVP, and the CICS IVP.

**Figure 36** shows the DBD for the IVP database. It is referenced by the PSB, which has a single PCB and one sensitive segment, shown in **Figure 37**.

```
DBD  NAME=IVPDB2,ACCESS=HDAM,RMNAME=(DFSIVD40,40,100)
DATASET DD1=DFSIVD2,DEVICE=3380,SIZE=2048
SEGM  NAME=A1111111,PARENT=0,BYTES=40,RULES=(LLL,LAST)
FIELD  NAME=(A1111111,SEQ,U),BYTES=010,START=00001,TY=CHAR
```

**Figure 36. DBD for the IVP Database**

```
PHONEAP  PCB TYPE=DB,DBDNAME=IVPDB2,PROC=AP,KEYLEN=10
SENSEG  NAME=A1111111,PARENT=0,PROC=AP
PSBGEN  LANG=JAVA,PSBNAME=DFSIVP37
```

**Figure 37. PSB for the JMP IVP**

Note that the PCB is given a name, as required by IMS Java. In this case a label is used, but a PCBNAME= parameter is also acceptable.

The control data set, named contrstmt, is shown in **Figure 38**.

```
OPTIONS PSBds=IMS.TEST1.PSBSRC
DBDds=IMS.TEST1.DBDSRC
GenJavaSource=YES OutPath=samples/ivpJMP
Package=samples.ivp
GenTrace=yes
PSB PSBName=DFSIVP37 JavaName=DFSIVP37DatabaseView
PCB PCBName=PHONEAP JavaName=PhoneBook
SEGM DBDName=IVPDB2 SegmentName=A1111111 JavaName=Person
FIELD Name=A1111111 Start=1 Bytes=10 JavaName=LastName JavaType=CHAR
FIELD Start=11 Bytes=10 JavaName=FirstName JavaType=CHAR
FIELD Start=21 Bytes=10 JavaName=Extension JavaType=CHAR
FIELD Start=31 Bytes=7 JavaName=ZipCode JavaType=CHAR
```

**Figure 38. Control Data Set for JMP IVP**

The utility processes a single PSB named DFSIVP37 and its DBD, IVPDB2. The utility reads the PSB and DBD from data sets IMS.TEST1.PSBSRC and IMS.TEST1.DBDSRC. The name of the generated class is DFSIVP37DatabaseView (the first eight characters of this name are set from the PSB name) and the class is
written to the HFS file, DFSIVP37DatabaseView.java under the current directory.
STDOUT is redirected to SYSOUT, but (in the absence of errors) consists of only
startup and normal completion messages. The DLIModel Java Report (produced
whenever IMS Java classes are generated) is written to the HFS file,
DFSIVP37JavaReport.txt.

The DLIModel Java Report, shown in Figure 39, describes the generated IMS Java
metadata class DFSIVP37DatabaseView.

DLIModel IMS Java Report
===================================
Class: DFSIVP37DatabaseView in package: samples.ivp generated for PSB: DFSIVP37

===================================
PCB: PhoneBook

Segment: Person
Field: LastName Type=CHAR Length=10 ++ Primary Key Field ++
Field: FirstName Type=CHAR Length=10
Field: Extension Type=CHAR Length=10
Field: ZipCode Type=CHAR Length=7

Figure 39. DLIModel IMS Java Report for JMP IVP

Notice these things in the DLIModel IMS Java Report:

- The class name, DFSIVP37DatabaseView, is based on the IMS PSB name.
- The PCB name, DFSIVP37, is the same as the label in the PSB PCB statement.

Each field line displays the starting position and length of the field in the segment,
and its type. Again, since there are no control statements or XMI COBOL copybook
files to specify otherwise, the type of all fields defaults to CHAR.

JBP IVP Metadata Sample

You can generate the IMS Java metadata class needed to run the JBP IVP using
the input files found in /usr/lpp/ims/imsjava81/dlimodel/samples/ivpJBP/. After
you run the utility, you can compare the generated file with the IMS Java-provided
file in /usr/lpp/ims/imsjava81/samples/ivp/.

The IMS Java metadata source file DFSIVP67DatabaseView.java, can only be
used, after it is compiled, for the JBP IVP.

Figure 36 on page 114 shows the DBD for the IVP database. It is referenced by the
PSB, which has a single PCB and one sensitive segment, shown in Figure 37 on

Figure 36 on page 114 shows the DBD for the IVP database. It is referenced by the
PSB, which has a single PCB and one sensitive segment, shown in Figure 37 on

DBD   NAME=IVPDB2,ACCESS=HDAM,RMNAME=(DFSHDC40,40,100)
DATASET DD1=DFSIVD2,DEVICE=3380,SIZE=2048
SEGM   NAME=A1111111,PARENT=0,BYTES=40,RULES=(LLL,LAST)
FIELD   NAME=(A1111111,SEQ,U),BYTES=010,START=00001,TYPE=C
DBGEN
END

Figure 40. DBD for the IVP Database
Note that the PCB is given a name, as required by IMS Java. In this case a label is used, but a PCBNAME= parameter is also acceptable.

The control data set, named contrstmt, is shown in Figure 38 on page 114.

The utility processes a single PSB named DFSIVP67 and its DBD, IVPDB2. The utility reads the PSB and DBD from data sets IMS.TEST1.PSBSRC and IMS.TEST1.DBDSRC. The name of the generated class is DFSIVP67DatabaseView (the first eight characters of this name are set from the PSB name) and the class is written to the HFS file, DFSIVP67DatabaseView.java under the current directory.

STDOUT is redirected to SYSOUT, but (in the absence of errors) consists of only startup and normal completion messages. The DLIModel IMS Java Report (produced whenever IMS Java classes are generated) is written to the HFS file, DFSIVP67JavaReport.txt.

The DLIModel IMS Java Report, shown in Figure 39 on page 115, describes the generated IMS Java metadata class DFSIVP67DatabaseView.

The DLIModel IMS Java Report shows the generated class DFSIVP67DatabaseView for PSB DFSIVP67.

```
OPTIONS PSBds=IMS.TEST1.PSBSRC
       DBDds=IMS.TEST1.DBDSRC
       GenJavaSource=yes Outpath=samples/ivpJBP
       Package=samples.ivp GenTrace=yes
PSB PSBName=DFSIVP67 JavaName=DFSIVP67DatabaseView
PCB PCBName=PHONEAP JavaName=PhoneBook
SEGm DBDName=IVPDB2 SegmentName=A1111111 JavaName=Person
FIELD Name=A1111111 Start=1 Bytes=10 JavaName=LastName JavaType=CHAR
FIELD Start=11 Bytes=10 JavaName=FirstName JavaType=CHAR
FIELD Start=21 Bytes=10 JavaName=Extension JavaType=CHAR
FIELD Start=31 Bytes=7 JavaName=ZipCode JavaType=CHAR
```

Figure 42. Control Data Set for JBP IVP

```
PHONEAP PCB TYPE=DB, DBDNAME=IVPDB2, PROCOPT=A, KEYLEN=10
SENSEG NAME=A1111111, PARENT=0, PROCOPT=AP
PSBGEN LANG=JAVA, PSBNAME=DFSIVP67
END
```

Figure 41. PSB for the JBP IVP

Note that the PCB is given a name, as required by IMS Java. In this case a label is used, but a PCBNAME= parameter is also acceptable.

The control data set, named contrstmt, is shown in Figure 38 on page 114.

```
OPTIONS PSBds=IMS.TEST1.PSBSRC
       DBDds=IMS.TEST1.DBDSRC
       GenJavaSource=yes Outpath=samples/ivpJBP
       Package=samples.ivp GenTrace=yes
PSB PSBName=DFSIVP67 JavaName=DFSIVP67DatabaseView
PCB PCBName=PHONEAP JavaName=PhoneBook
SEGm DBDName=IVPDB2 SegmentName=A1111111 JavaName=Person
FIELD Name=A1111111 Start=1 Bytes=10 JavaName=LastName JavaType=CHAR
FIELD Start=11 Bytes=10 JavaName=FirstName JavaType=CHAR
FIELD Start=21 Bytes=10 JavaName=Extension JavaType=CHAR
FIELD Start=31 Bytes=7 JavaName=ZipCode JavaType=CHAR
```

Sample Metadata with COBOL Copybook XMI

This example uses the physical database and PSB shown in Figure 45 on page 117 and Figure 44 on page 117 to illustrate the use of control statements and a COBOL
XMI file to define additional fields and additional name and data type information. It is the same example that is used in Chapter 6, “JDBC Access to IMS Data,” on page 73 to describe how to access IMS databases from Java applications using JDBC. This example also illustrates how control statements might be split across more than one file.

```
DLRPCB1 PCB TYPE=DB, DBDNAME=DEALERDB, PROCOPT=GO, KEYLEN=42
  SENSEG NAME=DEALER, PARENT=0
  SENSEG NAME=MODEL, PARENT=DEALER
  SENSEG NAME=ORDER, PARENT=MODEL
  SENSEG NAME=SAL, PARENT=MODEL
  SENSEG NAME=STOCK, PARENT=MODEL
PSBGEN PSBNAME=AUTPSB4, MAXQ=200
END
```

**Figure 44. PSB for COBOL Copybook XMI Example**

```
DBD NAME=DEALERDB, ACCESS=(HDAM, OSAM), RNNAME=(DFSHDC40.1.10)
  SEGM NAME=DEALER, PARENT=0, BYTES=94
    FIELD NAME=(DLRNO, SEQ, U), BYTES=4, START=1, TYPE=C
    FIELD NAME=DLRNAME, BYTES=30, START=5, TYPE=C
  SEGM NAME=MODEL, PARENT=DEALER, BYTES=43
    FIELD NAME=(MODTYPE, SEQ, U), BYTES=2, START=1, TYPE=C
    FIELD NAME=MAKE, BYTES=10, START=3, TYPE=C
    FIELD NAME=YEAR, BYTES=4, START=23, TYPE=C
    FIELD NAME=MSRP, BYTES=5, START=27, TYPE=P
  SEGM NAME=ORDER, PARENT=MODEL, BYTES=127
    FIELD NAME=(ORDNBR, SEQ, U), BYTES=6, START=1, TYPE=C
    FIELD NAME=LASTNME, BYTES=25, START=50, TYPE=C
    FIELD NAME=FIRSTNME, BYTES=25, START=75, TYPE=C
  SEGM NAME=SAL, PARENT=MODEL, BYTES=113
    FIELD NAME=(SALDATE, SEQ, U), BYTES=8, START=1, TYPE=C
    FIELD NAME=LASTNME, BYTES=25, START=9, TYPE=C
    FIELD NAME=FIRSTNME, BYTES=25, START=34, TYPE=C
    FIELD NAME=STKVIN, BYTES=20, START=94, TYPE=C
  SEGM NAME=STOCK, PARENT=MODEL, BYTES=62
    FIELD NAME=(STKVIN, SEQ, U), BYTES=20, START=1, TYPE=C
    FIELD NAME=COLOR, BYTES=10, START=37, TYPE=C
    FIELD NAME=PRICE, BYTES=5, START=47, TYPE=C
    FIELD NAME=LOT, BYTES=10, START=53, TYPE=C
DBDGEN
FINISH
END
```

**Figure 45. Physical DBD for COBOL Copybook XMI Example**

The example is run from the UNIX System Services prompt, as shown in **Figure 46**. This example assumes that you are running it from the `usr/lpp/ims/imsjava81/dlimodel` directory.

```
> java com.ibm.ims.metagen.DLIModel samples/cobolXMI/cntrstmt
```

**Figure 46. UNIX System Services Command for Control Statements Example**

The top-level control statement file, `cntrstmt`, is shown in **Figure 47 on page 118**.
This control file establishes the options for the execution, as in previous examples. It names the PSB to be processed, assigns a Java name for the generated class for this PSB, and provides a Java name for a PCB in that PSB.

Unlike previous examples, this example includes a second-level control statement file, called cntrstm2, which is shown in Figure 48 on page 119 and a COBOL copybook XMI file. This control file and the COBOL copybook XMI file provide details of additional fields in the segments of the database referenced from DLR_PCB1 and additional information about existing fields in that database. Note these facts about this second control statement file:

- The information it contains is only necessary because there are additional facts about the segments in this database that are needed by this (hypothetical) Java application. If your DBD names all the fields that are used by your application, and if all the fields can be treated as CHAR data type, and if your application can use the standard 8-character names, then you would not need to supply SEGM or Field control statements.

- The Segment and Field control statements need only be split off into a second file if it is convenient to do so, perhaps because this additional segment information needs to be shared by other applications. In such cases you might group all field information for a whole database (as is shown in Figure 48 on page 119) or for each segment into its own file. If this is not advantageous for your data, it is equally acceptable to place all control statements in a single, top-level file.

OPTIONS

```plaintext
PSBds=IMS.TEST1.PSBSRC
DBDds=IMS.TEST1.DBDSRC
GenJavaSource=YES
GenTrace=yes
Package=com.ibm.ims.tooling
```

PSB PSBName=autpsb4 JavaName=DealerDatabaseView
PCB PCBName=DLRPCB1 JavaName=DealershipDB
INCLUDE Dataset=samples/example4/cntrstm2

Figure 47. Top-Level Control Data Set for COBOL Copybook XMI Example

This names the PSB to be processed, assigns a Java name for the generated class for this PSB, and provides a Java name for a PCB in that PSB.
Under the SEGM statement for DEALER, the first Field statement identifies an existing field, DLRNO, by both its DBD name and its start position and length. These facts will be checked for consistency against the DBD. If the field is identified correctly, then it is assigned a Java name, DealerNumber, and a data type, CHAR (which is the default, and therefore not necessary to specify).

The second Field statement identifies an existing field by only its start position and length. If such a field exists, it is assigned the Java name, DealerName. This abbreviated method of identifying the field achieves the same result, but is not quite so safe because no 8-character name check is carried out. The data type for DealerName default is CHAR.

The third Field statement under the DEALER SEGM statement defines a new field—a field that is physically present in the segment, but is not described by a FIELD macro in the DBD. It specifies the start position and length of this field, assigns it a Java name of DealerAddress, and a type of CHAR.

The fourth Field statement defines another new field, YTDSales, of type PACKEDDECIMAL. This data type requires a type qualifier that defines the field format, and in this example, a qualifier of S9(18) is supplied.
The remainder of the control statements describe information for the other segments and fields in the DBD in a similar manner, except for the STOCK segment.

The fields in the STOCK segment are described in the COBOL copybook XMI file. The COBOL copybook XMI file is generated from a COBOL copybook file. Figure 49 shows the COBOL copybook that describe the STOCK fields. Before this copybook can be used as input into the DLIModel utility, it must be converted into XMI.

```cobol
01 AutoStock.
   05 StockVINumber pic x(20).
   05 DateInfo.
      10 DateIn pic x(8).
      10 DateOut pic x(8).
   05 Color pic x(10).
   05 Price pic 9(6).
   05 Lot pic x(10).
```

**Figure 49. Copybook for COBOL Copybook XMI Example**

When DLIModel is executed, it generates the DLIModel Java Report in Figure 51 on page 121 together with a matching metadata class (not shown).

The copybook shown in Figure 49 when its XMI is used as input to the utility, is equivalent to the control statements shown in Figure 50.

```cobol
SEGM DBDName=DEALERDB SegmentName=STOCK JavaName=STOCKXX
FIELD Start=1 Bytes=20 JavaType=CHAR JavaName=StockVINumber
FIELD Start=21 Bytes=8 JavaType=CHAR JavaName=DateIn
FIELD Start=29 Bytes=8 JavaType=CHAR JavaName=DateOut
FIELD Start=37 Bytes=10 JavaType=CHAR JavaName=Color
FIELD Start=47 Bytes=6 JavaType=ZONEDDECIMAL TypeQualifier=9(6) JavaName=Price
FIELD Start=53 Bytes=10 JavaType=CHAR JavaName=Lot
```

**Figure 50. Equivalent Control Statements for COBOL Copybook XMI Example**
In this DLIModel Java Report, the names of segments and fields are the Java names supplied in the control statements and the COBOL copybook XMI. The eight-character IMS names do not appear because the Java developer does not need to know these names.
DLIModel Utility Examples
Chapter 8. Problem Determination

This chapter describes how to debug your Java applications that use IMS Java and determine the source of problems within your applications.

The following topics provide additional information:
- "Exceptions"
- "XML Tracing for IMS Java" on page 124
- "Debugging an Unresettable JVM in a JMP or JBP Region" on page 127

Exceptions

Exceptions are thrown as a result of non-blank status codes and non-zero return codes (in cases when there were no PCBs to deliver status codes) from IMS DL/I calls. Even though an exception is thrown by the JavaToDLI class for every non-blank status code, some of these exceptions are caught by the application or database packages and converted to return values.

How Exceptions Map to DL/I Status Codes

The com.ibm.ims.base.IMSException class extends the java.lang.Exception class. The DLIEException class extends the IMSException class. The DLIEException class includes all errors that occur within the IMS Java library that are not a result of any call to IMS.

You can use the following methods to get information from an IMSException object:

getAIB
Returns the IMS application interface block (AIB) from the DL/I call that caused the exception. The IMS AIB is null for the DLIEException object. The methods on the AIB can be called to return other information at the time of the failure, including the resource or PCB name and the PCB itself.

getStatusCode
Returns the IMS status code from the DL/I call that caused the exception. This method works with the JavaToDLI set of constants. The status code is zero (0) for a DLIEException object.

getFunction
Returns the IMS function from the DL/I call that caused the exception. The function is zero (0) for a DLIEException object.

The following database access methods of the DLICconnection class return false if they receive a GB status code (no more such segments or segment not found) or a GE status code (no such segment or end of database):
- DLICconnection.getUniqueSegment
- DLICconnection.getNextSegment
- DLICconnection.getUniqueRecord
- DLICconnection.getNextRecord
- DLICconnection.getNextSegmentInParent
Exceptions

The IMSMessageQueue.getUniqueMessage method returns false if it receives a QC (no more messages) status code. The IMSMessageQueue.getNextMessage method returns false if it receives a QD status code, which means that there are no more segments for multi-segment messages.

The example in Figure 52 extracts information from an IMSException object:

```java
try {
    DealerDatabaseView dealerView = new DealerDatabaseView();
    DLIConnection connection = DLIConnection.createInstance(dealerView);
    connection.getUniqueSegment(dealerSegment, dealerSSAList);
} catch (IMSException e) {
    short statusCode = e.getStatusCode();
    String failingFunction = e.getFunction();
}
```

Figure 52. IMSException Class Example

Related Reading: For more information about DL/I status codes, see IMS Version 8: Application Programming: Database Manager and IMS Version 8: Application Programming: Transaction Manager

SQLException Objects

An SQLException object is thrown to indicate that an error has occurred either in the Java address space or during database processing.

Each SQLException provides the following information:
- A string that describes the error.
  - This string is available through the use of the getMessage() method.
- An “SQLstate” string that follows XOPEN SQLstate conventions.
  - The values of the SQLstate string are described in the XOPEN SQL specification.
- A link to the next SQL exception if more than one was generated.
  - The next exception is used as a source of additional error information.

XML Tracing for IMS Java

Using the com.ibm.ims.base.XMLTrace class, you can debug your Java applications by tracing, or documenting, the flow of control throughout your application. By setting up trace points throughout your application for output, you can isolate problem areas and, therefore, know where to make adjustments to produce the results you expect. In addition, because the XMLTrace class supports writing input parameters and results, and the IMS Java library methods use this feature, you can verify that correct results occur across method boundaries.

The XMLTrace class is different from the DLIModel utility trace. For information about how to enable tracing for the DLIModel utility, see OPTIONS Statement on page 104.

Note: The XMLTrace class replaces the IMSTrace class. However, applications that use the IMSTrace class will still function properly.
WebSphere Application Server Security Requirements for XML Tracing

Before you can trace your application that runs on WebSphere Application Server V5.0 for z/OS, you must add permissions to the WebSphere Application Server for z/OS server.policy file and create a was.policy for the application EAR file.

To add permissions to the WebSphere Application Server for z/OS server.policy file:

1. Open the WebSphere Application Server for z/OS server.policy file, which is in the properties directory of the WebSphere Application Server installation directory, and find the following code, which was added when you installed the custom service (if this code is not in the file, add it):

   ```
   grant codeBase "file:/imsjava/-" {
      permission java.util.PropertyPermission "*", "read, write";
      permission java.lang.RuntimePermission "loadLibrary.JavTDLI";
      permission java.io.FilePermission "/tmp/*", "read, write";
   };
   ```

2. Below permission java.io.FilePermission "/tmp/*", "read, write"; add the following permission, replacing traceOutputDir with the directory name for the trace output file:

   ```
   permission java.io.FilePermission "/traceOutputDir/*", "read, write";
   ```

To create the was.policy file:

1. Create a new file named was.policy that contains the following code, replacing traceOutputDir with the directory name for the trace output file:

   ```
   grant codeBase "file:${application}" {
      permission java.io.FilePermission "/traceOutputDir/*", "read, write";
   };
   ```

2. Put the was.policy file in the META-INF directory of your application’s EAR file.

Enabling XML Tracing

To debug with XMLTrace, you must first turn on the tracing function by calling one of the XMLTrace.enable methods. Because tracing does not occur until this variable is set, it is best to do so within a static block of your main application class. Then, you must decide how closely you want to trace the IMS Java library’s flow of control and how much tracing you want to add to your application code.

You can determine the amount of tracing in the IMS Java library by providing the trace level in the XMLTrace.enable method. By default, this value is set to XMLTrace.TRACE_EXCEPTIONS, which traces the construction of IMS Java-provided exceptions. XMLTrace also defines constants for three types of additional tracing. These constants provide successively more tracing from IMSTrace.TRACE_CTOR1 (level-one tracing of constructions) to IMSTrace.TRACE_DATA3 (level-three tracing of data).

XMLTrace has the following trace levels:

<table>
<thead>
<tr>
<th>Trace level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE_EXCEPTIONS</td>
<td>Traces exceptions</td>
</tr>
<tr>
<td>TRACE_CTOR1</td>
<td>Traces level-1 constructors</td>
</tr>
<tr>
<td>TRACE_METHOD1</td>
<td>Traces level-1 parameters, return values, methods, and constructors</td>
</tr>
</tbody>
</table>
Tracing the IMS Java Library Methods

To enable the tracing that is shipped with IMS Java library methods:

1. Call the XMLTrace.enable method and specify the root element name and the trace level. For example:
   ```java
   XMLTrace.enable("MyTrace", XMLTrace.TRACE_METHOD1);
   ```

2. Set an output stream (a print stream or a character output writer) as the current trace stream. For example:
   a. Set the system error stream as the current trace stream:
      ```java
      XMLTrace.setOutputStream(System.err);
      ```
   b. Set a StringWriter object (or any other type of writer) as the current trace stream:
      ```java
      StringWriter stringWriter = new StringWriter();
      XMLTrace.setOutputWriter(stringWriter);
      ```

3. Close the XML trace:
   ```java
   XMLTrace.close();
   ```

Steps 1 and 2 are best implemented within a static block of your main application class, as shown in Figure 53.

```java
public static void main(String args[]){
    static {
        XMLTrace.enable("MyTrace", XMLTrace.TRACE_METHOD1);
        XMLTrace.setOutputStream(System.err);
    }
}
```

Figure 53. Setting a Trace within a Static Method

Tracing Your Application

You can add trace statements to your application, similar to those provided by the IMS Java library, by defining an integer variable that you test prior to writing trace statements. Using a variable other than XMLTrace.libTraceLevel enables you to control the level of tracing in your application independently of the tracing in the IMS Java library. For example, you can turn off the tracing of IMS Java Library routines by setting XMLTrace.libTraceLevel to zero, but still trace your application code.
To enable tracing for your application:

1. Define an integer variable to contain the trace level for application-provided code:
   ```java
   public int applicationTraceLevel = XMLTrace.TRACE_CTOR3;
   ```
2. Set up the XMLTrace method to trace methods, parameters, and return values as necessary.

### Debugging an Unresettable JVM in a JMP or JBP Region

If you need to debug reset trace events for the persistent reusable JVM in a JBP or JMP region, you need to run the JVM in debug mode. The following messages in your job log indicate that you should run in debug mode to determine the problem:

DFSJVMO0: ResetJavaVM() RC=-1
DFSJVMO0: DestroyJavaVM() RC=-1

To run the JVM in debug mode, add DEBUG=Y to the DFSJVMEV sample member, or the member that is specified by the DFSJMP or DFSJBP ENVIRON= parameter.

**Related Reading:**

- For more information about running the JVM in debug mode, see *IBM Developer Kit for OS/390, Java 2 Technology Edition: New IBM Technology featuring Persistent Reusable Java Virtual Machines*.
- For more information about the DFSJVMEV member and the ENVIRON= parameter, see *IMS Version 8: Installation Volume 2: System Definition and Tailoring*.
XML Tracing
Appendix A. Preparing to Run the Dealership Samples

To run the dealership sample, you must prepare IMS by modifying the stage 1 input statements and loading the databases.

An IMS Java metadata class, which is a Java class that describes the IMS databases, is required for all applications. The IMS Java metadata class for the dealership sample applications, AUTPSB11DatabaseView, is provided compiled and is included with the application files. You do not have to do anything further to prepare this file.

The following topics provide additional information:

- "Modifying IMS Stage 1 Input Statements"
- "Loading the Dealership Sample Databases"

Modifying IMS Stage 1 Input Statements

Before you can access the sample dealership databases with the sample applications, you must modify the IMS system definition stage 1 input statements.

To modify the stage 1 input statements:

1. Add the following DATABASE macro statements to the IMS stage 1 input statements:

   DATABASE DBD=AUTODB,ACCESS=UP
   DATABASE DBD=EMPDB2,ACCESS=UP
   DATABASE DBD=SINDEX11,ACCESS=UP
   DATABASE DBD=SINDEX22,ACCESS=UP

2. Add a APPLCTN macro statement to the IMS stage 1 input statements for the sample application's program resource requirements. The sample applications use AUTPSB11 as the PSB. All sample applications require an APPLCTN statement for the AUTPSB11 PSB. For example:

   APPLCTN PSB=AUTPSB11,PGMTYPE=TP,SCHEDTYP=PARALLEL

3. If you are running the JMP version of the dealership sample application, add the TRANSACT macro statement following the APPLCTN macro statement. The JMP dealership sample application is nonconversational. For example:

   TRANSACT CODE=AUTRAN11,PRTY=(7,10,2),INQUIRY=NO,MODE=SNGL, X
   MSGTYPE=(SNGLSEG,NONRESPONSE,1)

Loading the Dealership Sample Databases

To run the sample dealership applications, you must first load the databases that the applications access. The files that are needed to load these databases are in the samples directory:

`pathprefix/usr/ims/imsjava81/samples/dealership/databases`. To use these files, however, you must move them from HFS files to PDS members. The following steps provide sample jobs that move the files. If you use these sample jobs, replace `path` with `pathprefix/usr/ims/imsjava81/samples/dealership/databases`.

To load the dealership sample databases:

1. Move the following DBD source files (HFS) to your DBD source library (PDS members):
   - AUTODDB (physical DBD of the auto database)
   - EMPDB2 (physical DBD of the employee database)
Loading the Dealership Sample Databases

- SINDEX11 (first secondary index)
- SINDEX22 (second secondary index)
- AUTOLDB (logical DBD of the auto database)
-EMPLDB2 (logical DBD of the employee database)

The following sample job moves these DBDs to PDS members:

```java
//name    JOB parameters
//MV2PDS1 EXEC PGM=IKJEFT01
//SYSPRINT DD SYSOUT=* 
//SYSSTSNT DD SYSOUT=* 
//O1 DD DISP=SHR,DSN=hlq.dbsdsrc(AUTODB) 
//I1 DD DISP=SHR,PATH='path/AUTODB' 
//O2 DD DISP=SHR,DSN=hlq.dbsdsrc(EMPDB2) 
//I2 DD DISP=SHR,PATH='path/EMPDB2' 
//O3 DD DISP=SHR,DSN=hlq.dbsdsrc(SINDEX11) 
//I3 DD DISP=SHR,PATH='path/SINDEX11' 
//O4 DD DISP=SHR,DSN=hlq.dbsdsrc(SINDEX22) 
//I4 DD DISP=SHR,PATH='path/SINDEX22' 
//O5 DD DISP=SHR,DSN=hlq.dbsdsrc(AUTOLDB) 
//I5 DD DISP=SHR,PATH='path/AUTOLDB' 
//O6 DD DISP=SHR,DSN=hlq.dbsdsrc(EMPLDB2) 
//I6 DD DISP=SHR,PATH='path/EMPLDB2' 
//SYSTIN DD* 
OCOPY INDD(I1) OUTDD(01) 
OCOPY INDD(I2) OUTDD(02) 
OCOPY INDD(I3) OUTDD(03) 
OCOPY INDD(I4) OUTDD(04) 
OCOPY INDD(I5) OUTDD(05) 
OCOPY INDD(I6) OUTDD(06) 
/*
```

2. Generate the DBDs using the DBDGEN utility:
   a. Move the JCL file named AUTDBD to a partitioned data set from which it
      can be run.
   b. Edit the JCL statements as necessary.
   c. Run the job, which executes the DBDGEN procedure.

3. Move the following PSB source files (HFS) to your PSB source library (PDS
   members):
   - AUTPSBAL (for loading the auto database)
   - AUTPSBEL (for loading the employee database)
   - AUTPSB11 (for processing the databases)

   The following example moves these PSBs to PDS members:

```java
//name    JOB parameters
//MV2PDS2 EXEC PGM=IKJEFT01 
//SYSPRINT DD SYSOUT=* 
//SYSSTSNT DD SYSOUT=* 
//O1 DD DISP=SHR,DSN=hlq.psbsrc(AUTPSBAL) 
//I1 DD DISP=SHR,PATH='path/AUTPSBAL' 
//O2 DD DISP=SHR,PATH='path/AUTPSBEL' 
//O3 DD DISP=SHR,DSN=hlq.psbsrc(AUTPSB11) 
//I3 DD DISP=SHR,PATH='path/AUTPSB11' 
//SYSTIN DD* 
OCOPY INDD(I1) OUTDD(01) 
OCOPY INDD(I2) OUTDD(02) 
OCOPY INDD(I3) OUTDD(03) 
/*
```

4. Generate the PSBs by using the PSBGEN utility:
   a. Move the JCL file named AUTPSB to a partitioned data set from which it
      can be run.
   b. Edit the JCL statements if necessary.
c. Run the job, which executes the PSBGEN procedure.

5. Generate the ACBs for the IMS system that are used when running the sample application:
   a. Ensure that DFSACBCP is available in a partitioned data set.
   b. Move the JCL file named AUTACB to a partitioned data set from which it can be run.
   c. Edit the JCL statements if necessary.
   d. Run the job, which executes the ACBGEN procedure.

6. Initial load the databases:
   a. Move the JCL files named AUTLOAD and IV3H103A to a partitioned data set from which they can be run.
   b. Edit the JCL statements if necessary.
   c. Run the AUTLOAD job, which is an IMS batch job. System data sets must be available and the control region must not be running. This job completes the following steps:
      - Scratches old database data sets.
      - Allocates new database data sets.
      - Loads the physical AUTDB and EMPDB2 databases.
      - Resolves and updates logical relationships.
      - Builds the two secondary indexes.

   Because no data exists in the databases yet, the final three steps are null operations and therefore, 0004 return codes are acceptable.

7. Add data to the initialized databases:
   a. Move the JCL file named AUTSEED to a partitioned data set from which it can be run.

   The following sample job moves AUTSEED to a PDS member:

   ```
   //name JOB parameters
   //MV2PDS4 EXEC PGM=IKJEFT01
   //SYSPRINT DD SYSOUT**
   //SYSTSNT DD SYSOUT**
   //O1 DD DISP=SHR,DSN=hlq.library(AUTSEED)
   //I1 DD DISP=SHR,PATH='path/AUTSEED'
   //O2 DD DISP=SHR,DSN=hlq.library(IV3H103A)
   //I2 DD DISP=SHR,PATH='path/IV3H103A'
   //SYSTIN DD*
   OCOPY INDD(I1) OUTDD(01)
   OCOPY INDD(I2) OUTDD(02)
   /*
   ```
   b. Edit the JCL statements in AUTSEED if necessary.
c. Run the AUTSEED job, which executes the DFSDDLT0 procedure. This job completes the following steps:
   - Deletes the root segments, if present.
   - Adds roots and dependent segments to the database using the AUTPSB11 PSB.

You can run this job repeatedly without re-running the AUTLOAD job.

8. Optionally, confirm that the databases loaded correctly:

   a. Move the JCL file named AUTLIST to a partitioned data set from which it can be run.

   The following sample job moves AUTLIST to a PDS member:

   ```
   name  JOB parameters
   /MV2P0DS5 EXEC PGM=IKJEFT01
   /SYSPRINT DD SYSOUT=* 
   /SYSTSNT DD SYSOUT=* 
   /O1 DD DISP=SHR, DSN=hlq.library(AUTLIST)
   /I1 DD DISP=SHR, PATH='path/AUTLIST'
   /SYSTIN DD*
   OCOPY INDD(I1) OUTDD(01)
   /*
   
   b. Edit the JCL statements if necessary.
   
   c. Run the job, which executes the DFSDDLT0 procedure. This job lists segments to the database using the AUTPSB11 PSB.

9. Compile the dynamic allocation members for the databases:

   a. Move the JCL file named AUTODA to a partitioned data set from which it can be run.

   The following sample job moves AUTODA to a PDS member:

   ```
   name  JOB parameters
   /MV2P0DS6 EXEC PGM=IKJEFT01
   /SYSPRINT DD SYSOUT=* 
   /SYSTSNT DD SYSOUT=* 
   /O1 DD DISP=SHR, DSN=hlq.library(AUTODA)
   /I1 DD DISP=SHR, PATH='path/AUTODA'
   /SYSTIN DD*
   OCOPY INDD(I1) OUTDD(01)
   /*
   
   b. Edit the JCL statements if necessary.
   
   c. Run the job.
Appendix B. SQL Keywords

Because the IMS Java SQL parser supports portable SQL, you cannot use any SQL keywords as Java aliases for PCBs, segments, or fields. When you define Java aliases, do not use an SQL keyword. If a PCB, segment, or field has the same name as an SQL keyword, you must explicitly define a different Java alias for it. For information on defining Java aliases, see IMS Version 8: Utilities Reference.

If you use an SQL keyword as a name of a PCB, segment, or field, your application program receives an error when it attempts an SQL query.

The following SQL keywords cannot be used as PCB, segment, or field names:

ABORT  CROSS  IS  REAL
ANALYZE  CURRENT  JOIN  REFERENCES
AND  CURSOR  LAST  RESET
ALL  DECIMAL  LEADING  REVOKE
ALLOCATE  DECLARE  LEFT  RIGHT
ALTER  DEFAULT  LIKE  ROLLBACK
AND  DELETE  LISTEN  SELECT
ANY  DESC  LOAD  SET
ARE  DISTINCT  LOCAL  SETOF
AS  DO  LOCK  SHOW
ASC  DOUBLE  MAX  SMALLINT
ASSERTION  DROP  MIN  SUBSTRING
AT  END  MOVE  SUM
AVG  EXECUTE  NAMES  TABLE
BEGIN  EXISTS  NATIONAL  TO
BETWEEN  EXPLAIN  NATURAL  TRAILING
BINARY  EXTRACT  NCHAR  TRANSACTION
BIT  EXTEND  NEW  TRIM
BOOLEAN  FALSE  NO  TRUE
BOTH  FIRST  NONE  UNION
BY  FLOAT  NOT  UNIQUE
CASCADE  FOR  NOTIFY  UNLISTEN
CAST  FOREIGN  NULL  UNTIL
CHAR  FROM  NUMERIC  UPDATE
CHARACTER  FULL  ON  USER
CHECK  GRANT  OR  USING
CLOSE  GROUP  ORDER  VACUUM
CLUSTER  HAVING  OUTER  VALUES
COLLATE  IN  PARTIAL  VARCHAR
COLUMN  INNER  POSITION  VARYING
COMMIT  INSERT  PRECISION  VERBOSE
CONSTRAINT  INT  PRIMARY  VIEW
COPY  INTEGER  PRIVILEGES  WHERE
COUNT  INTERVAL  PROCEDURE  WITH
CREATE  INTO  PUBLIC  WORK
Appendix C. IMS Java Hierarchical Database Interface

The IMS Java hierarchical database interface is more closely related to the standard DL/I database call interface that is used with other languages, and provides a lower-level access to IMS database functions than the JDBC interface. Using IMS Java hierarchical database interface, you can build segment search arguments (SSAs) and call the functions of the DLIConnection object to read, insert, update, or delete segments. The application has full control to navigate the segment hierarchy.

Although you can use the IMS Java hierarchical database interface to access IMS data, it is recommended that you use JDBC. However, you can use this package if you need more controlled access than the higher-level JDBC package provides.

Related Reading: For detailed information about the classes in the IMS Java hierarchical database interface, see the IMS Java API Specification (Javadoc). Go to the IMS Web site at www.ibm.com/ims and link to the IMS Java page.

The following topics provide additional information:
- "Application Programming Using the DLIConnection Object"
- "Creating a DLIConnection Object"
- "Creating an SSAList Object" on page 136
- "Accessing IMS Data Using SSAs" on page 136

Application Programming Using the DLIConnection Object

To use a DLIConnection object to read, update, insert, and delete segment instances, your application must:
1. Acquire a DLISegment object for each segment using the cloneSegment method on the DLIDatabaseView subclass.
2. Provide a subclass of DLIDatabaseView that defines the segment hierarchy accessed by the application.
3. Create a DLIConnection object to access the database.
4. Create an SSAList object.
5. Invoke the database access methods of the DLIConnection class to read, write, or delete segments from the database.

Creating a DLIConnection Object

You must create a DLIConnection object in one of two ways:
- By providing a DLIDatabaseView object
- By providing the fully-qualified name of the DLIDatabaseView subclass

When you code directly to a DLIConnection object, it is faster to create and pass the DLIDatabaseView object because it simplifies finding the class by its name. Figure 54 on page 136 illustrates how to create a DLIConnection object:
Creating an SSAList Object

SSAs identify the segment to which a DL/I call applies. Because of the hierarchical structure of IMS databases, you often have to specify several levels of SSAs to access a segment at a low level in the hierarchy. An SSAList object is a collection of one or more SSA objects. Use the SSAList object when you make DL/I calls. The SSAList object is also where you specify which database that you want to access within a DLIDatabaseView object by providing the PCB reference name.

Figure 55 shows how to create an SSAList object that will find all “Alpha” cars that were made in 2004:

// Create an SSAList
SSALIST modelSSALIST = SSALIST.createInstance("DealershipDB");

// Construct an unqualified SSA for the Dealer segment
SSA dealerSSA = SSA.createInstance("Dealer");

// Construct a qualified SSA for the Model segment
SSA modelSSA = SSA.createInstance("Model", "CarMake", SSA.EQUALS, "Alpha");

// Add an additional qualification statement
modelSSA.addQualification(SSA.AND, "CarYear", SSA.EQUALS, "1989");

// Add the SSAs to the SSAList
modelSSALIST.addSSA(dealerSSA);
modelSSALIST.addSSA(modelSSA);

Accessing IMS Data Using SSAs

After you create an SSAList object, you can issue database calls by invoking the access method on the DLIConnection object and passing in the following:

- The SSAList object.
- An instance of the segment, which is the intended target of the database call results.

Get the passed-in instance of the segment by calling the cloneSegment method on the DLIDatabaseView subclass.

The following example how to call and print the results using the SSAList object that was built in "Creating an SSAList Object"

```
    DLISegment model = dealerView.cloneSegment("Model");
    boolean recordRead = connection.getUniqueSegment(model, modelSSALIST);
    while (recordRead) {
        System.out.println("Car Name: " + model.getString("modelName"));
    }
```
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.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs
and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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.
Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Trademarks**

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

<table>
<thead>
<tr>
<th>BookManager</th>
<th>MVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>OS/390</td>
</tr>
<tr>
<td>DB2 Universal Database</td>
<td>QMF</td>
</tr>
<tr>
<td>CICS</td>
<td>RACF</td>
</tr>
<tr>
<td>IBM</td>
<td>Rational Rose</td>
</tr>
<tr>
<td>IMS</td>
<td>WebSphere</td>
</tr>
<tr>
<td>IMS/ESA</td>
<td>z/OS</td>
</tr>
<tr>
<td>Library Reader</td>
<td></td>
</tr>
</tbody>
</table>

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, 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.

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

This bibliography includes all the publications cited in this book, including the publications in the IMS library.

- DB2 Universal Database for OS/390 and z/OS: Application Programming Guide and Reference for Java, SC26-9932
- Enterprise COBOL for z/OS and OS/390: Programming Guide, SC27-1412
- IBM Developer Kit for OS/390, Java 2 Technology Edition: New IBM Technology featuring Persistent Reusable Java Virtual Machines, SC34-6034
- z/OS: UNIX System Services Command Reference, SA22-7802
- z/OS: UNIX System Services File System Interface Reference, SA22-7808
- WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications, SA22-7836
- WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management User Interface, SA22-7838

IMS Version 8 Library

<table>
<thead>
<tr>
<th>Order Number</th>
<th>Acronym</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC27-1290</td>
<td>BPE</td>
<td>IMS Version 8: Base Primitive Environment Guide and Reference</td>
</tr>
<tr>
<td>SC27-1291</td>
<td>CR</td>
<td>IMS Version 8: Command Reference</td>
</tr>
<tr>
<td>SC27-1292</td>
<td>CQS</td>
<td>IMS Version 8: Common Queue Server Guide and Reference</td>
</tr>
<tr>
<td>SC27-1293</td>
<td>CSL</td>
<td>IMS Version 8: Common Service Layer Guide and Reference</td>
</tr>
<tr>
<td>SC27-1294</td>
<td>CG</td>
<td>IMS Version 8: Customization Guide</td>
</tr>
<tr>
<td>SC27-1295</td>
<td>DBRC</td>
<td>IMS Version 8: Database Recovery Control (DBRC) Guide and Reference</td>
</tr>
<tr>
<td>LY37-3742</td>
<td>DGR</td>
<td>IMS Version 8: Diagnosis Guide and Reference</td>
</tr>
<tr>
<td>LY37-3743</td>
<td>FAST</td>
<td>IMS Version 8: Failure Analysis Structure Tables (FAST) for Dump Analysis</td>
</tr>
<tr>
<td>SC27-1296</td>
<td>JGR</td>
<td>IMS Version 8: IMS Java Guide and Reference</td>
</tr>
<tr>
<td>GC27-1297</td>
<td>IIV</td>
<td>IMS Version 8: Installation Volume 1: Installation Verification</td>
</tr>
<tr>
<td>GC27-1298</td>
<td>ISDT</td>
<td>IMS Version 8: Installation Volume 2: System Definition and Tailoring</td>
</tr>
<tr>
<td>SC27-1300</td>
<td>MIG</td>
<td>IMS Version 8: Master Index and Glossary</td>
</tr>
<tr>
<td>GC27-1301</td>
<td>MC1</td>
<td>IMS Version 8: Messages and Codes, Volume 1</td>
</tr>
<tr>
<td>GC27-1302</td>
<td>MC2</td>
<td>IMS Version 8: Messages and Codes, Volume 2</td>
</tr>
<tr>
<td>SC27-1303</td>
<td>OTMA</td>
<td>IMS Version 8: Open Transaction Manager Access Guide</td>
</tr>
<tr>
<td>SC27-1304</td>
<td>OG</td>
<td>IMS Version 8: Operations Guide</td>
</tr>
<tr>
<td>GC27-1305</td>
<td>RPG</td>
<td>IMS Version 8: Release Planning Guide</td>
</tr>
<tr>
<td>SC27-1308</td>
<td>URDBTM</td>
<td>IMS Version 8: Utilities Reference: Database and Transaction Manager</td>
</tr>
<tr>
<td>SC27-1309</td>
<td>URS</td>
<td>IMS Version 8: Utilities Reference: System</td>
</tr>
<tr>
<td>GC27-1299</td>
<td>LPS</td>
<td>IMS Version 8: Licensed Program Specifications</td>
</tr>
</tbody>
</table>

Supplementary Publications
Publication Collections

<table>
<thead>
<tr>
<th>Order Number</th>
<th>Description</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>LK3T-7092</td>
<td>CD</td>
<td>IMS Version 8 Softcopy Library</td>
</tr>
<tr>
<td>LK3T-7144</td>
<td>CD</td>
<td>IMS Favorites</td>
</tr>
<tr>
<td>LBOF-7787</td>
<td>Hardcopy and CD</td>
<td>Licensed Bill of Forms (LBOF): IMS Version 8 Hardcopy and Softcopy Library</td>
</tr>
<tr>
<td>SBOF-7788</td>
<td>Hardcopy</td>
<td>Unlicensed Bill of Forms (SBOF): IMS Version 8 Unlicensed Hardcopy Library</td>
</tr>
<tr>
<td>SK2T-6700</td>
<td>CD</td>
<td>OS/390 Collection</td>
</tr>
<tr>
<td>SK3T-4270</td>
<td>CD</td>
<td>z/OS Software Products Collection</td>
</tr>
<tr>
<td>SK3T-4271</td>
<td>DVD</td>
<td>z/OS and Software Products DVD Collection</td>
</tr>
</tbody>
</table>

Accessibility Titles Cited in this Book

<table>
<thead>
<tr>
<th>Order Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA22-7787</td>
<td>z/OS TSO Primer</td>
</tr>
<tr>
<td>SA22-7794</td>
<td>TSO/E User’s Guide</td>
</tr>
<tr>
<td>SC34-4822</td>
<td>ISPF User’s Guide, Volume 1</td>
</tr>
</tbody>
</table>
Index

A
aggregate functions 83
   AS 83
   ASC 83
   AVG 83
   COUNT 83
data types 84
   DESC 83
   GROUP BY 83
   MAX 83
   MIN 83
   ORDER BY 83
renaming 84
result set column 84
result set type 84
SUM 83
AND operator
   IMS rules 83
applications
   message processing, building 16
   programming 135
AS keyword 83, 84
ASC keyword 83
AVG keyword 83

B
BIGINT data type 86
BINARY data type 86
BIT data type 86
byte data type 86

C
CHAR data type 86
CICS
   _CXX_LSYSLIB environment variable 67
   application 69
   applications
      IVP 68
      running 70
      writing 71
CICSPSB DLL 67
configuring for IMS Java 67
DFHJVMPR environment member 68
dljvmpr.props 68
IMS Java overview 67
IVP 68
LIBPATH variable 68
Makefile 67
   running applications 70
   sample application 69
   writing applications 71
CLASSPATH ENVAR keyword 60
COBOL
   See also Enterprise COBOL
   (continued)
   copybook types 88
   mapping to IMS 88
columns
   fields, compared to 73
   relational representation, in 74
com.ibm.connector2.ims.db package 4
com.ibm.ims.application package 4
com.ibm.ims.base package 4
com.ibm.ims.db package 4
com.ibm.ims.db.DLIDatabaseView class 90
configuration
   WebSphere Application Server for z/OS
   environment 49
Connection object 93
control statements
   DLIModel utility 98, 102
   example 116
   format 104
   including 109
   syntax 110
   typical sequence 103
   when to use 103
conversational transactions 21
copybook types 88
COUNT keyword 83
custom service, installing 38

data types
   aggregate functions 84
   conversion 86
   mapped to COBOL 88
database metamodel, XMI 101
databases
   describing to IMS Java 90
DATE data type 86
DB2 Recoverable Resource Manager Services
   attachment facility 15
DB2 UDB for z/OS access
   application programming 32
   committing work 32
   drivers 32
   FSDB2AF DD statement 16
   IMS databases, compared to 32
   JBP region, from a
      configuring 15
      programming model 28
   JMP region, from a
      configuring 15
      programming model 20
   rolling back work 32
   RRSAF 15
DB2 UDB for z/OS stored procedures
   developing 66
   environment variables 60
   IVP 61

© Copyright IBM Corp. 2000, 2005
DB2 UDB for z/OS stored procedures  (continued)

JAVAENV data set  59
overview with IMS Java  59
running  64
sample application
running  63
system requirements  59
DB2_HOME ENVAR keyword  60
DB2Auto application
running  63
DB2AutoClient application
running  63
db2sqljjdbc.properties  62
DBD (database description), sample  91
DD statements  112
  DFSDB2AF  16
  DFSRESLB  61
  JAVAERR  9
  JAVAOUT  9
  STEPLIB  9
dealership sample application
  DB2 UDB for z/OS stored procedures  63
dealership samples
  DBD  91
  EAR, installing  52
  PSB  91
DEBUG=Y
  for JVM debugging  127
debugging
  XMLTrace  124
DELETE keyword  81
example  81
deployment descriptor
example  57
requirements for IMS Java  56
DESC keyword  83
dfjvmpr.props file  68
DFSJMP procedure  9
DFSJVMEV member
  DB2 JDBC driver  16
  IVP changes  9
DFSJVMM MS
  DB2 JDBC driver  16
DFSJVMM MS member
  IVP changes  9
DFSMODEL member  110
DFSRESLB DD statement  61
DL/I data, accessing  136
DL/I status codes
  mapping to exceptions  123
DLIConnection
  creating  135
DLIConnection class  123
DLIDatabaseView class  90
DLIDriver
  loading  93
  registering  94
DLIException class  123
DLIModel IMS Java Report  91
DLIModel Java Report
  database summary  100

DLIModel utility
  control statements  98, 102
  description  97
  DLIModel IMS Java Report  91
  inputs and outputs  97
  installing  110
  limitations  99
  logical relationship example  115
  metadata classes
    creating  97
    output types  100
  overview  97
  procedure  110
  PSB requirements  99
  report  97
  restrictions  99
  running  110, 113
  running as MVS job  110
  running from UNIX System Services  113
  simple example  114
  using  90, 97, 113
DLITypeInfoList class  24
DOUBLE data type  86
DRA (database resource adapter)  49
driver
  registering with DriverManager  94
  DriverManager facility  84
  DSNAM E parameter  111

E

EJB (Enterprise JavaBean)
deployment descriptor requirements  56
Enterprise COBOL  29
  back end  29
  implementing  30
  CALL statement  29
  compiler  30
  front end  29, 30
  JVM, locating  31
  main method  30
  object oriented syntax  29
  performance  31
exceptions
  description  123
  IMSException object
    getAIB method  123
    getFunction method  123
    getStatusC ode method  123
    mapping to DL/I status codes  123
EXEC statement parameters  111

F

Field statement  103, 107
fields
  columns, compared to  73
  default value  80
  in SQL queries  76
  segment qualified  79
float data type  86
Java message processing (JMP) regions (continued)

IVP 8
DFSJMP procedure 9
DFSJVMEV member 9
DFSJVMMS member 9
JVM.out file 9
programming models 19

Java Message Processing (JMP) regions
program switching 33
JAVA_HOME ENVAR keyword 60
java.math.BigDecimal 86
java.sql.Connection interface 84
java.sql.DatabaseMetaData interface 85
java.sql.Date 86
java.sql.Driver interface 85
java.sql.PreparedStatement interface 86
java.sql.ResultSet interface 86
java.sql.ResultSetMetaData interface 86
java.sql.Statement interface 85
java.sql.Time 86
java.sql.Timestamp 86
JAVAEENV data set
creating 59
sample 60
JAVAERR DD statement 9
JAVAOUT DD statement 9

JBP (Java batch processing) regions i
DB2 UDB for z/OS access
application programming 32
configuring 15
programming model 20
description 7
IVP 11
samples.tar file 12
program switching 33
programming models 27

JMP (Java message processing) regions (continued)
DB2 UDB for z/OS access
application programming 32
configuring 15
programming model 20
description 7
IVP 8
DFSJMP procedure 9
DFSJVMEV member 9
DFSJVMMS member 9
JVM.out file 9
samples.tar file 9
program switching 33
programming models 19

JMP applications
message handling
conversational transactions 21
multi-segment messages 23
multiple input messages 25
repeating structures 24
programming models 19
joining segments 79

JVM, debugging
DEBUG=Y 127
log messages 127
reset trace events 127

L

LIBPATH ENVAR keyword 60
logical relationships
DLIModel utility example 115

M

main() method 18
MAX keyword 83
message processing application
building 16
messages
input, defining 16
multi-segment 23
output, defining 17
repeating structures
defining in IMS Java 24
SPA 21
subsequent 23
metadata classes
creating 97
generated 100
MIN keyword 83
multi-segment messages 23

O

object
DLICONnection, creating 135
ODBA (Open Database Access) 49
Options statement 102, 104
OR operator
   IMS rules 83
ORDER BY keyword 83
output messages, defining 17

P
PACKEDDECIMAL data type 86
PARM parameter 111
PCB statement 103
Persistent Reusable Java Virtual Machine 1
prepared statements
   java.sql.PreparedStatement interface 86
   PreparedStatement object 93
Problem Determination 123
PROC statement parameters 111
program switches 33
programming models
   JBP applications
      with rollback 28
      without rollback 27
   JMP applications 19
   DB2 UDB for z/OS data access 20
   IMS data access 20
      with rollback 20
      without rollback 19
WebSphere Application Server for z/OS applications
   bean-managed EJBs 54
   container-managed EJBs 55
   servlets 56
PSB (program specification block)
   metadata class 97
   sample 91
   XMI description 97
PSB statement 102, 106

R
Rational Rose
   metamodel 102
Recoverable Resource Manager Services attachment facility (RRSAF) 15
relational database
   hierarchical database, compared to 73
repeating structures
   accessing 24
   DLITypeInfoList class 24
   dotted notation 24
   sample output message 24
report
   DLIModel utility 97
   res-sharing-scope element 56
   res-type element 56
resource-ref element
   example 57
   requirements for IMS Java 56
ResultSet
   aggregate data types 84
   iterating 93
   TYPE_FORWARD_ONLY 84
   TYPE_SCROLL_INSENSITIVE 84
   ResultSet.getAsciiStream method 86
   ResultSet.getCursorName method 86
   ResultSet.getUnicodeStream method 86
   rows
      relational representation, in 76
      segment instances, compared to 73
   RRSAF (Recoverable Resource Manager Services attachment facility) 15

S
sample applications
   DB2 UDB for z/OS stored procedures 63
samples
   message processing application 16
   samples.tar file
      expanding 2
   secondary index field 109
   SEG statement 103, 106
   segments
      in SQL queries 76
      tables, compared to 73
   segments, selecting multiple 79
   SELECT keyword
      description 77
      example 77
      example query 76
      selecting multiple segments 79
setModifiableAlternatePCB(String) method 33
setup
   WebSphere Application Server for z/OS environment 49
short data type 86
SMALLINT data type 86
SPA 21
   message, defining 21
   SPA (scratch pad area) 21
   conversational transactions for IMS Java 23
SQL
   referring to Java Report 100
SQL (Structured Query Language)
   aggregate functions 83
      argument types 84
      result types 84
   AS 83, 84
   ASC 83
   AVG 83
   COUNT 83
   DELETE 81
   DESC 83
   example query 76
   FROM 81
   GROUP BY 83
   IMS requirements for 76
   INSERT 80
   keyword list 133
   MAX 83
   MIN 83
   ORDER BY 83
   PCB-qualified query 82
   prepared statements 86
SQL (Structured Query Language) (continued)

- PCB-qualified query 82
- segment-qualified fields 80
- segment-qualified fields 79
- SELECT keyword requirements 77
- SUM 83
- supported keywords 76
- UPDATE 81
- WHERE 82
- SQLException 124
- SQLstate 124
- SSA (segment search argument) 79
  - WHERE clause, relation to 82
- SSAList
  - creating an 136
  - DL/I data, accessing 136
- Statement object
  - retrieving 93
- status codes
  - mapping 123
- STDENV DD statement 112
- STDERR DD statement 112
- STDOUT DD statement 112
- stored procedures
  - See DB2 UDB for z/OS stored procedures
- String data type
  - boolean data type 86
- SUM keyword 83
- SYSOUT parameter 111
- system requirements
  - DB2 UDB for z/OS stored procedures 59
  - SYSTSIN DD statement 112

T

- tables
  - relational representation, in 74
  - segments, compared to 73
- TIME data type 86
- TIMESTAMP data type 86
- TINYINT data type 86
- TMSUFFIX ENVAR keyword 60
- tracing
  - IMS Java library methods 126
  - J2EE 47
  - Trace statements, adding 126
  - WebSphere Application Server for z/OS 47
- XMLTrace 124
- transactions
  - conversational 21
- types
  - data, mapped to COBOL 88
  - supported 86

U

- UPDATE keyword 81
  - example 81

V

- VARCHAR data type 86

W

- WebSphere Application Server for z/OS
  - applications 35
    - IVP 39
      - samples 42
    - classpath, setting 45
    - configuring
      - access to IMS 36
      - custom service, installing 38
      - IMS JDBC resource adapter, installing 37
      - prerequisites 36
    - deployment descriptor 56
    - IVP 39
      - application, installing 40
      - data source, installing 39
      - prerequisites 39
      - samples.tar file 39
      - testing 41
    - overview 35
    - restrictions
      - container-managed signon 36
      - java.sql.Connection object 36
      - shared connections 36
    - running your application
      - application, installing 46
      - classpath, setting 45
      - data source, installing 45
      - prerequisites 44
    - sample applications
      - application, installing 43
      - data source, installing 42
      - prerequisites 42
      - testing 44
      - server.policy file 38
      - system requirements 36
      - tracing 47
- WebSphere Application Server for z/OS environment
  - setup 49
  - WHERE keyword 82
    - fields, valid 82
    - operators, valid 82
    - SSAs, relation to 82

X

- Xdfld statement 109
- XMI
  - creating 97
  - metamodel 101
- XMLTrace
  - application 126
  - enabling 125
- XMLTrace class 124
- XMLTrace.enable 125
- XMLTrace.IMS Java library methods 126
- XMLTrace.libTraceLevel 126
Z
ZONEDECIMAL data type  86