OpenEdge Development:
Java Open Clients
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>Preface–1</td>
</tr>
<tr>
<td>1.</td>
<td>Configuring and Deploying Java Open Client Applications</td>
<td>1–1</td>
</tr>
<tr>
<td></td>
<td>Selecting an Open Client Runtime package</td>
<td>1–2</td>
</tr>
<tr>
<td></td>
<td>Preparing to generate proxies for a Java client in Windows using ProxyGen or Batch ProxyGen</td>
<td>1–6</td>
</tr>
<tr>
<td></td>
<td>Preparing to generate proxies for a Java client on UNIX using Batch ProxyGen</td>
<td>1–7</td>
</tr>
<tr>
<td></td>
<td>Building an Open Client application that uses a Java proxy</td>
<td>1–8</td>
</tr>
<tr>
<td></td>
<td>Deploying an Open Client application that uses a Java proxy</td>
<td>1–9</td>
</tr>
<tr>
<td></td>
<td>Sample Java client applications</td>
<td>1–11</td>
</tr>
<tr>
<td>2.</td>
<td>Proxy Objects and Methods</td>
<td>2–1</td>
</tr>
<tr>
<td></td>
<td>Proxy objects</td>
<td>2–2</td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td>2–2</td>
</tr>
<tr>
<td></td>
<td>Proxy methods</td>
<td>2–3</td>
</tr>
<tr>
<td></td>
<td>Connection methods</td>
<td>2–3</td>
</tr>
<tr>
<td></td>
<td>Remote ABL methods</td>
<td>2–3</td>
</tr>
<tr>
<td></td>
<td>Class factory methods</td>
<td>2–4</td>
</tr>
<tr>
<td></td>
<td>Common methods</td>
<td>2–5</td>
</tr>
<tr>
<td></td>
<td>Running methods on session-free AppObjects</td>
<td>2–9</td>
</tr>
<tr>
<td></td>
<td>Session-free threading model</td>
<td>2–9</td>
</tr>
<tr>
<td></td>
<td>Sharing of the connection pool between session-free AppObjects</td>
<td>2–9</td>
</tr>
<tr>
<td></td>
<td>Sample proxy</td>
<td>2–11</td>
</tr>
<tr>
<td></td>
<td>Sample AppObject</td>
<td>2–11</td>
</tr>
<tr>
<td></td>
<td>Sample SubAppObject</td>
<td>2–12</td>
</tr>
<tr>
<td></td>
<td>Sample ProcObject</td>
<td>2–13</td>
</tr>
<tr>
<td>3.</td>
<td>Connecting to an AppServer</td>
<td>3–1</td>
</tr>
<tr>
<td></td>
<td>Connection class</td>
<td>3–2</td>
</tr>
<tr>
<td></td>
<td>Establishing the connection</td>
<td>3–3</td>
</tr>
<tr>
<td></td>
<td>Releasing a session-free Connection object</td>
<td>3–5</td>
</tr>
<tr>
<td></td>
<td>Connection states</td>
<td>3–6</td>
</tr>
</tbody>
</table>
4. Passing Parameters ................................................................. 4–1
   ABL data type mappings ..................................................... 4–2
      INPUT parameters ......................................................... 4–2
      Date and time conversions .............................................. 4–3
      Arrays as parameters .................................................. 4–3
      Unknown value (?) as a parameter ................................... 4–4
      INPUT-OUTPUT and OUTPUT parameters ............................ 4–4
   Passing temp-tables and ProDataSets ................................. 4–8
      Static and dynamic temp-tables and ProDataSets ............... 4–8
      ABL mechanisms for passing temp-tables and ProDataSets .... 4–8
      Java mechanisms for passing temp-tables and ProDataSets ... 4–9
   Mapping temp-table field data types and meta data ............... 4–10

5. Accessing ABL ProDataSets ................................................. 5–1
   ProDataSets and using ProDataGraphs to access them ............ 5–3
      ProDataSet structure and usage ..................................... 5–3
      ProDataGraph function and foundations .......................... 5–3
      ProDataGraph object model ......................................... 5–7
   Preparing and passing ProDataSets as ProDataGraph parameters .5–9
      Preparing and managing a ProDataGraph parameter .......... 5–9
      Managing ProDataObject mappings to temp-tables .......... 5–11
      Passing a ProDataGraph as OUTPUT .............................. 5–12
      Passing a ProDataGraph as INPUT or INPUT-OUTPUT ..... 5–17
   Updating a ProDataSet ..................................................... 5–24
      Typical cycle for ProDataSet updates ............................ 5–24
      Managing a ProDataGraph through the update cycle ..... 5–24
      Checking for errors .................................................... 5–26
      Finding a row in a ProDataGraph ................................. 5–27
      Adding a row to a ProDataGraph ................................. 5–28
      Deleting a row from a ProDataGraph ............................ 5–28
   Passing temp-tables as ProDataGraph parameters ................. 5–29
   ProDataGraph class .......................................................... 5–30
      Constructors ............................................................. 5–30
      Inherited methods ..................................................... 5–31
      Extended methods ..................................................... 5–31
   ProDataObject class ........................................................ 5–36
      Inherited methods ..................................................... 5–36
      Extended methods ..................................................... 5–44
   ProChangeSummary class .................................................. 5–48
      Inherited methods ..................................................... 5–48
      Extended method ...................................................... 5–49
   ProDataGraphMetaData class ............................................. 5–50
      Constructor .............................................................. 5–50
      Methods ................................................................. 5–50
   ProDataObjectMetaData class ............................................ 5–53
      Constructor .............................................................. 5–53
      Methods ................................................................. 5–55
   ProDataRelationMetaData class ......................................... 5–59
      Constructors ............................................................ 5–59
      Methods ................................................................. 5–61
   Using Java SDO classes to access Property meta data ............ 5–63
      Type interface .......................................................... 5–63
      Property interface .................................................... 5–64
   Sample application ........................................................ 5–66
6. Extending Proxy Objects .......................................................... 6–1
   Example of extending proxy objects ...................................... 6–2

7. Accessing Proxy Properties .................................................... 7–1
   Setting properties on the command line ................................. 7–2
   RunTimeProperties and Connection classes ............................ 7–3
      RunTimeProperties class ........................................... 7–3
      Connection class .................................................... 7–3
   Accessing properties with methods .................................... 7–4
      General property accessor methods for the RunTimeProperties class 7–4
      General property accessor methods for Connection objects ....... 7–5
   Available properties ..................................................... 7–6
   Accessing properties using purposed accessor methods .......... 7–6
   Relationship between the RunTimeProperties class and the Connection class 7–12
   Modifying an instantiated Connection object’s properties ....... 7–14
   Alphabetical listing of properties ..................................... 7–15
      PROGRESS.Session.AppServerKeepalive .......................... 7–15
      PROGRESS.Session.certificateStore ............................... 7–15
      PROGRESS.Session.connectionLifetime ......................... 7–15
      PROGRESS.Session.enableTracing ............................... 7–16
      PROGRESS.Session.IdleConnectionTimeout ...................... 7–16
      PROGRESS.Session.initialConnections ......................... 7–16
      PROGRESS.Session.logFileName .................................. 7–16
      PROGRESS.Session.loggingLevel .................................. 7–17
      PROGRESS.Session.maxConnections ............................... 7–17
      PROGRESS.Session.minConnections ............................... 7–17
      PROGRESS.Session.noHostVerify .................................. 7–18
      PROGRESS.Session.noSslSessionReuse ............................ 7–18
      PROGRESS.Session.nsClientMaxPort ................................ 7–18
      PROGRESS.Session.nsClientMinPort ................................ 7–18
      PROGRESS.Session.nsClientPicklistExpiration .................. 7–19
      PROGRESS.Session.nsClientPicklistSize ......................... 7–19
      PROGRESS.Session.nsClientPortRetry ............................ 7–19
      PROGRESS.Session.nsClientPortRetryInterval ................... 7–19
      PROGRESS.Session.proxyHost ....................................... 7–19
      PROGRESS.Session.proxyPassword .................................. 7–20
      PROGRESS.Session.proxyPort ....................................... 7–20
      PROGRESS.Session.proxyUserId .................................... 7–21
      PROGRESS.Session.requestWaitTimeout .......................... 7–21
      PROGRESS.Session.sessionModel .................................. 7–22
      PROGRESS.Session.waitForBusy .................................... 7–22

8. Handling Errors ................................................................. 8–1
   Exceptions ........................................................................ 8–2
   General ABL exceptions ................................................... 8–3
      Exception class hierarchy ........................................... 8–3
      Exception descriptions .............................................. 8–3
   Output ResultSet exceptions ........................................... 8–5
9. **Using SmartDataObjects from Java Clients** .................................................. 9–1
   What is a SmartDataObject? ................................................................. 9–2
   Accessing remote SmartDataObjects from Java ................................. 9–3
      Requirements for accessing SmartDataObjects ................................. 9–3
      Example Java access to a SmartDataObject ..................................... 9–4
   SmartDataObject access tools and documentation ............................. 9–5
      Objects required to access SmartDataObjects ................................ 9–5
      Developing and deploying a SmartDataObject-aware application ......... 9–6
   Writing a SmartDataObject-aware Java client ................................... 9–7
   Connecting to an AppServer using an SDOAppObject ....................... 9–8
   Creating an SDOResultSet object ...................................................... 9–9
      The _createSDOResultSet() method .............................................. 9–9
   Setting and getting values on the SDOParameters object .................. 9–10
   Understanding SDOResultSet scrolling modes .................................. 9–13
   Understanding SDOResultSet stateless mode .................................... 9–14
   Working with SDOResultSet objects ............................................... 9–16
      Detaching SDOResultSet objects [extension] ................................. 9–16
   Navigating SDOResultSet rows ....................................................... 9–17
   Getting SDOResultSet column values ............................................. 9–18
   Updating SDOResultSet objects ...................................................... 9–19
   Miscellaneous management methods .............................................. 9–24
   Meta data methods ................................................................. 9–27
   Multi-threading ................................................................. 9–30
   Errors and exceptions ............................................................. 9–31
      SDOResultSet exceptions ......................................................... 9–31
      Typical errors by method category ............................................. 9–32
   Choosing JDBC ResultSets or SDOResultSets to access OpenEdge data ... 9–34
   Accessing the sample application .................................................. 9–35

10. **Building Java applets as Open Clients** ................................. 10–1
    Asking for permission in an applet ............................................... 10–2
       Asking for permission in Microsoft Internet Explorer ..................... 10–2
       Asking for permission in Netscape version 6 ................................ 10–3
       Supporting both browsers in one applet ..................................... 10–4
    Signing an Open Client applet ..................................................... 10–5
    Adding your applet to an HTML page ............................................ 10–6
       Microsoft Internet Explorer ..................................................... 10–6
       Navigator ................................................................. 10–7
       Supporting both browsers with the same page ................................ 10–8
    Sample applet ................................................................. 10–9

11. **Using the Open Client Java OpenAPI to Directly Access the AppServer** ........................ 11–1
    Overview of Java OpenAPI classes ............................................. 11–2
    Connecting to the AppServer ..................................................... 11–3
    Running procedures and user-defined functions ................................ 11–4
       Running a non-persistent procedure on an OpenAppObject .......... 11–5
       Running a persistent procedure (OpenProcObject) on an OpenAppObject .................................................. 11–5
       Running an internal procedure or user-defined function on an OpenProcObject .................................................. 11–6
Tables

Table 1–1: Open Client package options .................................................. 1–3
Table 1–2: Security supported by package ................................................. 1–4
Table 4–1: ABL-to-Java data type mapping for INPUT parameters ................. 4–2
Table 4–2: GeorgianCalendar object processing ........................................ 4–3
Table 4–3: ABL-to-Java data type mapping for INPUT-OUTPUT and OUTPUT parameters ............................................................. 4–5
Table 4–4: ABL with array-to-Java data-type mapping for INPUT-OUTPUT and OUTPUT array parameters ......................................................... 4–6
Table 4–5: Parameter array constants for temp-table field metadata ................ 4–10
Table 5–1: OpenEdge classes that extend Java SDO ..................................... 5–6
Table 5–2: ABL ProDataSet mappings to Java ProDataGraph ......................... 5–9
Table 5–3: Java ProDataObject mappings to temp-table fields ...................... 5–11
Table 7–1: Available properties ............................................................... 7–6
Table 7–2: Session methods ................................................................. 7–7
Table 7–3: Thread control methods .......................................................... 7–9
Table 7–4: Proxy server specification methods .......................................... 7–9
Table 7–5: Secure Sockets Layer management methods ................................. 7–10
Table 7–6: Tracing methods ................................................................. 7–10
Table 7–7: Relationship between RunTimeProperties and Connection object properties ................................................................. 7–12
Table 7–8: Relationship between Connection object properties and session model ................................................................. 7–14
Table 7–9: requestWaitTimeout property values ........................................ 7–21
Table 7–10: sessionModel property values ................................................ 7–22
Table 8–1: General ABL exceptions for Java proxies .................................... 8–3
Table 9–1: Navigating SDOResultSet rows ................................................ 9–17
Table 9–2: Managing operations on SDOResultSet rows ............................... 9–21
Table 9–3: SDOResultSet methods that support Batch mode ......................... 9–23
Table 9–4: Standard JDBC 2 management methods ..................................... 9–24
Table 9–5: Nonstandard SDOResultSet management methods ...................... 9–25
Table 9–6: Standard JDBC meta data methods .......................................... 9–27
Table 9–7: Nonstandard SDOResultSetMetaData methods ......................... 9–28
Table 9–8: Output ResultSet exceptions for SDOResultSet ......................... 9–31
Table 11–1: Java OpenAPI Classes .......................................................... 11–2
Table 11–2: ABL parameter data types for the OpenAPI ............................... 11–17
Table 11–3: User-defined function return types for the OpenAPI .................... 11–19
Table 11–4: Data type mappings between ABL and Java array types ................ 11–20
Table C–1: SQL ResultSet data type mappings for temp-table fields .......... C–2
Table C–2: Passing an INPUT temp-table as an SQL ResultSet .................... C–6
Table C–3: Passing an OUTPUT temp-table as an SQL ResultSet .................. C–6
Table C–4: Passing an INPUT-OUTPUT temp-table as an SQL ResultSet ....... C–7
Table C–5: Mapping between SQL and ABL schema data types for standard JDBC ResultSet .......................................................... C–11
Table C–6: ProResultSet type conversions for temp-table fields .................... C–17
Table C–7: ABL to SQL data type mappings .............................................. C–19
Table C–8: Comparing some array and flat model methods .......................... C–21
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5–1</td>
<td>ProDataGraph object model</td>
<td>5–8</td>
</tr>
<tr>
<td>Figure 9–1</td>
<td>Java Open Client access to a remote SmartDataObject</td>
<td>9–3</td>
</tr>
</tbody>
</table>
Procedures

AccountInfo.p ................................................................. C–23
This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Example procedures
- Third party acknowledgements
Purpose

The Open Client Toolkit (a component of OpenEdge® Studio) exposes AppServer functionality to Open Clients (non-Progress clients). As discussed in *OpenEdge Development: Open Client Introduction and Programming*, the ProxyGen tool allows you to generate proxy objects for Java (and other) Open Clients. These proxy objects encapsulate remote ABL procedures and functions supported on an AppServer. Your Java Open Client application can then access these AppServer procedures and functions through methods of the generated proxy objects.

This book contains Java-specific information about creating and using Open Clients. It describes how to develop Java Open Clients and how to provide access to OpenEdge ABL business logic from Java clients.

Audience

This book is intended for Java programmers who want to develop Open Clients and OpenEdge developers who want to provide access to their ABL business logic from Java clients.

Before reading this book, you should be familiar with *OpenEdge Development: Open Client Introduction and Programming* and *OpenEdge Getting Started: Application and Integration Services*.

Organization

Chapter 1, “Configuring and Deploying Java Open Client Applications”

Describes configuration prerequisites specific to Java Open Client development and deployment.

Chapter 2, “Proxy Objects and Methods”

Describes the objects and methods generated by ProxyGen for a Java Open Client.

Chapter 3, “Connecting to an AppServer”

Describes how to connect to an AppServer, by instantiating an AppObject and, optionally, a Connection object.

Chapter 4, “Passing Parameters”

Describes mappings for basic data types and static and dynamic temp-tables when data is passed between a Java application and OpenEdge.

Chapter 5, “Accessing ABL ProDataSets”

Describes how to pass and manage ProDataSet and temp-table parameters as OpenEdge ProDataGraph objects.

Chapter 6, “Extending Proxy Objects”

Describes how to extend the AppObjects, SubAppObjects, and ProcObjects created by ProxyGen for Java applications.
Chapter 7, “Accessing Proxy Properties”

Describes the different ways you can access the proxy properties that govern behavior across an entire application.

Chapter 8, “Handling Errors”

Describes error handling in Java Open Client applications.

Chapter 9, “Using SmartDataObjects from Java Clients”

Describes how to access and use an interface that enables any Java™ application to access a remote OpenEdge SmartDataObject. The supported interface is based on the Java Database Connectivity (JDBC) 2 ResultSet. The interface includes many of the standard ResultSet methods and adds a number of extensions.

Chapter 10, “Building Java applets as Open Clients”

Describes the process for building a Java applet that runs as an Open Client; that is, the applet can make proxy method calls to remote AppServer procedures from the Web browser environment. You must include additional code to allow the Web browser to accept the Open Client applet for execution.

Chapter 11, “Using the Open Client Java OpenAPI to Directly Access the AppServer”

Describes an API for accessing application services on the AppServer from a Java client without the need to generate Open Client proxies using ProxyGen.

Appendix A, “Accessing a SmartDataObject API Directly.”

Describes the requirements to access SmartDataObjects from any Open Client that needs to access SmartDataObject extensions.


Describes managing certificate store files and converting digital certificates. If your Open Client application uses HTTPS (SSL), you must provide digital certificates with the application.

Appendix C, “Passing Temp-tables as SQL ResultSet Parameters”

Describes how to pass and manage temp-tables as SQL ResultSet objects.
Using this manual

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation, see the OpenEdge Product Documentation Overview page on PSDN: http://communities.progress.com/pcom/docs/DOC-16074.

References to ABL compiler and run-time features

ABL is both a compiled and interpreted language that executes in a run-time engine that the documentation refers to as the ABL Virtual Machine (AVM). When documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler’s allowance for parameter passing and the AVM’s possible response to that parameter passing at run time: “ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error.” The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: “The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets.”

References to ABL data types

ABL provides built-in data types, pre-defined class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever integer appears, this is a reference to the INTEGER or INT64 data type.
- Wherever decimal appears, this is a reference to the DECIMAL data type.
- Wherever numeric appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to pre-defined class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.
## Typographical conventions

This manual uses the following typographical conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td><strong>SMALL, BOLD CAPITAL LETTERS</strong></td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td><strong>KEY1+KEY2</strong></td>
<td>A plus sign between key names indicates a <em>simultaneous</em> key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td><strong>KEY1 KEY2</strong></td>
<td>A space between key names indicates a <em>sequential</em> key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
<tr>
<td><strong>Syntax:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Fixed width</strong></td>
<td>A fixed-width font is used in syntax statements, code examples, system output, and filenames.</td>
</tr>
<tr>
<td><em>Fixed-width italics</em></td>
<td>Fixed-width italics indicate variables in syntax statements.</td>
</tr>
<tr>
<td><em>Fixed-width bold</em></td>
<td>Fixed-width bold indicates variables with special emphasis.</td>
</tr>
<tr>
<td><strong>UPPERCASE fixed width</strong></td>
<td>Uppercase words are ABL keywords. Although these are always shown in uppercase, you can type them in either uppercase or lowercase in a procedure.</td>
</tr>
<tr>
<td>![three arrows]</td>
<td>This icon (three arrows) introduces a multi-step procedure.</td>
</tr>
<tr>
<td>![single arrow]</td>
<td>This icon (one arrow) introduces a single-step procedure.</td>
</tr>
<tr>
<td><strong>Period (.) or colon (:)</strong></td>
<td>All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.</td>
</tr>
<tr>
<td>![large brackets]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>![small brackets]</td>
<td>Small brackets are part of ABL.</td>
</tr>
<tr>
<td>![large braces]</td>
<td>Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.</td>
</tr>
<tr>
<td>![small braces]</td>
<td>Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.</td>
</tr>
</tbody>
</table>
Example procedures

This manual provides numerous example procedures that illustrate syntax and concepts. You can access the example files and details for installing the examples from the Documentation and Samples directory (doc_samples) on the OpenEdge product DVD.

After you install the examples, you can find the Java Open Client samples in the following location:

```
Doc_and_Samples_Install/src/samples/open4gl/java/
```

To find the samples on the PSDN WEb site, start from the OpenEdge Product Documentation Overview page: http://communities.progress.com/pcom/docs/DOC-16074.
Third party acknowledgements

OpenEdge includes AdventNet - Agent Toolkit licensed from AdventNet, Inc. http://www.adventnet.com. All rights to such copyright material rest with AdventNet.

OpenEdge includes ANTLR (Another Tool for Language Recognition) software Copyright © 2003-2006, Terence Parr All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Concurrent Java software Copyright 1994-2000 Sun Microsystems, Inc. All Rights Reserved. -Neither the name of or trademarks of Sun may be used to endorse or promote products including or derived from the Java Software technology without specific prior written permission; and Redistributions of source or binary code must contain the above copyright notice, this notice and the following disclaimers: This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT, ARE HEREBY EXCLUDED. SUN MICROSYSTEMS, INC. AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN MICROSYSTEMS, INC. OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN MICROSYSTEMS, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

OpenEdge includes DataDirect software Copyright © 1991-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC Type 4 driver); Copyright © 1993-2009 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC); Copyright © 1988-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for ODBC); and Copyright © 1988-2007 Progress Software
Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect64 for ODBC).

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC software, which include ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC software, which include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2006 The OpenSSL Project. All rights reserved. And Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

OpenEdge includes DataDirect products for the Microsoft SQL Server database which contain a licensed implementation of the Microsoft TDS Protocol.

OpenEdge includes software authored by David M. Gay. Copyright © 1991, 2000, 2001 by Lucent Technologies (dtoa.c); Copyright © 1991, 1996 by Lucent Technologies (g_fmt.c); and Copyright © 1991 by Lucent Technologies (rm_prod.s). Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes software authored by David M. Gay. Copyright © 1998-2001 by Lucent Technologies All Rights Reserved (decstrtod.c; strtodg.c); Copyright © 1998, 2000 by Lucent Technologies All Rights Reserved (decstrtol.c; strtol.c); Copyright © 1998 by Lucent Technologies All Rights Reserved (dmisc.c; gdtoc.h; gethex.c; gmisc.c; sum.c); Copyright © 1998, 1999 by Lucent Technologies All Rights Reserved (gdtoc.c; misc.c; smisc.c; ulp.c); Copyright © 1998-2000 by Lucent Technologies All Rights Reserved (gdtocimp.h); Copyright © 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Full copies of these licenses can be found in the installation directory, in the c:/OpenEdge/licenses folder. Permission to use, copy, modify, and distribute this software and its documentation for any purpose without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
OpenEdge includes http package software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.


OpenEdge includes Infragistics NetAdvantage for .NET v2009 Vol 2 Copyright © 1996-2009 Infragistics, Inc. All rights reserved.

OpenEdge includes JSTL software Copyright 1994-2006 Sun Microsystems, Inc. All Rights Reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes OpenSSL software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Quartz Enterprise Job Scheduler software Copyright © 2001-2003 James House. All rights reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. This product uses and includes within its distribution, software developed by the Apache Software Foundation (http://www.apache.org/).

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.
OpenEdge includes Sonic software, which includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan,” “FOP,” “The Jakarta Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

OpenEdge includes Sonic software, which includes software Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project (http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names “Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the Products without prior written permission. For written permission, please contact info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.

OpenEdge includes Sonic software, which includes software developed by IBM. Copyright © 1995-2003 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright © 2000-2001 ModelObjects Group. All rights reserved. The name “ModelObjects” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “ModelObjects”, nor may “ModelObjects” appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.
OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes Sonic software, which includes software developed by the University Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

OpenEdge includes UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes Vermont Views Terminal Handling Package software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

OpenEdge includes XML Tools, which includes versions 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which are available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license can also be found in the

Preface–11
installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

OpenEdge includes XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the "License"); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.

OpenEdge includes YAJL software Copyright 2007, Lloyd Hilaiel. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. Neither the name of Lloyd Hilaiel nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Configuring and Deploying Java Open Client Applications

This chapter describes configuration prerequisites specific to Java™ Open Client development and deployment, as detailed in the following sections:

- Selecting an Open Client Runtime package
- Preparing to generate proxies for a Java client in Windows using ProxyGen or Batch ProxyGen
- Preparing to generate proxies for a Java client on UNIX using Batch ProxyGen
- Building an Open Client application that uses a Java proxy
- Deploying an Open Client application that uses a Java proxy
- Sample Java client applications
Selecting an Open Client Runtime package

Based on the run-time configuration you want to support, you have the option of selecting any of the following Open Client Runtime distribution packages:

- **Basic Open Client Runtime packages** — Supports the AppServer and AppServerDC URL connection protocols on an intranet.

- **SSL Open Client Runtime packages** — Supports the AppServer, AppServerS, AppServerDC, and AppServerDCS URL connection protocols on an intranet. These packages support user-implemented authentication to a Web server and Proxy servers.

- **HTTP Open Client Runtime packages** — Supports the AppServer, AppServerDC, and HTTP URL connection protocols for both the Internet or an intranet. These packages support user-implemented authentication to a Web server and Proxy servers.

- **HTTPS Lite Open Client Runtime packages** — Supports the AppServer, AppServerDC, HTTP, and HTTPS URL connection protocols for both the Internet or an intranet. These packages support a minimal set of HTTPS (SSL) functionality, for faster downloading. Progress Software Corporation recommends you use these packages for applets when minimizing download time is critical.

The packages above support many, but not all, SSL-enabled Web servers. The SSL configuration for the individual Web server and the digital certificate used to authenticate the Web server identity determines whether you can use these packages.

The packages above support user-implemented authentication to a Web server, data encryption, and Proxy servers.

- **HTTPS Standard Open Client Runtime packages** — Supports the AppServer, AppServerDC, HTTP, and HTTPS URL connection protocols for both the Internet or an intranet. These packages provide a full set of HTTPS (SSL) functionality. Use them when download time is not critical.

These packages support most SSL-enabled Web servers. The individual Web server's SSL configuration and the digital certificate used to authenticate the identity of the Web server determines whether you can use these packages.

These packages support user-implemented authentication to a Web server, data encryption, and Proxy servers.

**Note:** If you use the HTTP, HTTPS Lite, or HTTPS Standard Packages, you must have a Web server hosting the AppServer Internet Adapter (AIA) Java servlet to support the HTTP/S protocol. Also, for every Web server that hosts an AIA servlet and supports the HTTPS protocol, the client machine must have the root digital certificate of the Certificate Authority that issued the digital certificate for the Web server.

As you develop your HTTPS, AppServerS, and AppServerDCS applications, you should consider that the application deployer might require certain data encryption and digital signature algorithms when they configure their Web server’s.

Your HTTPS requirements might exceed the capabilities of the HTTPS Lite package you select. Deployers also might require the use of SSL version 2 or TLS protocols instead of the standard SSL version 3. If any of these requirements exists, you must use the HTTPS Standard packages.
Selecting an Open Client Runtime package

For more information about selecting a package, see Table 1–1. It lists types of supported applications, supported protocols, and Open Client Runtime packages you would use depending on the run-time configuration of your application and the protocols you want to support.

### Table 1–1: Open Client package options

<table>
<thead>
<tr>
<th>If your application run-time configuration is . . .</th>
<th>And the supported protocol is . . .</th>
<th>The Open Client Runtime package you use is . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Java application, Java servlet, or Java applet running in the Netscape browser</td>
<td>AppServer or AppServerDC</td>
<td>Basic (java/o4glrt.jar) (Recommended for intranet network configurations)</td>
</tr>
<tr>
<td></td>
<td>AppServerS or AppServerDCS</td>
<td>SSL (java/o4glrts.jar) (Recommended for intranet network configurations)</td>
</tr>
<tr>
<td></td>
<td>HTTP</td>
<td>HTTP (java/o4glrth.jar)</td>
</tr>
<tr>
<td></td>
<td>HTTPS</td>
<td>HTTPS Lite (java/o4glrths1.jar) — OR — HTTPS Standard (java/o4glrths.jar)</td>
</tr>
<tr>
<td>A Java applet running in the Internet Explorer browser</td>
<td>AppServer or AppServerDC</td>
<td>Basic (java/o4glrt.cab) (Recommended for Intranet network configurations)</td>
</tr>
<tr>
<td></td>
<td>AppServerS or AppServerDCS</td>
<td>SSL (java/o4glrts.cab) (Recommended for Intranet network configurations)</td>
</tr>
<tr>
<td></td>
<td>HTTP</td>
<td>HTTP (java/o4glrth.cab)</td>
</tr>
<tr>
<td></td>
<td>HTTPS</td>
<td>HTTPS Lite (java/o4glrths1.cab) — OR — HTTPS Standard (java/o4glrths.cab)</td>
</tr>
</tbody>
</table>

All packages support the AppServer protocol, and the HTTPS packages support HTTP.

For example, based on the options described in Table 1–1:

- If you have an Intranet UNIX Java application that requires only AppServer support, you would use o4glrt.jar.
- If you have an Internet Explorer Java Applet that requires HTTP and limited HTTPS support, you would use o4glrths1.cab.
- If you have a Java application that uses AppServer, AppServerDC, or HTTP, you would use o4glrth.jar.
- If you have a Java application that uses SSL-based AppServer connections (AppServerS or AppServerDCS), you would use o4glrts.jar.
If you are using OpenEdge Architect, you will also need to copy the following third-party Eclipse jar files to the same directory where you copy the Java Open Client Runtime package you selected. These files reside in the specified OpenEdge installation directory, as shown:

```
OpenEdge-install-directory/java/ext/
    common.jar
    commonj.sdo.jar
   .ecore.jar
   .ecore.change.jar
   .ecore.sdo.jar
   .ecore.resources.jar
   .ecore.xmi.jar
```

Table 1–2 lists the security supported by each Open Client Runtime package.

<table>
<thead>
<tr>
<th>For this package...</th>
<th>This security is supported...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>User-implemented authentication using a userid and password</td>
</tr>
<tr>
<td>SSL</td>
<td>• Authentication that includes:</td>
</tr>
<tr>
<td></td>
<td>– User-implemented authentication using a userid and password</td>
</tr>
<tr>
<td></td>
<td>– SSL V2, SSL V3, and TLS V1, Digital Signatures: RSA w. MD5, RSA w. SHA1, DSA 2. SHA1, and</td>
</tr>
<tr>
<td></td>
<td>Key Exchange: RSA, Diffie-Hellman</td>
</tr>
<tr>
<td></td>
<td>• Data encryption using RC4-128 bit key, RC4-40 bit key, DES-56 bit key, DES-40 bit key,</td>
</tr>
<tr>
<td></td>
<td>Triple-DES-168 bit key, RC2-40 bit key</td>
</tr>
<tr>
<td>HTTP</td>
<td>• Authentication that includes:</td>
</tr>
<tr>
<td></td>
<td>– User-implemented authentication using a userid and password</td>
</tr>
<tr>
<td></td>
<td>– Basic authentication to the AIA's Web server</td>
</tr>
<tr>
<td></td>
<td>– Basic Proxy server authentication</td>
</tr>
<tr>
<td></td>
<td>• Proxy servers</td>
</tr>
</tbody>
</table>
### Table 1–2: Security supported by package

<table>
<thead>
<tr>
<th>For this package . . .</th>
<th>This security is supported . . .</th>
</tr>
</thead>
</table>
| HTTPS Lite (limited security) | • Authentication that includes:  
  – User-implemented authentication using a userid and password  
  – Basic Proxy server authentication  
  – HTTP basic authentication to the AIA’s Web server  
  – HTTPS using SSL V3, RSA with MD5 digital signatures, and RSA Key Exchange  
• Data encryption using RC4-128 bit key and RC4-40 bit key  
• Proxy servers |
| HTTPS Standard (full security) | • Authentication that includes:  
  – User-implemented authentication using a userid and password  
  – Basic Proxy server authentication  
  – HTTP basic authentication to the AIA’s Web server  
  – HTTPS using SSL V2, SSL V3, and TLS V1, Digital Signatures: RSA w. MD5, RSA w. SHA1, DSA 2. SHA1, and Key Exchange: RSA, Diffie-Hellman  
• Data encryption using RC4-128 bit key, RC4-40 bit key, DES-56 bit key, DES-40 bit key, Triple-DES-168 bit key, RC2-40 bit key  
• Proxy servers |
Preparing to generate proxies for a Java client in Windows using ProxyGen or Batch ProxyGen

There are several system configuration steps needed before you can generate proxies with ProxyGen.

To configure your Windows environment to generate a Java proxy:

1. Install Progress® OpenEdge® Studio, OpenEdge Development Server, or 4GL Development on a Windows system where you plan to run ProxyGen.
   
   With these products, OpenEdge installs the JavaSoft Java Development Kit (JDK), which ProxyGen uses by default to compile a Java proxy.

2. If you want to use an alternate Java compiler to compile your Java proxies, install the appropriate JDK.
   
   If you install an alternate JDK for ProxyGen to compile your Java proxies, you must specify your compiler location and CLASSPATH on the Java tab of the Generate dialog box of ProxyGen. For more information, see the section about the Java tab in OpenEdge Development: Open Client Introduction and Programming.

3. Make sure the ABL r-code for the proxy is accessible to your system.

4. Make sure both the Propath syntax and the r-code paths relative to the Propath setting (defined in the project file) are valid on this system.

5. Run ProxyGen from the icon.
   
   Or
   
   Run Batch ProxyGen by executing the bproxygen command with an existing project file.

For more information, see the chapter on generating proxies and Web service definitions in OpenEdge Development: Open Client Introduction and Programming.
Preparing to generate proxies for a Java client on UNIX using Batch ProxyGen

There are several system configuration steps needed before you can generate proxies with ProxyGen.

To configure your UNIX environment to generate a predefined Java proxy using Batch ProxyGen:

1. Install OpenEdge Development Server or 4GL Development on a UNIX system where you plan to run Batch ProxyGen.

2. Make sure you have a JDK on your system, for ProxyGen to compile your Java proxies.
   If you already installed an OpenEdge product that included a JDK, you do not need to perform this step.

3. Using ProxyGen in Windows, specify the compiler location and CLASSPATH, and save the ProxyGen project file (.xpxg). For more information, see OpenEdge Development: Open Client Introduction and Programming.

4. Copy the ProxyGen project file (.xpxg) to any directory.

5. Make sure the ABL r-code for the proxy is accessible to your system.

6. Make sure both the Propath syntax and the r-code paths relative to the Propath setting (defined in the project file) are valid on this system.
   In the Propath on UNIX, Windows drive letters are ignored and backslashes (\) are automatically changed to forward slashes (/). You also can use dot (.) as a Propath component on UNIX.

7. Run Batch ProxyGen by executing the bproxygen command. For more information, see the chapter on generating proxies and Web service definitions in OpenEdge Development: Open Client Introduction and Programming.
Building an Open Client application that uses a Java proxy

To build an Open Client application that uses a Java proxy requires the OpenEdge Toolkit and a properly configured Java SDK.

To build a client application that uses a Java proxy:

1. Set up the Java client development environment.

   The OpenEdge ToolKit installation includes the JavaSoft JDK; however, it does not set up the system for general use of the JDK. If you are developing your client application on the same machine where you installed OpenEdge ToolKit and you want to use the JDK for your client application development, see the JavaSoft documentation for information on setting up the environment to use this JDK.

2. To build the client application on a machine other than the proxy generation machine, you must:
   a. Copy the proxy .class files to a directory that mirrors the package hierarchy specified on the Java tab of the Generate dialog box in ProxyGen during the generation of the proxy. The package name implies a directory structure that you must maintain when you copy the proxy. You also can put the class files into a .zip or .jar file, as with any other Java classes.
   b. Copy the proxy .java files to any directory on the system, typically together with the corresponding .class files.

   **Note:** ProxyGen provides the proxy .java files only for documentation. Do not attempt to modify and rebuild your proxy from these files.
   c. Select and copy the Java Open Client Runtime package to any directory on the system. For information on selecting a Java Open Client Runtime package, see the “Selecting an Open Client Runtime package” section on page 1–2.

3. Update the Java CLASSPATH setting for the client application’s development environment to include the following:
   a. The directory containing the proxy (.class files) or the zip or jar file from Step 2. For class files, this is the directory above the package hierarchy.
   b. The Open Client Runtime package you copied in Step 2.

4. Write, compile, and execute the client application. (This is the focus of the remainder of this book.)
Deploying an Open Client application that uses a Java proxy

To deploy a client application that uses a Java proxy, you must first perform the following steps in the specified order, on the system where you plan to run the application.

To deploy a client application that uses a Java proxy:

1. Install a Java Virtual Machine (JVM). Many platforms already come with a JVM. For more information on the available Java run-time environment (JRE) for your platform, see the sections on Java requirements in OpenEdge Getting Started: Installation and Configuration.

2. Copy the required files for the client application and proxy to the deployment system:
   a. Copy the client application.
   b. Copy the proxy .class files to a directory that mirrors the package hierarchy specified on the Java tab of the Generate dialog box in ProxyGen during the generation of the proxy. The package name implies a directory structure that you must maintain when you copy the proxy. You also can put the class files into a .zip or .jar file, as with any other Java classes.
   c. Copy the same Java Open Client Runtime package that you used to build the application to any directory on your system. See the “Building an Open Client application that uses a Java proxy” section on page 1–8.
   d. Copy the following third-party Eclipse jar files to the same directory where you copied the Java Open Client Runtime package in Step c, above. These files reside in the specified OpenEdge installation directory, as shown:

```
OpenEdge-install-directory/java/ext/
    common.jar
    commonj.sdo.jar
   .ecore.jar
   .ecore.change.jar
   .ecore.sdo.jar
   .ecore.resources.jar
   .ecore.xmi.jar
```

3. Update the CLASSPATH setting for the Java environment that will run the client application to include the following:
   a. The client application classes.
   b. The directory containing the proxy (.class files) or the .zip or .jar file from Step 2. For class files, this is the directory above the package hierarchy.
   c. The Open Client Runtime package and Eclipse jar files you copied in Step 2.

Note: The mechanism to update the CLASSPATH setting depends on the Java environment you are using. For instructions on updating CLASSPATH, see the documentation for your Java environment.
4. If you use HTTPS (SSL), copy your digital certificates to the location required by your application. The Open Client Toolkit includes the certificate management tool (procertm utility), which provides a way to import, export and remove certificates to and from .jar and .zip files. See Appendix B, “Java Open Client Certificate Management Utility.”

5. Run your Java Open Client application as designed.
Sample Java client applications

Sample Java client applications are located in a directory in the Documentation and Samples directory (doc_samples) of the OpenEdge product DVD. For more information on accessing this directory, see the section on example procedures in the “Preface”.

You can also find the samples by going to the OpenEdge Product Documentation Overview page on PSDN: http://communities.progress.com/pcom/docs/DOC-16074.

This samples directory contains separate subdirectories for different sets of samples and support files. Each sample subdirectory contains a readme.txt file with instructions on building and running the sample.

This book references some of these samples, but also contains many other code samples and examples.
Using the Open Client architecture, Java Open Clients access AppServer™ business logic through proxy code. The client programmer writes a Java client application and uses a proxy generated by ProxyGen to access AppServer functionality through methods on the generated proxy objects.

The client proxy code uses the Java Open Client Runtime to communicate with the AppServer. The Open Client Runtime converts parameters and return values between ABL and Java data types as needed.

HTML documentation on the classes and interfaces provided by Open Client Runtime is in $OpenEdge-install-directory/java/doc/.$

Java proxies generated by ProxyGen contain one or more classes. The Java proxies can be generated in Windows or on UNIX platforms. For information on the Open Client architecture, see $OpenEdge Development: Open Client Introduction and Programming.$

This chapter describes the Java proxy classes and methods generated by ProxyGen, as detailed in the following sections:

- Proxy objects
- Proxy methods
- Sample proxy
Proxy objects

For a Java Open Client, ProxyGen generates a set of Java classes for each proxy object, as shown in the following table:

<table>
<thead>
<tr>
<th>For each . . .</th>
<th>ProxyGen generates . . .</th>
<th>Example</th>
</tr>
</thead>
</table>
| AppObject     | Two Java classes         | public class Account
                |         | public class AccountImpl |
| SubAppObject  | Two Java classes         | public class Tax
                |         | public class TaxImpl |
| ProcObject    | Two Java classes         | public class AccountInfo
                |         | public class AccountInfoImpl |

In each case, the source for the public (delegating) class also is available. In the examples above, these would be Account.java, Tax.java, and AccountInfo.java.

ProxyGen generates the proxy classes into a package, if you specify one on the Java tab of the Generate dialog box in ProxyGen. You also can access remote OpenEdge SmartDataObjects from a Java client with or without a ProxyGen-generated proxy. For more information, see Chapter 9, “Using SmartDataObjects from Java Clients.”

ProxyGen also generates AppObject.log, an activity log file with status and error information (for example, Account.log).

All these files are placed in the output directory specified on the General tab of the Generate dialog box in ProxyGen.

Package

All proxy classes may be created within a package specified on the Java tab of the Generate dialog box in ProxyGen. The class files are placed in a directory structure defined by the package.
Proxy methods

This section describes the following proxy methods:

- Connection methods.
- Remote ABL methods.
- Class factory methods.
- Common methods.

It also discusses running methods on session-free AppObjects.

Connection methods

The constructor of the AppObject is used to establish a connection to an AppServer. For details, see Chapter 3, “Connecting to an AppServer”.

Remote ABL methods

ProxyGen maps each ABL non-persistent procedure, internal procedure, and user-defined function exposed on the AppServer to a remote ABL method. These methods are part of an AppObject, SubAppObject, or ProcObject. ProxyGen generates method names using automatic conversion and conventions. For more information on proxy generation, see OpenEdge Development: Open Client Introduction and Programming.

Two sample non-persistent procedures follow:

```plaintext
AddAccount.p:
DEFINE INPUT PARAMETER accountNum AS INTEGER.
DEFINE INPUT PARAMETER name AS CHARACTER.

RemoveAccount.p:
DEFINE INPUT PARAMETER accountNum AS INTEGER.
DEFINE OUTPUT PARAMETER name AS CHARACTER.
```

ProxyGen generates the following Java proxy methods:

```java
public void AddAccount(int accountNum, String name)
public void RemoveAccount(int accountNum, StringHolder name)
```

Passing parameters

ProxyGen maps ABL data types to equivalent data types in Java for ABL INPUT, OUTPUT, and INPUT-OUTPUT parameters. For details, see Chapter 4, “Passing Parameters.”
Handling return values

When using ProxyGen, you can optionally specify whether the ABL RETURN-VALUE should be added to the Java proxy method, for each non-persistent procedure and internal procedure (user-defined functions always return a value.) If this is not specified, the method returns void. If specified for the non-persistent procedures above, ProxyGen generates the following Java proxy methods:

```java
public String AddAccount(int accountNum, String name)
public String RemoveAccount(int accountNum, StringHolder name)
```

Java Open Client supports array return values from user-defined functions, except for LONGCHAR and MEMPTR data types.

Also, if you did not specify to return the ABL RETURN-VALUE, the client can access the current value of the ABL RETURN-VALUE function by calling the _getProcReturnString() common method. For more information, see the “Common methods” section on page 2–5.

Class factory methods

ProxyGen generates two class factory methods.

SubAppObject

The following method allows AppObjects to create SubAppObjects that share an AppServer connection with an existing AppObject:

**Syntax**

```java
public SubAppObject createAO_SubAppObject()
```

For example, a SubAppObject named Tax defined in ProxyGen generates this Java method:

```java
public Tax createAO_Tax()
```

ProcObject

The following method allows AppObjects or SubAppObjects to create ProcObjects that share an AppServer connection with an existing AppObject or SubAppObject:

**Syntax**

```java
public ProcObject createPO_ProcObject()
```

For example, a persistent procedure AccountInfo.p added in ProxyGen generates this Java method:

```java
public AccountInfo createPO_AccountInfo()
```
Proxy methods

SDOResultSet

ProxyGen generates the following class factory methods for AppObjects and SubAppObjects to create SDOResultSet objects:

Syntax

```java
public SDOResultSet _createSDOResultSet(String procName)
public SDOResultSet _createSDOResultSet(String procName, String whereClause, String sortBy)
public SDOResultSet _createSDOResultSet(String procName, String whereClause, String sortBy, SDOPameters params)
```

These methods create an SDOResultSet object and run the SmartDataObject specified by `procName`.

Common methods

The Open Client interface provides common methods which provide information about the current state of Java proxy AppObjects, SubAppObjects, and ProcObjects, with respect to their connection to the application service. The section lists these methods and describes the information they provide. This information might differ based on whether the application service is run session-managed or session-free. (For an overview of session models, see OpenEdge Development: Open Client Introduction and Programming.)

Cancel all requests

The following method raises a STOP condition in the context of each outstanding request on the AppServer:

Syntax

```java
public void _cancelAllRequests()
```

The `_cancelAllRequests()` method is most useful for multi-threaded clients.

This method throws a `com.progress.open4gl.Open4GLEException`.

AppObjects, SubAppObjects, and ProcObjects have a `_cancelAllRequests()` method that a client can call to do the following:

- Raise the STOP condition on a request that was initiated by any object in the proxy and is running (normally during a long execution)
- Normal STOP condition processing applies, including the ability to trap the STOP condition
- Close any open ResultSets

When a client invokes a proxy method, the result is a request to the AppServer. The request itself can be in one of three states. Also, note that when the method completes and returns control to the application, the request still can be active if there is an open output ResultSet (that is, the connection is in a STREAMING state).
The three request states are:

- **QUEUED** — The connection is busy with a request from another thread. The current request is queued until the connection becomes available.

- **RUNNING** — The current request is being run on the connection, and the connection is in the RUNNING state.

- **STREAMING** — The current request is transmitting records over the connection, and the connection is in the STREAMING state.

The `_cancelAllRequests()` method operates on a request according to its state, as follows:

- **QUEUED** — Cancels the request and throws an exception if the STOP condition is not handled.

- **RUNNING** — Forwards a STOP request to the ABL interpreter on the AppServer, which handles it according to normal ABL STOP condition rules.

- **STREAMING** — Initiates the close of any currently open output ResultSets, but does not call the close() method. You must then execute the close() method on all affected output ResultSets, to complete the closure. For large ResultSets, this is more efficient than using the close() method alone. For more information on the close() method, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters”.

If there are no requests in the above states when you run `_cancelAllRequests()`, the method has no effect.

**Connection ID**

The `_getConnectionId()` method is most often used to identify entries in AppServer log files.

For session-managed applications, the following method returns a string containing a unique identifier for this connection.

For session-free applications, the following method returns a string containing a unique identifier for the connection most recently used by the current thread:

**Syntax**

```java
public String _getConnectionId()
```

The `_getConnectionId()` method throws a `com.progress.open4gl.Open4GLErrorException` if the Application Service is not connected.

**Streaming**

For session-managed applications, the `_isStreaming()` method determines if the method most recently run on any thread is currently streaming an output result set.

For session-free applications, the `_isStreaming()` method determines if the method most recently run on the current thread is currently streaming an output result set. This state information cannot be retrieved by another thread.
The following method returns true if there is an open OUTPUT TABLE or TABLE-HANDLE for this connection:

**Syntax**

```java
public boolean _isStreaming()
```

The _isStreaming() method throws a com.progress.open4gl.Open4GLException. For more information on streaming, see the information on output ResultSet parameters in Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

**Procedure return string**

For session-managed applications, the _getProcReturnString() method retrieves the return value of the method most recently run on any thread.

For session-free applications, the _getProcReturnString() method retrieves the return value of the method most recently run on the current thread. The return value cannot be retrieved by another thread.

The following method returns the current value of the AppServer RETURN-VALUE function:

**Syntax**

```java
public String _getProcReturnString()
```

For more information on handling return values, see the “Handling return values” section on page 2–4.

**Release**

The _release() method makes this object unavailable for further use. For example:

**Syntax**

```java
public void _release()
```

If you execute _release() on the last available proxy object that shared a particular connection, the client disconnects the AppServer. If you execute _release() on a ProcObject, the associated persistent procedure on the AppServer is deleted.

**Request ID**

For session-managed applications, the _getRequestId() method returns a unique string identifying the request most recently run on any thread.

For session-free applications, the _getRequestId() method returns a unique string identifying the request most recently run on the current thread. The Request ID cannot be retrieved by another thread.

The following method throws a com.progress.open4gl.Open4GLException if the session is not available:

**Syntax**

```java
public String _getRequestId()
```
SSL subject name

The _getSSLSubjectName() method provides the SSL server's subject name that is obtained from its validated digital certificate. The following method throws a com.progress.open4gl.Open4GLException if the session is not available:

Syntax

```
public String _getSSLSubjectName()
```
Running methods on session-free AppObjects

When a session-free AppObject is instantiated, a pool of connections to the application service is established, as specified by the run-time properties provided. Once instantiated, methods on that AppObject can be called in accordance with the standard Open Client programming model. In the session-free model, each external method call transparently runs a request using a separate connection from the connection pool. (An external method is one that corresponds to an external ABL procedure on the AppServer.) As such, a multi-threaded application may run remote application service requests concurrently. As each request completes, the connection is released back to the pool and is available for another request.

Connections for persistent procedures are handled slightly differently than for non-persistent procedures. When a persistent procedure is instantiated, a connection is reserved from the connection pool. All subsequent internal procedures and user-defined functions run on that persistent procedure use that same connection. The connection is released back to the connection pool only when the persistent procedure is released by calling the _release() method on the ProcObject.

Session-free threading model

Open Client applications that use the session-free model are presumed to employ a threading model such that each request executes on a separate thread. This becomes particularly important in the following areas:

- When the application constrains the size of the connection pool, requests can become blocked if connections are unavailable. This is accomplished by the Open Client Runtime blocking the thread on which the request is run.

- Certain request-specific methods must be executed on the same thread on which the request was run. For more information, see the “Common methods” section on page 2–5.

- A thread can only execute one request at a time. A thread that runs a request that returns output result sets may not run another request until all the result set data has been retrieved.

Sharing of the connection pool between session-free AppObjects

A session-free AppObject is instantiated using an explicitly provided Connection object. The Connection object instance establishes a reference to the AppObject’s connection pool. If the same Connection object instance is then used to instantiate other session-free AppObject instances, the subsequent AppObject instances will share the connection pool referenced by the Connection object. SubAppObjects and ProcObjects always share the connection pool of their associated AppObject.

The Connection object's reference to the connection pool will be maintained until the releaseConnection() method is called on the Connection object. This reference may affect the life cycle of the connection pool. That is, the Connection object may be used to sustain the existence of the connection pool beyond the lifetime of the AppObject (and its associated SubAppObjects and ProcObjects). It is your responsibility to call releaseConnection() on the Connection object to remove the reference to the connection pool.
It should be noted that a single `Connection` object can be used to instantiate instances of different `AppObject` classes, provided that the `Connection` object refers to an `AppServer` that provides the necessary application services. This is possible since a single `AppServer` can serve multiple application services. However, no validation of this is provided.

The `Connection` object does not maintain a reference to a session-managed `AppObject`, regardless of how it is constructed. If a session-managed `AppObject` is constructed using a `Connection` object that contains a reference to a connection pool, an exception is thrown. In other words, `Connection` objects cannot be shared between session-free and session-managed `AppObjects`. 
Sample proxy

This section describes examples of methods that are part of an AppObject, a SubAppObject, and a ProcObject.

**Note:** These samples are not available on the OpenEdge product DVD or PSDN.

**Sample AppObject**

The following methods comprise the definition of a sample Java AppObject, Account, where each method appears as it is displayed in Java:

- The AppObject constructors that connect to an Application Service. For example:

  ```java
  public Account(com.progress.open4gl.javaproxy.Connection connectObj)
  throws Open4GLException, IOException, ConnectException,
  SystemErrorException

  public Account(String urlString, String userId, String password,
  String Info)
  throws Open4GLException, IOException, ConnectException,
  SystemErrorException

  public Account(String userId, String password, String Info)
  throws Open4GLException, IOException, ConnectException,
  SystemErrorException

  public Account() throws Open4GLException, IOException, ConnectException,
  SystemErrorException
  ```

- A SubAppObject class factory method that creates a SubAppObject called Tax. For example:

  ```java
  public Tax createAO_Tax() throws Open4GLException, RunTime4GLException,
  SystemErrorException
  ```

- A ProcObject class factory method that creates a ProcObject called AccountInfo and runs and instantiates the persistent procedure AccountInfo.p on the AppServer. For example:

  ```java
  public AccountInfo createPO_AccountInfo(int accountNum)
  throws Open4GLException, RunTime4GLException, SystemErrorException
  ```
• Built-in class factory methods that run and instantiate a SmartDataObject on the AppServer. For example:

```java
public SDOResultSet _createSDOResultSet(String procName)
    throws Open4GLException, ProSQLException
public SDOResultSet _createSDOResultSet(String procName,
    String whereClause, String sortBy)
    throws Open4GLException, ProSQLException
public SDOResultSet _createSDOResultSet(String procName,
    String whereClause, String sortBy, SDOPameters params)
    throws Open4GLException, ProSQLException
```

• Remote ABL methods to run the non-persistent procedures Add (defined by Add.p) and Remove (defined by Remove.p) on the AppServer. For example:

```java
public void Add(int accountNum, String name)
public void Remove(int accountNum)
```

• Common methods in all Java proxy objects. For example:

```java
public void _cancelAllRequests() throws Open4GLException
public String _getConnectionId() throws Open4GLException
public String _getProcReturnString()
public boolean _isStreaming() throws Open4GLException
public String _getRequestId() throws Open4GLException
public String _getSSLSubjectName() throws Open4GLException
public void _release()
```

**Sample SubAppObject**

The following methods comprise the definition of a sample Java SubAppObject, Tax:

• A ProcObject class factory method that runs and instantiates the persistent procedure TaxInfo.p on the AppServer. For example:

```java
public TaxInfo createPO_TaxInfo(int accountNum)
    throws Open4GLException, RunTime4GLException,
    SystemErrorException
```

• Built-in class factory methods that run and instantiate a SmartDataObject on the AppServer, as shown on the sample AppObject in the “Sample AppObject” section on page 2–11.
• A remote ABL method to run the non-persistent procedure SetStatus (defined by SetStatus.p) on the AppServer. For example:

```java
public void SetStatus(int status)
```

• Common methods in all Java proxy objects, as shown in the “Sample AppObject” section on page 2–11.

**Sample ProcObject**

The following methods comprise the definition of a sample Java ProcObject, AccountInfo:

• Remote ABL methods to run the internal procedures getPaymentsInfo, setDirectDeposit, and getDirectDeposit on the AppServer, found in the persistent procedure AccountInfo.p. For example:

```java
public void getPaymentsInfo(Date fromDate, ResultSet payeeList, int paymentsNum, ResultSetHolder paymentsInfo)

public void setDirectDeposit(ResultSet ddData)

public void getDirectDeposit(ResultSetHolder ddData)
```

• Common methods in all Java proxy objects, as shown in the “Sample AppObject” section on page 2–11.
Connecting to an AppServer

This chapter covers all the mechanisms and procedures you will need to connect to an AppServer. This chapter contains the following sections:

- Connection class
- Establishing the connection
- Connection states
Connecting to an AppServer

Connection class

OpenEdge provides a Connection class, com.progress.open4gl.javaproxy.Connection. This Connection object provides a means to store AppServer connection information, which can be passed to the AppObject constructor when connecting to the AppServer.

The Connection object has two constructors, as shown in the syntax boxes that follows.

The following constructor specifies the AppServer connection information (url), user ID, password, and information required by the AppServer application at connection time:

Syntax

```java
public Connection(String url, String userid, String password,
               String AppServer-info)
```

For more information on the AppServer URL connection parameter format and default connection information, see the sections on connecting to an AppServer using a URL in OpenEdge Application Server: Developing AppServer Applications.

The following constructor specifies the user ID, password, and information required by the AppServer application at connection time:

Syntax

```java
public Connection(String userid, String password, String appserver-info)
```

The constructor above uses default AppServer connection information. This defaults the URL to AppServer://localhost:5162/appService, where appService is the AppService setting specified on the General tab of the Generate dialog box in ProxyGen.

Before you establish a connection, you can set properties for the Connection object. See Chapter 7, “Accessing Proxy Properties,” for more information on setting and updating Connection object properties.
Establishing the connection

To establish a connection to an AppServer, you must instantiate an AppObject. You may create an AppObject using one of the four available constructors, described in this section.

The following constructor establishes an AppServer connection, with the AppServer information specified in the Connection object (connectObj):

```java
public AppObject(com.progress.open4gl.javaproxy.Connection connectObj)
```

Where `AppObject` is the name of the AppObject as defined in ProxyGen.

A `Connection` object can be instantiated with one of the constructors documented in the “Connection class” section on page 3–2.

The following constructor establishes an AppServer connection, with the specified AppServer connection information (url), user ID, password, and information required by the AppServer application at connection time:

```java
public AppObject(String url, String userid, String password,
    String appserver-info)
```

Where `AppObject` is the name of the AppObject as defined in ProxyGen.

For more information on the AppServer URL connection parameter format and default connection information, see the sections on connecting to an AppServer using a URL in *OpenEdge Application Server: Developing AppServer Applications*.

The following constructor establishes an AppServer connection, with the specified user ID, password, and information required by the AppServer application at connection time:

```java
public AppObject(String userid, String password, String appserver-info)
```

Where `AppObject` is the name of the AppObject as defined in ProxyGen.

The above constructor uses default AppServer connection information. This defaults the URL to `AppServer://localhost:5162/appService`, where `appService` is the `AppService` setting specified on the General tab of the Generate dialog box in ProxyGen.

The following constructor establishes an AppServer connection, with no specified AppServer information:

```java
public AppObject()
```

Where `AppObject` is the name of the AppObject as defined in ProxyGen.

The constructor above uses default AppServer connection information. This defaults the URL to `AppServer://localhost:5162/appService`, where `appService` is the `AppService` setting specified on the General tab of the Generate dialog box in ProxyGen.

Also see the information on connecting to an AppServer in *OpenEdge Development: Open Client Introduction and Programming*. 

---

3–3
Connecting to an AppServer

**Returning a user defined string to the client from the AppServer connection procedure**

When you establish a connection to the AppServer with the Java Open Client, you instantiate an AppObject using one of the four constructors provided by ProxyGen. If the connection to the AppServer fails, the AppObject constructor throws a standard exception. This exception can also contain a user defined string, if you have set up your AppServer to do so.

The Connection object communicates with a remote ABL procedure stored on the AppServer known as the connection procedure. If that procedure contains the ABL RETURN string statement, then that string will be contained in the connection failure exception.

If the connection is successful, and the connection procedure has an ABL RETURN string statement, you can access the string using the AppObject method _getProcReturnString(). If the connect procedure does not return a value, then the _getProcReturnString() method returns null.

The following sample Java Open Client code illustrates this functionality:

```java
try {
    //Create Customer AppObject to connect
    Customer appObj = new Customer(ConnectObj);
    System.out.println((String) appObj._getProcReturnString());
} catch (com.progress.open4gl.Open4GLException ex) {
    // if there is a application defined return string, then display it
    if (((String)(ex._getProcReturnString()) != null)) {
        System.out.println((String)(ex._getProcReturnString()));
    } else {
        // display Progress defined error message
        System.out.println(ex.toString());
        // or display new generic message
        System.out.println("Connection failed");
    }
} finally {
    if (appObj != null) {
        appObj._release();
    }
}
```

If the connection is successful and the connect procedure returns a string, then the string value is displayed by the first message box. If the connect procedure failed and returns a string, then the string value is displayed by the second message box.

**Note:** This feature is only available to AppServer applications using the managed-session model, since it is this type of application that uses a connection procedure.
Establishing the connection

Supported AppServer modes

For Java Open Clients, the supported AppServer operating modes are the session-managed modes of state-aware, state-reset, and stateless, and the session-free mode, state-free. For more information, see the information on session models in the chapter on generating proxies and Web service definitions in OpenEdge Development: Open Client Introduction and Programming.

To access a session-free AppServer, you must set the proxy property PROGRESS.Session.sessionModel to 1. For more information, see Chapter 7, “Accessing Proxy Properties.”

Releasing a session-free Connection object

For session-free applications, you must explicitly release the connections held by a Connection object when they are no longer needed. Use the following method on the Connection object for this purpose:

public void releaseConnection()

Note: This method is not necessary for and has no effect in session-managed applications.
Connection states

A connection between a Java Open Client and AppServer can be in one of the following three states that define the status of the current AppServer method call:

- **RUNNING** — An AppServer method is executing. No other AppServer methods can be executed when the connection is in the **RUNNING** state.

- **STREAMING** — There is at least one open output ResultSet. AppServer methods cannot be executed when the connection is in the **STREAMING** state. To determine whether a connection is in the **STREAMING** state, invoke the `isStreaming()` method on any proxy object sharing the connection.

- **IDLE** — The connection is not in a **RUNNING** or **STREAMING** state.
Passing Parameters

This chapter provides details about data type mappings in various parameter passing scenarios. This chapter contains the following sections:

- ABL data type mappings
- Passing temp-tables and ProDataSets
ABL data type mappings

INPUT ABL parameters are mapped directly to Java data types and classes. OUTPUT and INPUT–OUTPUT parameters are mapped to holder classes in the com.progress.open4gl package. The contents of the holder class are the same Java data types and classes as the INPUT parameters, with the exception of temp-table (TABLE and TABLE–HANDLE) parameters mapped to an SQL ResultSet. For these temp-table parameters, the content of the holder class is not the same data type as the INPUT parameters.

INPUT parameters

Table 4–1 shows the ABL-to-Java data type mapping for INPUT parameters.

Table 4–1: ABL-to-Java data type mapping for INPUT parameters

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Java proxy data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>COM–HANDLE</td>
<td>com.progress.open4gl.COMHandle</td>
</tr>
<tr>
<td>DATASET</td>
<td>com.progress.open4gl.ProDataGraph</td>
</tr>
<tr>
<td>DATASET–HANDLE</td>
<td>com.progress.open4gl.ProDataGraph</td>
</tr>
<tr>
<td>DATE</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DATETIME</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DATETIME–TZ</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>HANDLE</td>
<td>com.progress.open4gl.Handle</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>com.progress.open4gl.Memptr</td>
</tr>
<tr>
<td>RAW</td>
<td>byte[]</td>
</tr>
<tr>
<td>RECID</td>
<td>long</td>
</tr>
<tr>
<td>ROWID</td>
<td>com.progress.open4gl.Rowid</td>
</tr>
<tr>
<td>TABLE</td>
<td>com.progress.open4gl.ProDataGraph</td>
</tr>
<tr>
<td>TABLE–HANDLE</td>
<td>com.progress.open4gl.ProDataGraph</td>
</tr>
<tr>
<td>TABLE–HANDLE</td>
<td>com.progress.open4gl.ProDataGraph</td>
</tr>
<tr>
<td>TABLE–HANDLE</td>
<td>java.sql.ResultSet</td>
</tr>
<tr>
<td>WIDGET–HANDLE</td>
<td>com.progress.open4gl.Handle</td>
</tr>
</tbody>
</table>

1. ProDataSet (DATASET and DATASET–HANDLE) and temp-table (TABLE and TABLE–HANDLE) parameters require special handling, including the mapping of temp-table fields. For more information on ProDataSet and temp-table parameters, see the “Passing temp-tables and ProDataSets” section on page 4–8.
The COM-HANDLE, HANDLE, RECID, ROWID, and WIDGET-HANDLE ABL data types are not meaningful outside the OpenEdge environment. Their use is restricted to obtaining the value from an ABL procedure and passing it back to another ABL procedure.

**Note:** The ABL BUFFER parameter type is not supported in Java Open Clients.

### Date and time conversions

The ABL DATE, DATETIME, and DATETIME-TZ data types are all mapped to the java.util.GregorianCalendar class. This class supports all the necessary functionality for these data types (it supports date, time, and time zone data). The Java Open Client processes the data differently based on the ABL data type, as shown in Table 4–2.

**Table 4–2: GeorgianCalendar object processing**

<table>
<thead>
<tr>
<th>Progress data type</th>
<th>Process</th>
</tr>
</thead>
</table>
| DATE               | **Input:** Time and time zone information is ignored  
|                    | **Output:** Client time zone is used. Time is set to 0 (midnight) |
| DATETIME           | **Input:** Time zone information is ignored  
|                    | **Output:** Client time zone is used |
| DATETIME-TZ        | **Input and Output:** Date, time, and time zone is preserved |

### Arrays as parameters

ABL allows arrays (called EXTENTs in ABL) to be passed as parameters. In the Java client, they are mapped to arrays where the base type is defined in Table 4–1. For example, an ABL CHARACTER array is mapped to java.lang.String[]. An ABL INT64 array is mapped to the intrinsic type long[].

The size of an array is not part of its definition in Java, so the EXTENT value is not reflected in the Java proxy’s parameter definition. If the extent of the data passed by the client does not match the extent of the parameter declaration of the ABL procedure, the proxy returns an error to the client at runtime.

**Note:** Arrays defined with EXTENT 0 are treated as scalars (consistent with ABL). The proxy will contain a scalar for the parameter, and not an array.

ProxyGen indicates the EXTENT of the parameters (if any) in the comments of the proxy.
Unknown value (?) as a parameter

The Unknown value (?) is mapped to a Java object with a null value. The int, boolean, and long data types, however, are intrinsic types, not Java classes, so the null value does not belong to the set of values they support. If a proxy is defined to support the ABL Unknown value (?) in ProxyGen, parameters that otherwise would be intrinsic data types are generated as Java classes. For example, ProxyGen generates an Integer parameter instead of an int, a Boolean parameter instead of a boolean, and a Long parameter instead of long.

The following is an example of a proxy method signature that does not support the ABL Unknown value (?) for the first and third parameters:

```
foo(int, Date, boolean)
```

The following is an example of a proxy method that allows all the parameter values to be Unknown value (?):

```
foo(Integer, Date, Boolean)
```

Arrays and Unknown value (?)

Since arrays are Java objects, they can be assigned the null value.

However, int[], boolean[], and long[] are arrays of intrinsic types, so the individual elements of the array cannot be null. The int[], boolean[], or long[] array as a whole can be null, which means that every element of the array on the AppServer will be UNKNOWN. If you want to assign Unknown value (?) to individual array elements, use the Allow Unknown check box in ProxyGen, and the generated proxy will contain Integer[], Boolean[] or Long[] for the parameter instead of int[], boolean[], or long[], respectively.

INPUT-OUTPUT and OUTPUT parameters

Java supports no direct mechanism to return a value from a method using parameters. An application can provide holder classes for parameters that contain the value you want to return.

Holder classes

For each Java data type that maps to an ABL data type, there is a Holder class. These Holder classes belong to the com.progress.open4gl package and extend the com.progress.open4gl.Holder class, as shown in Example 4–1.

Example 4–1: OUTPUT and INPUT-OUTPUT holder class

```java
// Holder Definition
public class Holder {
    public Holder();
    public Holder(Object value);
    public void setValue(Object value);
    public Object getValue();
    public boolean isNull();
}
```
This class has two constructors, one that does and one that does not set a value for the parameter. For an OUTPUT parameter, you do not need to set a value in the holder object. For an INPUT-OUTPUT parameter, you typically do need to set a value to pass as input in the holder object. You can also set the value using the `setValue()` method after the holder is created.

For an INPUT-OUTPUT parameter, you can set an input value to the ABL Unknown value (?) by calling `setValue(null)` on the holder object. For INPUT-OUTPUT and OUTPUT parameters, the application can find out whether an unknown output value is returned by calling the `isNull()` method or testing for a null return value from the `getValue()` method on the holder object.

For each parameter data type extension, the holder class and method names change as follows:

**Syntax**

```java
public class DataTypeNameHolder {
    public DataTypeNameHolder();
    public DataTypeNameHolder(DataType value);
    public void setDataTypeNameValue(DataType value);
    public Object getDataTypeNameValue();
}
```

`DataTypeName` is an initial upper-case name that closely matches the short intrinsic data type or class name for the value, and `DataType` is the exact primitive data type or full Java class name (with no changes in letter case).

Table 4–3 shows the ABL-to-Java mapping of INPUT-OUTPUT and OUTPUT parameters to specific holder classes.

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Java proxy data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>com.progress.open4gl.StringHolder</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>com.progress.open4gl.COMHandleHolder</td>
</tr>
<tr>
<td>DATASET</td>
<td>com.progress.open4gl.ProDataGraphHolder</td>
</tr>
<tr>
<td>DATASET-HANDLE</td>
<td>com.progress.open4gl.ProDataGraphHolder</td>
</tr>
<tr>
<td>DATE</td>
<td>com.progress.open4gl.DateHolder</td>
</tr>
<tr>
<td>DATETIME</td>
<td>com.progress.open4gl.BigDecimalHolder</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>com.progress.open4gl.BigDecimalHolder</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>com.progress.open4gl.BigDecimalHolder</td>
</tr>
<tr>
<td>INT64</td>
<td>com.progress.open4gl.LongHolder</td>
</tr>
<tr>
<td>INTEGER</td>
<td>com.progress.open4gl.IntHolder</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>com.progress.open4gl.BooleanHolder</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>com.progress.open4gl.StringHolder</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>com.progress.open4gl.MemptrHolder</td>
</tr>
<tr>
<td>RAW</td>
<td>com.progress.open4gl.ByteArrayHolder</td>
</tr>
<tr>
<td>RECID</td>
<td>com.progress.open4gl.LongHolder</td>
</tr>
</tbody>
</table>
Table 4–4 shows the ABL array-to-Java mapping of INPUT-OUTPUT and OUTPUT array parameters to specific holder classes.

Table 4–4: ABL with array-to-Java data-type mapping for INPUT-OUTPUT and OUTPUT array parameters

<table>
<thead>
<tr>
<th>ABL array option</th>
<th>Java proxy data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>com.progress.open4gl.StringArrayHolder</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>com.progress.open4gl.COMHandleArrayHolder</td>
</tr>
<tr>
<td>DATE</td>
<td>com.progress.open4gl.DateArrayHolder</td>
</tr>
<tr>
<td>DATETIME</td>
<td>com.progress.open4gl.DateArrayHolder</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>com.progress.open4gl.DateArrayHolder</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>com.progress.open4gl.BigDecimalArrayHolder</td>
</tr>
<tr>
<td>INT64</td>
<td>com.progress.open4gl.LongArrayHolder</td>
</tr>
<tr>
<td>INTEGER</td>
<td>com.progress.open4gl.IntArrayHolder</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>com.progress.open4gl.BooleanArrayHolder</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>com.progress.open4gl.StringArrayHolder</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>com.progress.open4gl.MemptrArrayHolder</td>
</tr>
<tr>
<td>RAW</td>
<td>com.progress.open4gl.ByteArrayArrayHolder</td>
</tr>
<tr>
<td>RECID</td>
<td>com.progress.open4gl.LongArrayHolder</td>
</tr>
<tr>
<td>ROWID</td>
<td>com.progress.open4gl.RowidArrayHolder</td>
</tr>
<tr>
<td>WIDGET–HANDLE</td>
<td>com.progress.open4gl.HandleArrayHolder</td>
</tr>
</tbody>
</table>

Table 4–4 shows the ABL array-to-Java mapping of INPUT-OUTPUT and OUTPUT array parameters to specific holder classes.

Table 4–4 shows the ABL array-to-Java mapping of INPUT-OUTPUT and OUTPUT array parameters to specific holder classes.

Table 4–3: ABL-to-Java data type mapping for INPUT-OUTPUT and OUTPUT parameters (2 of 2)

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Java proxy data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWID</td>
<td>com.progress.open4gl.RowidHolder</td>
</tr>
<tr>
<td>TABLE</td>
<td>com.progress.open4gl.ProDataGraphHolder</td>
</tr>
<tr>
<td>TABLE-HANDLE</td>
<td>com.progress.open4gl.ProDataGraphHolder</td>
</tr>
<tr>
<td>TABLE</td>
<td>com.progress.open4gl.ResultSetHolder</td>
</tr>
<tr>
<td>TABLE-HANDLE</td>
<td>com.progress.open4gl.ResultSetHolder</td>
</tr>
<tr>
<td>WIDGET–HANDLE</td>
<td>com.progress.open4gl.HandleHolder</td>
</tr>
</tbody>
</table>

1. ProDataSet (DATASET and DATASET–HANDLE) and temp-table (TABLE and TABLE–HANDLE) parameters require special handling, including the mapping of temp-table fields. For more information on ProDataSet and temp-table parameters, see the “Passing temp-tables and ProDataSets” section on page 4–8.
**Holder class definitions**

The holder class definitions for each data type follow the form shown in Example 4–2.

### Example 4–2: Holder class definitions

```java
public class IntHolder extends Holder {
    public IntHolder();
    public IntHolder(int value);
    public void setIntValue(int value);
    public int getIntValue();
}
/* Note the extra constructor on this one. */
public class BigDecimalHolder extends Holder {
    public BigDecimalHolder();
    public BigDecimalHolder(java.math.BigDecimal value);
    public BigDecimalHolder(double);
    public void setBigDecimalValue(BigDecimal value);
    public java.math.BigDecimal getBigDecimalValue();
}
public class StringHolder extends Holder {
    public StringHolder();
    public StringHolder(java.lang.String value);
    public void setStringValue(java.lang.String value);
    public java.lang.String getStringValue();
}
public class BooleanHolder extends Holder {
    public BooleanHolder();
    public BooleanHolder (boolean value);
    public void setBooleanValue(boolean value);
    public boolean getBooleanValue();
}
public class LongArrayHolder extends Holder {
    public LongArrayHolder();
    public LongArrayHolder(long[ ] value);
    public LongArrayHolder(Long[ ] value);
    public void setLongArrayValue(long[ ] value);
    public void setLongArrayValue(Long[ ] value);
    public Long[ ] getLongArrayValue();
}
/* Holder classes for other data types follow similar forms */
```
Passing Parameters

Passing temp-tables and ProDataSets

OpenEdge allows you to pass ABL ProDataSets and temp-tables as Java Open Client proxy method parameters. A temp-table is a complex data structure that maintains the equivalent of a relational database table in memory. That is, a temp-table contains a list of records (or rows), each of which contains an identical list of fields (or columns) that can include arrays. The records of a temp-table can be populated programmatically from any data source available in ABL.

A ProDataSet is a complex data structure based on temp-tables that maintains the equivalent of a relational database subset in memory. A ProDataSet allows multiple temp-tables to be treated as a unit, including features to manage them not available to individual temp-tables. These features include the ability to define and manage parent-child relationships (data-relations) among the tables in the ProDataSet and to track changes to the data. Temp-tables of ProDataSets also can be populated automatically from supported data sources or (like individual temp-tables) programmatically from all data sources available in ABL.

For more information on:

- Temp-tables, see OpenEdge Getting Started: ABL Essentials
- ProDataSets, see OpenEdge Development: ProDataSets

Static and dynamic temp-tables and ProDataSets

In ABL, a static temp-table or ProDataSet is one whose schema (meta data) is defined and built at compile time. A dynamic temp-table or ProDataSet is one whose schema is defined and built entirely at run time. Depending on how you pass static and dynamic temp-tables and ProDataSets, your Java client coding requirements might differ.

ABL mechanisms for passing temp-tables and ProDataSets

In ABL, you pass a static temp-table using a TABLE parameter and pass a static ProDataSet using a DATASET parameter. In the Java Open Client, you must ensure that the meta data for the proxy method parameter maps exactly to the schema of the corresponding static temp-table or ProDataSet passed in ABL.

In ABL, you pass a dynamic temp-table using a TABLE-HANDLE parameter and pass a dynamic ProDataSet using a DATASET-HANDLE parameter. In the Java Open Client or in ABL application service, there is no requirement to know the meta data or schema of the complex data that is passed. However, you must otherwise introspect the parameter meta data in the Java client and the parameter schema in the application service in order to access the data at either end.

Note: Of course, even for a dynamic data structure, if you know the meta data or schema of the corresponding parameter at the opposite end, the programming is much less complex.

Depending on the mechanisms you use to pass a temp-table or ProDataSet in ABL and the corresponding parameter in the Java Open Client, OpenEdge provides different options to optimize and manage the data transfer.
Java mechanisms for passing temp-tables and ProDataSets

In the Java Open Client, the default mechanism for passing either a single temp-table or a ProDataSet parameter (static or dynamic) is the OpenEdge ProDataGraph. The ProDataGraph class is an implementation of the DataGraph interface defined by Java Service Data Objects (Java SDO), a standard for managing complex data jointly developed by IBM and BEA. An alternative mechanism for passing temp-tables (but not ProDataSets) is the SQL ResultSet interface, supported by the Java Database Connectivity (JDBC) standard.

ProDataGraph mechanism

The ProDataGraph provides a cached and scrollable model for transferring complex data between the AppServer and the Java Open Client. This model allows you to access any row of any temp-table from beginning to end and in any direction. You can use the same model and programming techniques for accessing an individual temp-table parameter or for accessing temp-tables as part of a ProDataSet parameter. It thereby provides a convenient model for passing and managing both types of complex data without having to repeatedly retrieve the data from the AppServer.

For more information on passing and managing a temp-table or ProDataSet as a ProDataGraph object, see Chapter 5, “Accessing ABL ProDataSets.”

SQL ResultSets mechanism

The SQL ResultSet provides a data streaming model for accessing temp-tables only and is the only mechanism for accessing complex data in the Java Open Client prior to OpenEdge Release 10.1A. This model works similar to a one-way tape reader or writer. It allows you to access the records (rows) of a temp-table parameter in only one direction, from beginning to end. To access previous rows, you must re-start the retrieval from the beginning. It is provided mainly for backward compatibility.

For more information on passing and managing a temp-table as an SQL ResultSet, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

Choosing the parameter passing mechanism

The ProDataGraph is the default mechanism for accessing temp-tables and ProDataSets. However, you can access temp-table parameters as an SQL ResultSet in a given method for the following conditions:

- The Java Open Client proxies are generated from an Open Client interface defined in an OpenEdge release prior to Release 10.1A.
- A given proxy method does not also pass a ProDataSet parameter (not available in releases prior to 10.1A).

Thus, ProxyGen provides an option for you to select the SQL ResultSet as the mechanism for passing temp-table parameters. You can specify this option for an entire AppObject/SubAppObject or for individual methods (procedures and user-defined functions). However, if a given method passes a ProDataSet, this option is unavailable and you must map any temp-table parameter to a ProDataGraph in that method.
Also to facilitate backward compatibility, in Open Client project files created prior to OpenEdge Release 10.1A, the SQL ResultSet option is selected by default.

For more information, see the ProxyGen help and *OpenEdge Development: Open Client Introduction and Programming*.

**Mapping temp-table field data types and meta data**

The mappings for the individual fields of a temp-table differs from the mappings available for other types of proxy method parameters. Also, OpenEdge supports a common mechanism for specifying the ABL data types of temp-table fields in the metadata for Java parameters that map to them.

**Java data type mappings for temp-table fields**

The Java data types for mapping a temp-table fields differ between the OpenEdge ProDataGraph and SQL ResultSet interfaces, because of the differences between the Java SDO and JDBC standards. For the list of Java data types available to map temp-table fields using a ProDataGraph, see Chapter 5, “Accessing ABL ProDataSets.” For the list of Java data types available to map temp-table fields using an SQL ResultSet, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

**Meta data for specifying the ABL data types of temp-table fields**

The Java Open Client does support a common mechanism for specifying the ABL data types in the metadata for temp-table fields, whether they are mapped using the OpenEdge ProDataGraph or the SQL ResultSet. This metadata support is provided by the com.progress.open4gl.Parameter class.

Wherever you specify the ABL data type of an individual temp-table field, you can use the Parameter class constants listed in Table 4–5. Thus, the sections of this manual that describe how to create temp-table field metadata refer to the class constants listed here.

<table>
<thead>
<tr>
<th>Progress temp-table field data type</th>
<th>com.progress.open4gl.Parameter class constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>PRO_BLOB</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>PRO_CHARACTER</td>
</tr>
<tr>
<td>CLOB</td>
<td>PRO_CLOB</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>PRO_COMHANDLE</td>
</tr>
<tr>
<td>DATE</td>
<td>PRO_DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>PRO_DATETIME</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>PRO_DATETIMETZ</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PRO_DECIMAL</td>
</tr>
<tr>
<td>INT64</td>
<td>PRO_LONG</td>
</tr>
<tr>
<td>INTEGER</td>
<td>PRO_INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>PRO_LOGICAL</td>
</tr>
</tbody>
</table>
Table 4–5: Parameter class constants for temp-table field metadata (2 of 2)

<table>
<thead>
<tr>
<th>Progress temp-table field data type</th>
<th>com.progress.open4gl.Parameter class constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>PRO_RAW</td>
</tr>
<tr>
<td>RECID</td>
<td>PRO_RECID</td>
</tr>
<tr>
<td>ROWID</td>
<td>PRO_RowID</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
</tbody>
</table>
Accessing ABL ProDataSets

An ABL ProDataSet is an in-memory relational data object that can encapsulate one or more ABL temp-tables (sets of data rows) and any parent-child relationships among them. It also includes mechanisms to track changes, which facilitates synchronization with any data sources used to initialize it. In effect, a ProDataSet models a relational database, its entity relations, and its change state in memory. Because of the power and efficiency with which it can organize and communicate complex relational data, the ProDataSet is a primary mechanism for bundling and passing complex data within applications that conform to the OpenEdge Reference Architecture (OERA).

The Java Open Client allows you to pass ProDataSets as application service parameters by mapping them to an OpenEdge class, ProDataGraph. The ProDataGraph class implements and extends the DataGraph interface defined by Java Service Data Objects (Java SDO), an emerging standard for representing and managing complex relational data in Java applications. In addition to ProDataSets, you can also use the ProDataGraph class to pass temp-table parameters of an OpenEdge application service (in place of SQL ResultSet parameters used in Java Open Clients prior to OpenEdge Release 10.1A). Thus, you can use a Java Open Client to implement the presentation layer of an OpenEdge application that conforms to the OERA.

For more information on:

- ProDataSets, see OpenEdge Development: ProDataSets
- OERA, see Progress Software Development Network (PSDN) at: http://communities.progress.com/pcom/community/psdn

This chapter describes how to use a ProDataGraph in the following sections:

- ProDataSets and using ProDataGraphs to access them
- Preparing and passing ProDataSets as ProDataGraph parameters
- Updating a ProDataSet
- Passing temp-tables as ProDataGraph parameters
• ProDataGraph class
• ProDataObject class
• ProChangeSummary class
• ProDataGraphMetaData class
• ProDataObjectMetaData class
• ProDataRelationMetaData class
• Using Java SDO classes to access Property meta data
• Sample application
ProDataSets and using ProDataGraphs to access them

A single ProDataGraph parameter in an Open Client method maps to a single ProDataSet parameter passed by the application service. The ProDataGraph is thus designed to pass the content and structure of the ProDataSet without any loss of data.

ProDataSet structure and usage

An ABL ProDataSet encapsulates a set of temp-tables through a collection of buffers that refer to these temp-tables. Each temp-table can hold a specified subset of data that it receives from one or more data sources, as determined by the application service. These data sources can include database tables, operating system files, and just about any source of data suitable for retrieval and storage as a well-defined set of fields in tabular form. The ProDataSet can optionally include data-relations, which specify parent-child relationships among the temp-table buffers, and indexes to manage large sets of data efficiently. Data-relations and indexes allow the application service to more easily fill and navigate the ProDataSet with data.

Abstracting data sources

To the extent that the set of temp-tables in a ProDataSet appears to reflect the tables of a single database, the ProDataSet can model and thus represent an abstraction of that database. Thus, a ProDataSet allows the application service to maintain an atomic subset of its data sources that it can pass as a unit to its clients and other services. As such, the ProDataSet is entirely separate from and can be managed independently of the data sources to which it is related. The data-relations of a ProDataSet can closely model the existing entity relations among its data sources using similar key relationships, or they can establish entirely different relationships among the temp-tables of the ProDataSet, as defined by the application service.

Tracking changes

A ProDataSet also provides a means to track changes to its initial data, using before-image versions of its temp-tables. This mechanism includes a means to identify and correlate these changes with other changes that may have occurred to its original data sources. Thus, the ProDataSet allows the application service to maintain a well-ordered business state that it can share with its clients and other services.

The rest of this chapter makes reference to specific ProDataSet syntax, as necessary to describe the operation and usage of the ProDataGraph class. For more information on the ABL ProDataSet and its syntax, see OpenEdge Development: ProDataSets.

Note: The function and capabilities of ProDataSets also closely mirror the function and capabilities of the Microsoft ADO .NET DataSets available for use by .NET Open Clients. For more information, see OpenEdge Development: .NET Open Clients.

ProDataGraph function and foundations

The ProDataGraph class contains all of the data and schema information required to exchange information with a corresponding ProDataSet or temp-table parameter in an application service. Thus, you can use a ProDataGraph parameter to map any of the following parameter types in the application service:

- **DATASET** — Static ProDataSet parameter
- **DATASET-HANDLE** — Dynamic ProDataSet parameter
• **TABLE** — Static temp-table parameter

• **TABLE-HANDLE** — Dynamic temp-table parameter

As long as the ProDataGraph is properly prepared, ABL handles all the requirements of passing data between a ProDataGraph parameter and the corresponding application service parameter using these different parameter types. This ABL support also guarantees that Java client programming requirements are mostly identical for both static and dynamic versions of equivalent ProDataSet and temp-table parameters.

**Note:** The only difference is how the schema can be passed for input static and dynamic parameters. For more information, see the “Passing a ProDataGraph as INPUT or INPUT-OUTPUT” section on page 5–17.

While much of this chapter applies to both individual temp-table and ProDataSet parameters, for more information on passing temp-tables as ProDataGraph parameters, see the “Passing temp-tables as ProDataGraph parameters” section on page 5–29.

**Java SDO Foundations**

The ProDataGraph class is based on the DataGraph interface specified by the Java Service Data Objects (Java SDO) specification published jointly by BEA and IBM as Java Specification Request (JSR) 235. The OpenEdge class is an extension of the Java SDO reference implementation included as part of the IBM Eclipse Modeling Framework (EMF 2.0.1).

**Note:** If the work of the JSR 235 Expert Group supersedes the current Java SDO API, the OpenEdge ProDataGraph implementation might have to change in the future.

The Java SDO interface specification consists of several interface objects, including the DataGraph, which closely model the ABL ProDataSet. Like the ProDataSet, it provides:

• Relationship support

• A detached data source

• XML serialization

• Data introspection (meta data or schema support)

• Change tracking

The DataObject is the fundamental interface of Java SDO and is the Java SDO representation of structured data. It encapsulates a specified set of properties (Property objects). Each property has a Type object, which can specify one of the following data types:

• A primitive or derived type, such as int (primitive) or Date (derived), typically representative of a column property

• A reference type, which refers to the type of another DataObject, and is representative of a reference property

Each DataObject provides read and write access methods for all its properties. (For more information on using the Java SDO Property and Type interfaces, see the “Using Java SDO classes to access Property meta data” section on page 5–63.)
A DataGraph encapsulates a tree of DataObject instances starting with a single root DataObject link. The reference properties of each DataObject link to other DataObject instances in the tree. In a DataGraph of relational data, a DataObject commonly represents a row of data, where its column properties represent the data fields of the row and its reference properties link to ordered or otherwise related rows. Foreign keys are represented by references to DataObject instances representing a row or rows in another table. Thus, reference properties can maintain relationships between parent and child tables. The rows of a single table are represented by a DataObject list (java.util.List).

The Property and Type interfaces that define data fields in a table row represented by a DataObject also provide the Java SDO meta data API. This API provides access to the DataObject schema, allowing the client to introspect the DataObject to get its type information.

Thus, the whole character of a DataGraph depends on the specific definitions of the DataObject instances in its tree. While this tree seems to imply a hierarchical structure, it can be used to model many other complex data structures, depending on how its DataObject instances are defined and interrelated. In order to more closely map a ProDataSet, the OpenEdge ProDataGraph represents a constraint on the standard DataGraph. A ProDataGraph encapsulates a tree of DataObject instances, starting with a specially-defined root DataObject. This root DataObject essentially references the temp-tables (DataObject lists) contained by the ProDataGraph and its corresponding ProDataSet. Each DataObject in a list is implemented as an OpenEdge ProDataObject class, which is a DataObject extension designed to map a temp-table record (table row). (For more information on this tree structure, see the “ProDataGraph object model” section on page 5–7.)

A DataGraph also encapsulates a ChangeSummary object whose interface tracks changes to any of its underlying DataObject instances. The ChangeSummary object is initially empty and populated as the DataGraph is modified. This ChangeSummary allows Java SDO to apply changes back to the data source of a DataGraph by comparing changed properties and DataObject instances with the current state of the data source. The ProDataGraph class encapsulates a corresponding ProChangeSummary object, which extends the ChangeSummary interface to more closely model the operation of the corresponding ProDataSet.

Java SDO also provides components called Data Mediator Services, which provide access to heterogeneous data sources for a DataGraph. They enable the DataGraph to function independently of any one data source so it can serve as a common unit of transfer for data throughout an application. The OpenEdge ProDataGraph does not require Data Mediator Services, as the data source for a ProDataGraph (and its corresponding ProDataSet) is typically managed by the OpenEdge application service.

For more information on:

- Documentation for the Service Data Objects in EMF 2.0.1, see http://eclipse.org/emf/docs.php
- JSR 235, see http://www.jcp.org/en/jsr/detail?id=235
Java SDO extensions

While Java SDO closely models the capabilities of the ABL ProDataSet, the ProDataGraph extensions provide additional support to address the following Java SDO limitations:

- Java SDO provides a minimal meta data API that allows clients to introspect data in a DataGraph. However, to more easily define and access the meta data of a ProDataGraph, OpenEdge provides its own classes, ProDataGraphMetaDa and ProDataObjectMetaData, to store the schema for a ProDataGraph.

- Java SDO provides relationship support by allowing a DataObject (table row) to contain references to other DataObject instances (representing rows in another table). However, it provides no means to describe these relationships as provided for ProDataSets. There is no meta data that describes the field names in the parent and child tables that form the primary-foreign key relationships. OpenEdge provides its own ProDataRelationMetaData class to describe each entity relation (data-relation) in a ProDataGraph.

- Java SDO does not support the java.util.GregorianCalendar class as a data type for DataObject column properties. This data type is essential to map ProDataSet fields that have the DATETIME or DATETIME-TZ data type. The OpenEdge ProDataObject class supports the java.util.GregorianCalendar class as a data type for ProDataObject column properties (fields).

  Note: Java SDO supports the java.util.Date class, but many of its methods are deprecated as of JDK 1.1 because they are not suitable for internationalization.

- The Java SDO ChangeSummary object that tracks changes to any DataObject rows in a DataGraph returns these changes as a list of DataObject rows. However, to efficiently marshall these updates back to the AppServer, these changed rows must be returned to the AppServer as a changes-only ProDataGraph, which maps to a changes-only ProDataSet in the application service. The OpenEdge ProChangeSummary class tracks all ProDataObject changes and returns them in a changes-only ProDataGraph.

Table 5–1 summarizes all the OpenEdge classes that support the ProDataGraph extension to Java SDO. They all reside in the com.progress.open4gl package. For more information on how these objects work together, in a typical ProDataGraph, see the “ProDataGraph object model” section on page 5–7.

Table 5–1: OpenEdge classes that extend Java SDO

<table>
<thead>
<tr>
<th>OpenEdge class</th>
<th>Basis</th>
<th>For more information, see . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDataGraph</td>
<td>Extends DataGraph to map ProDataSets</td>
<td>The “ProDataGraph class” section on page 5–30</td>
</tr>
<tr>
<td>ProDataObject</td>
<td>Extends DataObject to support java.util.GregorianCalendar and provide relational navigation among tables</td>
<td>The “ProDataObject class” section on page 5–36</td>
</tr>
</tbody>
</table>
## ProDataGraph object model

In general, a ProDataGraph directly references the following objects:

- One root DataObject, which directly references the ProDataObject list for each table in the ProDataGraph.

- One ProChangeSummary, which contains a history of all changes to ProDataObject instances in the ProDataGraph.

- One ProDataGraphMetaData, which contains information about the original ABL ProDataSet, such as its name, and directly references the following objects:
  - One ProDataObjectMetaData for each table in the ProDataGraph that specifies the schema of the table.
  - One ProDataRelationMetaData for each data-relation defined for the ProDataGraph.

---

### Table 5–1: OpenEdge classes that extend Java SDO

<table>
<thead>
<tr>
<th>OpenEdge class</th>
<th>Basis</th>
<th>For more information, see . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProChangeSummary</td>
<td>Extends ChangeSummary to return all ProDataObject changes in a changes-only ProDataGraph</td>
<td>The “ProChangeSummary class” section on page 5–48</td>
</tr>
<tr>
<td>ProDataGraphMetaData</td>
<td>OpenEdge-based class to more easily manage ProDataGraph schema information</td>
<td>The “ProDataGraphMetaData class” section on page 5–50</td>
</tr>
<tr>
<td>ProDataObjectMetaData</td>
<td>OpenEdge-based class to more easily manage ProDataObject schema information</td>
<td>The “ProDataObjectMetaData class” section on page 5–53</td>
</tr>
<tr>
<td>ProDataRelationMetaData</td>
<td>OpenEdge-based class to describe ProDataGraph data-relations</td>
<td>The “ProDataRelationMetaData class” section on page 5–59</td>
</tr>
</tbody>
</table>
Figure 5–1 shows a ProDataGraph instance, the objects it encapsulates, and their relationships. The **Root DataObject** references ProDataObject lists for four tables: **Customer**, **Order**, **OrderLine**, and **SalesRep**. The ProDataGraph encapsulates two ProDataRelationMetaData objects shown as **ProDataRelation1** (Order rows of each Customer) and **ProDataRelation2** (OrderLine rows of each Order). Note that you can navigate through the related (parent and child) tables of a ProDataGraph using extended methods on ProDataObject. This is unlike a DataGraph, which requires that you follow reference properties directly to locate DataObject lists referenced by a DataObject.

![ProDataGraph object model](image)

**Note:** Note that the ProDataGraphMetaData and ProDataObjectMetaData objects are not shown, but the content of these objects is implied by the data in the example.
Preparing and passing ProDataSets as ProDataGraph parameters

Table 5–2 shows how the components of an ABL ProDataSet map to a ProDataGraph and its supporting objects.

Table 5–2: ABL ProDataSet mappings to Java ProDataGraph

<table>
<thead>
<tr>
<th>ABL component</th>
<th>Java object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDataSet (static or dynamic)</td>
<td>ProDataGraph containing a schema-matching ProDataGraphMetaData object</td>
</tr>
<tr>
<td>Temp-table (static or dynamic)</td>
<td>ProDataObject list supported by a schema-matching ProDataObjectMetaData object</td>
</tr>
<tr>
<td>Buffer (temp-table row)</td>
<td>Any corresponding ProDataObject</td>
</tr>
<tr>
<td>Before-table</td>
<td>The original property values derived from a ProDataObject list in the ProDataGraph returned by the ProChangeSummary.getChanges(dataObj)</td>
</tr>
<tr>
<td>Field</td>
<td>Property object in a ProDataObject</td>
</tr>
<tr>
<td>Data-relation</td>
<td>ProDataRelationMetaData</td>
</tr>
<tr>
<td>Any primary or unique index</td>
<td>ProDataRelationMetaData</td>
</tr>
<tr>
<td>Any non-unique index</td>
<td>Not mapped</td>
</tr>
</tbody>
</table>

1. To return values for a before-table, you get a table (ProDataObject list) from a changes-only ProDataGraph and return the list of original property values for each DataObject dataObj using ProChangeSummary.getOldValues(dataObj).

For more information on the ABL ProDataSet, see OpenEdge Development: ProDataSets. The following sections describe the features for passing ProDataGraph parameters.

Preparing and managing a ProDataGraph parameter

ABL requires a different programming model to manage DATASET or DATASET–HANDLE (static or dynamic ProDataSet) parameters. However, to manage a ProDataGraph parameter in a Java Open Client application, you use the same programming model to map both types of ProDataSet parameters.
Parameter passing modes

From the viewpoint of the Java Open Client, the minimum information that you must provide in a ProDataGraph to map a ProDataSet parameter depends on the parameter passing mode of the application service. For example:

- **OUTPUT parameters** — You must pass a `com.progress.open4gl.ProDataGraphHolder` object instance (see the “ProDataGraphHolder class” section on page 5–10). Use of this holder class is similar to holder classes for other parameter data type mappings. On output, the holder object contains a reference to a `com.progress.open4gl.ProDataGraph` object that you can access. For more information on managing a ProDataGraph for an OUTPUT ProDataSet parameter, see the “Passing a ProDataGraph as OUTPUT” section on page 5–12.

- **INPUT parameters** — You must pass a `com.progress.open4gl.ProDataGraph` object instance. Before passing the ProDataGraph, as a minimum, you must create and initialize (or obtain) the object with a ProDataGraphMetaData object specifying the number and meta data of its underlying tables (ProDataObjectMetaData objects), and which must map to the ProDataSet temp-table schemas. In this meta data, you must also include ProDataRelationMetaData objects that map to all DATA-RELATION objects defined in a static ProDataSet parameter. Differences do exist for how the schema must be passed between static and dynamic ProDataSets. For more information on managing a ProDataGraph for an INPUT ProDataSet parameter, see the “Passing a ProDataGraph as INPUT or INPUT-OUTPUT” section on page 5–17.

- **INPUT-OUTPUT parameters** — You must pass an instance of `com.progress.open4gl.ProDataGraphHolder` that contains a reference to a client-supplied `com.progress.open4gl.ProDataGraph` object instance (see the “ProDataGraphHolder class” section on page 5–10). Use of this holder class is similar to holder classes for other parameter data type mappings. The requirements for supplying the input ProDataGraph are the same as for passing an INPUT parameter. On output, the same holder object is changed to reference a `com.progress.open4gl.ProDataGraph` object that represents the ProDataSet that is passed as output. For more information on managing a ProDataGraph for an INPUT-OUTPUT ProDataSet parameter, see the “Passing a ProDataGraph as INPUT or INPUT-OUTPUT” section on page 5–17.

ProDataGraphHolder class

The Example 5–1 class definition shows the constructor and method signatures for the ProDataGraphHolder class.

**Example 5–1: ProDataGraphHolder class definition**

```java
public class ProDataGraphHolder extends Holder {
    ProDataGraphHolder();
    ProDataGraphHolder(com.progess.open4gl.ProDataGraph);
    void setProDataGraphValue(com.progess.open4gl.ProDataGraph);
    ProDataGraph getProDataGraphValue();
    public boolean isNull();
}
```

For more information on holder classes, see Chapter 4, “Passing Parameters.”
Managing ProDataObject mappings to temp-tables

OpenEdge releases prior to 10.1A provide only a streaming data model for accessing temp-tables (TABLE and TABLE-HANDLE only) as parameters using the java.sql.ResultSet object. This model requires the client application (for output parameters) or the Open Client proxy (for input parameters) to request data from the sender (through the SQL ResultSet) instead of allowing the sender to set the data in the receiver. It relies on a call back mechanism to access the data one row at a time. Thus, a temp-table is accessible in only one direction, like a one-directional tape. For more information on the SQL ResultSet mechanism, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

The Java SDO architecture available in the current release is based on a disconnected data model that caches rows of temp-table data in memory using the DataObject interface. This model provides a scrollable access mechanism that allows you to directly access any row of a temp-table that is part of an ABL DATASET or DATASET-HANDLE parameter and also any row of single temp-table passed as a TABLE or TABLE-HANDLE parameter. (For more information on accessing TABLE and TABLE-HANDLE parameters, see the “Passing temp-tables as ProDataGraph parameters” section on page 5–29.)

The SQL ResultSet and Java SDO DataObject mechanisms support a different set of mappings for fields in a temp-table. The OpenEdge ProDataObject further extends this DataObject mechanism to map ABL data types not supported in the base mechanism. The following sections describe the mappings for temp-table fields supported by this extended ProDataObject mechanism.

Mapping single-valued fields

As described in a previous section (see the “Java SDO Foundations” section on page 5–4), a DataObject (and its ProDataObject extension) represents a temp-table field as a column Property object containing its value, with an associated Type object, specifying its data type. Table 5–3 shows the ProDataObject mappings to temp-table fields supported by ABL.

Table 5–3: Java ProDataObject mappings to temp-table fields

<table>
<thead>
<tr>
<th>ABL temp-table field data type</th>
<th>Java ProDataObject Property data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>CLOB</td>
<td>java.io.String</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>long</td>
</tr>
<tr>
<td>DATE</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DATETIME</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>byte[]</td>
</tr>
</tbody>
</table>
In addition to the value, the Property object contains other information about the ProDataObject column that corresponds to the temp-table field, including its position in the row of columns (indicated by a property index) and the property name, which is identical to the ABL name of the temp-table field.

The ProDataObject class has methods for setting and getting the values of its column Property objects, accessed by property name or index according to data type. For more information, see the “ProDataObject class” section on page 5–36.

Mapping array fields

In ABL, a temp-table field can be defined with an extent (an integer value). This extent specifies the size of an array field, which is a one-dimensional array of the specified data type. Similarly in a Java SDO DataObject, the Type associated with a Property object can be defined as many-valued. Thus, a many-valued property of any Java data type listed in Table 5–3 can be mapped to a temp-table array field of the corresponding ABL data type.

The ProDataObject class has methods for setting and getting the values of its many-valued column Property objects, accessed as List objects by property name or index. To determine if a Property object is many-valued, you can use the isMany() method on the Property. For more information on working with the values of many-valued properties, see the descriptions of the getList() and setList() methods in the “ProDataObject class” section on page 5–36.

Mapping the Unknown value (?)

In ABL, a temp-table field can be specified with an Unknown value (?), similar to null in Java. If you need to determine if a field has been set to the Unknown value (?), you can use one of the ProDataObject isSet() methods to test property by name or index. These methods return true if the corresponding temp-table field is set to a value other than the Unknown value (?) and return false if the field is set to the Unknown value (?). For more information on the isSet() methods, see the “ProDataObject class” section on page 5–36. If you want to set a column property so the corresponding temp-table field is set to the Unknown value (?), do not set the property to any value.

Specifying initial values

Java SDO does not support setting an initial (or default) value for a property within a DataObject. So, you must explicitly set all initial values for column properties.

Passing a ProDataGraph as OUTPUT

To pass a ProDataGraph as output, you create a ProDataGraphHolder instance to contain the ProDataGraph, and pass the holder as the proxy method parameter that maps to the application service DATASET or DATASET–HANDLE OUTPUT parameter. You do not need to create an initial ProDataGraph object to instantiate the ProDataGraphHolder instance. The Java Open Client interface creates the ProDataGraph required to hold the data returned from the corresponding output ProDataSet on the AppServer.
Preparing and passing ProDataSets as ProDataGraph parameters

To return an output ProDataGraph, get the value of the getProDataGraphValue() method invoked on the output ProDataGraphHolder parameter.

The actual value and content of the output ProDataGraph can vary depending on how the ProDataSet is defined by the application service. For example, if the OUTPUT ProDataSet is passed as a DATASET-HANDLE (dynamically), this value can be null. A null value indicates that the DATASET-HANDLE parameter was set to the Unknown value (?) or undefined (defined with no schema and data) by the application service. A dynamic ProDataSet might also be passed without data-relations created for it, and so on.

Once you have the output ProDataGraph, you can access its data, depending on whether you already know the schema of the corresponding ProDataSet.

Accessing a ProDataGraph with a known schema

If you already know the schema of the ProDataSet, you can use this information to access the data in the ProDataGraph. If you do not know the schema, see the “Accessing the ProDataGraph meta data for an unknown schema” section on page 5–15.

To access the data in a ProDataGraph:

1. Return the ProDataObject list from the ProDataGraph that maps to a specified temp-table using the following ProDataGraph method, where tableName is the ABL name of the temp-table:

   Syntax
   ```java
   java.util.List getProDataObjects(String tableName)
   ```

2. Iterate through the ProDataObject list returned in Step 1 and locate a ProDataObject instance that contains data that you want.

3. Use the appropriate ProDataObject property access methods to return the values of specified column properties of the ProDataObject found in Step 2, where propertyIndex is the index into the ProDataObject column property list, and name is the ABL name of the temp-table field that the property maps to.

   Note: For more information on determining the propertyIndex of a column property and other information about column properties of a ProDataObject, see the “Using Java SDO classes to access Property meta data” section on page 5–63.

   - To return the value of a specified single-valued column property in the form of the Property data type, use one of the following ProDataObject methods:

     Syntax
     ```java
     DataType getDataTypeName(int propertyIndex)
     DataType getDataTypeName(String name)
     ```

     Where DataType is the full Java classname or intrinsic type name of the property data type and DataTypeName is a name that closely matches the name of the data type that the method returns:
To identify the Java intrinsic data type or class of the column property that maps to the ABL data type of the temp-table field, see Table 5–3. For example, the following two methods return values for an int property (mapped to an INTEGER field) and a BigDecimal property (mapped to a DECIMAL field):

**Syntax**

```java
int getInt(String name)
java.math.BigDecimal getBigDecimal(String name)
```

- To return the value of a specified many-valued property (which maps to a temp-table array field), use one of the following ProDataObject methods:

**Syntax**

```java
java.util.List getList(int propertyIndex)
java.util.List getList(String name)
```

The objects in the List all have the data type of the column Property.

**Note:** You can check that a column Property is many-valued, and therefore returns a List, by testing the value returned by its isMany() method.

- To return the value of a specified column property in the form of the `java.lang.Object` class, use one of the following ProDataObject methods:

**Syntax**

```java
java.lang.Object get(int propertyIndex)
java.lang.Object get(String name)
```

The returned Object has the data type of the specified column property, including `java.util.List` if the property is many-valued.

4. If you need to access parent or child relations of the ProDataObject that you access in Step 3, you can use the following methods:

- To return a list of child rows for the specified ProDataObject, use the following ProDataObject method:

**Syntax**

```java
java.util.List getChildRows(java.lang.String relationName)
```

The method returns a list of child rows (ProDataObject instances) for this parent ProDataObject according to the data-relation specified by name (`relationName`). Iterate through the list to locate a ProDataObject that you want to access.
Preparing and passing ProDataSets as ProDataGraph parameters

- To return a parent row for the specified ProDataObject, use the following ProDataObject method:

**Syntax**

```
ProDataObject getParentRow(java.lang.String relationName)
```

The method returns a parent ProDataObject for this child ProDataObject according to the data-relation specified by name (relationName).

**Note:** For a ProDataGraph, the relationName identifies a DATA-RELATION defined in the corresponding ProDataSet parameter of the application service, and this name is identical to the ABL name of the specified DATA-RELATION.

Once you have a child or parent ProDataObject, you can continue with Step 3 to access its data.

For more information on ProDataObject methods, see the “ProDataObject class” section on page 5–36.

**Accessing the ProDataGraph meta data for an unknown schema**

You might not know any or all of the schema for an output ProDataGraph. In this case, you can introspect the ProDataGraph to identify all of its meta data, including the names and locations of all ProDataGraph components that map to the corresponding ProDataSet OUTPUT parameter. These components can include only the functional Java components of the ProDataGraph or they can also include the names, locations, and data types of the ProDataSet, its temp-tables, fields, and data-relations, as you might require.

You can get meta data for an unknown ProDataGraph in different ways. The following procedure shows one approach for accessing ProDataGraph, primarily by identifying and using component names. You can also identify component locations and use indexes (rather than names) to more directly and efficiently access components.

**To access the meta data of a ProDataGraph:**

1. If you need the ABL name of the ProDataSet that the ProDataGraph maps to, use the following ProDataGraph method:

**Syntax**

```
java.lang.String getProDataGraphName()
```

2. To get ABL schema information for temp-tables or get information on data-relations that have been passed in the ProDataGraph, get its associated ProDataGraphMetaData object using the following ProDataGraph method:

**Syntax**

```
ProDataGraphMetaData getMetaData()
```
3. To get ABL schema information for each temp-table passed in the ProDataGraph:
   a. Get the ABL names for all ProDataSet temp-tables (ProDataObject collections) in the ProDataGraph using the following ProDataGraphMetaData method:

   **Syntax**

   ```java
   java.lang.String[] getTableNames()
   ```

   b. Get the ProDataObjectMetaData for a temp-table using the following ProDataGraphMetaData method:

   **Syntax**

   ```java
   ProDataObjectMetaData getTableMetaData(int idx)
   ```

   Where the `idx` value corresponds to the index into the list of table names that refers to the name of the given temp-table.

c. Some useful schema information you can get from this meta data for an output temp-table is the field name, the ABL data type, any extent (for an array), and the specified user order of each temp-table field using the following ProDataObjectMetaData methods:

   **Syntax**

   ```java
   int getFieldName(int propertyIndex)
   int getProType(int propertyIndex)
   int getExtent(int propertyIndex)
   int getUserOrder(int propertyIndex)
   ```

   The `propertyIndex` is a 0-based value that you can obtain by looping through the number of temp-table fields returned by the following ProDataObjectMetaData method:

   **Syntax**

   ```java
   int getFieldCount()
   ```

   This value is also identical to the index in the corresponding ProDataObject property list that references the corresponding column property (see the “Using Java SDO classes to access Property meta data” section on page 5–63).

   For more information on ProDataObjectMetaData methods, see the “ProDataObjectMetaData class” section on page 5–53.

4. To get the meta data for data-relations passed in the ProDataGraph:

   a. Get the number of ProDataRelationMetaData objects in the ProDataGraph using the following ProDataGraphMetaData method:

   **Syntax**

   ```java
   int getNumRelations()
   ```
b. Get each ProDataRelationMetaData object associated with the ProDataGraph using the following ProDataGraphMetaData method:

**Syntax**

```java
ProDataRelationMetaData getRelationMetaData(int idx)
```

Where `idx` starts at 0 for the number of data-relations in the ProDataGraph.

c. Some useful data-relation information you can get from each ProDataRelationMetaData object for an output ProDataGraph is the specified data-relation name, parent table name, parent table index columns, child table name, and child table index columns. These are all specified according to a corresponding ProDataSet data-relation and accessible using the following ProDataRelationMetaData methods:

**Syntax**

```java
String getRelationName()
String getParentTable()
int[] getParentColumns()
String getChildTable()
int[] getChildColumns()
```

For more information on ProDataRelationMetaData methods, see the “ProDataRelationMetaData class” section on page 5–59.

For more information on ProDataGraphMetaData methods, see the “ProDataGraphMetaData class” section on page 5–50.

### Passing a ProDataGraph as INPUT or INPUT-OUTPUT

To pass a ProDataGraph for INPUT or INPUT-OUTPUT to an application service, you must as a minimum, create a ProDataGraph that maps to the corresponding ProDataSet parameter of the application service. While the mechanisms for programming an input ProDataGraph are identical regardless of the type of ProDataSet parameter it maps to, the requirements for preparing the ProDataGraph depend on the ProDataSet parameter passing mode, data type, and requirements of the application service.

To pass a ProDataGraph when the parameter passing mode of the ProDataSet is INPUT:

1. Prepare the ProDataGraph as required. For more information, see the “Preparing an input ProDataGraph” section on page 5–18.

2. Pass the ProDataGraph directly as a parameter to the proxy method.
By comparison, handling INPUT-OUT parameters is more complex.

To pass a ProDataGraph when the parameter passing mode of the ProDataSet is INPUT-OUTPUT:

1. Prepare the ProDataGraph as required. For more information, see the “Preparing an input ProDataGraph” section on page 5–18.

2. Create a ProDataGraphHolder object that includes a reference to the ProDataGraph. You can set the reference by using the constructor or by using the setProDataGraphValue() method on the holder object after you create it.

3. Pass the ProDataGraphHolder as a parameter to the proxy method.

4. Get the returned ProDataGraph from the ProDataGraphHolder using the getProDataGraphValue() method on the holder object and access it similar to an OUTPUT parameter. For more information, see the “Passing a ProDataGraph as OUTPUT” section on page 5–12.

Note: For some application services, especially those that conform to the OERA, you might never have to prepare a ProDataGraph parameter with meta data. You might well receive the initial ProDataGraph as an application service OUTPUT parameter, update the ProDataGraph in your Java Open Client, and pass the modified ProDataGraph back to the application service as an INPUT-OUTPUT parameter without having to touch any meta data. For more information, see the “Updating a ProDataSet” section on page 5–24.

Preparing an input ProDataGraph

If the application service defines a DATASET (static ProDataSet) parameter, you must create a ProDataGraph that at least represents an empty ProDataSet and contains the meta data (ProDataGraphMetaData) that describes the temp-tables and data-relations defined for the ABL static ProDataSet. There must be one ProDataObjectMetadata object for each temp-table and one ProDataRelationMetadata object for each DATA-RELATION object defined by the static ProDataSet. So, the meta data in the ProDataGraph must match the schema of the static ProDataSet in every particular.

Note: While you must define a ProDataGraph with all the meta data required to map a static ProDataSet schema, you can tell OpenEdge not to send temp-table schema information to the AppServer in order to optimize data transfer over the network. For more information, see the setNoSchemaMarshal() method in the “ProDataObjectMetaData class” section on page 5–53.

If the application service defines a DATASET-HANDLE (dynamic ProDataSet), you can pass a ProDataGraph with any or no meta data, including any or no ProDataRelationMetaData objects. As a minimum, you can pass a ProDataGraph with empty meta data (an empty ProDataGraphMetaData object), which produces an Unknown value (?) in the input DATASET-HANDLE. However, in practice, you must pass a ProDataGraph with meta data that represents the ProDataSet schema expected by the application service. So, most of the time, the programming for a ProDataGraph that maps to static or dynamic ProDataSet is identical.

For more information on the differences between static and dynamic ProDataSets, see OpenEdge Development: ProDataSets.
For a typical application, the general procedure for preparing a ProDataGraph for input is a two step process.

To prepare a typical ProDataGraph for input:
1. Create and initialize a ProDataGraph object with its meta data.
2. Create and add ProDataObject instances to the ProDataGraph.

Creating and initializing a ProDataGraph object with meta data

You can use a number of approaches for creating and initializing a ProDataGraph with meta data.

These approaches include two main variations:
1. Start by creating the meta data and use it to create the ProDataGraph using the appropriate constructor.
2. Start by creating a Java SDO DataGraph that conforms to the OpenEdge ProDataGraph object model, and create the ProDataGraph from this using the appropriate constructor.

The following procedure suggests one such approach starting with the meta data. For information on creating an OpenEdge ProDataGraph from a Java SDO DataGraph, see the appropriate ProDataGraph constructor in the “ProDataGraph class” section on page 5–30.

To create and initialize a ProDataGraph with meta data:
1. Create a ProDataGraphMetaData object using the following constructor:

   **Syntax**
   ```java
   ProDataGraphMetaData(String dataSetName)
   ```

   Where `dataSetName` is the ABL name of the ProDataSet in the application service.

2. Create a ProDataObjectMetaData object for each temp-table defined in the corresponding application service ProDataSet using the following constructor:

   **Syntax**
   ```java
   ProDataObjectMetaData(String tableName, int numFields, boolean bimageFlag, int numIndexes, String multiIxCols, String XMLNamespace, String XMLPrefix)
   ```

   The parameters specify, respectively:
   - The ABL name of the temp-table
   - The number of temp-table fields
   - An indication if there is a BEFORE-TABLE defined for the temp-table (required in order to update the temp-table)
   - The number of indexes on the temp-table
• A formatted string that specifies all the index information for this temp-table

• Any XML namespace (or null)

• Any XML prefix (or null)

For more information, see the “ProDataObjectMetaData class” section on page 5–53.

3. For each ProDataObjectMetaData object created in Step 2, add the column meta data to match the schema of a corresponding temp-table field. So, for each field in the temp-table, invoke the following ProDataObjectMetaData method:

**Syntax**

```java
void setFieldMetaData(int fieldNumber, String fieldName, int extent, int proType, int userOrder, int xmlMapping)
```

The parameters specify, respectively:

• The 1-based number of the temp-table field.

• The ABL field name.

• If an array field, a value greater than 1 indicating the extent.

• A value specified by a class constant defined in com.progress.open4gl.Parameter that indicates the ABL data type of the field. For more information, see the information on specifying data type meta data for temp-tables in Chapter 4, “Passing Parameters.”

• A 0-based user position order for the field.

• An XML serialization value for later use, currently set to 0.

For more information on creating table meta data, see the “ProDataObjectMetaData class” section on page 5–53.

4. Add each ProDataObjectMetaData object completed in Step 3 to the ProDataGraphMetaData object created in Step 1 using the following ProDataGraphMetaData method:

**Syntax**

```java
void addTable(ProDataObjectMetaData tablemd)
```
5. For each DATA-RELATION object defined for the ProDataSet in the application service, create a corresponding ProDataRelationMetaData object using the following constructor:

**Syntax**

```java
ProDataRelationMetaData(String name, ProDataObjectMetaData parent, ProDataObjectMetaData child)
ProDataRelationMetaData(String name, int parentIdx, int childIdx, int numPairs, String pairsList, )
```

The second constructor allows you to set foreign-primary key relationships for one or more columns between parent and child tables explicitly. The `parentIdx` and `childIdx` correspond to indexes into the list of table names returned by the `ProDataGraphMetaData` method, `getTableNames()`. You can also set these relationships in the instantiated `ProDataRelationMetaData` object using the overloaded `setColumns()` methods.

For more information on creating data-relations, see the “ProDataRelationMetaData class” section on page 5–59.

6. Add each `ProDataRelationMetaData` object completed in Step 5 to the `ProDataGraphMetaData` object created in Step 1 using the following `ProDataGraphMetaData` method:

**Syntax**

```java
void addDataRelation(ProDataRelationMetaData drmd)
```

7. Create the `ProDataGraph` from the `ProDataGraphMetaData` object completed in Step 4 and Step 6 using the following constructor:

**Syntax**

```java
ProDataGraph(ProDataGraphMetaData dgmd)
```

For more information on creating meta data for a `ProDataGraph`, see the “ProDataGraphMetaData class” section on page 5–50.
Adding data to a ProDataGraph

Once you have the ProDataGraph defined with its meta data, you can create and add the rows to the various tables (ProDataObject lists) and generate all the data-relation references between them to complete the ProDataGraph. You can add data to a ProDataGraph in different ways. The following procedure assumes that you know the ABL names of the temp-tables and fields and the field data types in the corresponding ProDataSet.

To add data to a ProDataGraph:

1. For each row you want to add to a temp-table, create a ProDataObject using the following ProDataGraph factory method:

   Syntax

   ProDataObject createProDataObject(String tableName)

   Where tableName is the temp-table name.

2. For each ProDataObject that you create, add the column property data using one of the following ProDataObject methods, where name is the ABL name for the corresponding temp-table field:

   - To set the value of a single-valued property, use the following method:

     Syntax

     void setDataTypeName(String name, DataType)

     Where DataType is the full Java classname or intrinsic type name of the property data type and DataTypeName is a name that closely matches the data type name for the value.

     To identify the Java data type or class of the column property that maps to the ABL data type of the temp-table field, see Table 5–3. For example, the following two methods set values for an int property (mapped to an INTEGER field) and a BigDecimal property (mapped to a DECIMAL field):

     Syntax

     void setInt(String name, int)
     void setBigDecimal(String name, java.math.BigDecimal)

   - To set the value of a many-valued property (which maps to a temp-table array field), use the following method, after loading the java.util.List object with the required values:

     Syntax

     void setList(String name, java.util.List)

     The values in the List all have the data type of the column property.
Preparing and passing ProDataSets as ProDataGraph parameters

• To set the value of a column property in the form of the java.lang.Object class, use the following method:

**Syntax**

```java
void set(String name, java.lang.Object)
```

The value in the Object has the data type of the column property or java.util.List if the property is many-valued.

You can also set the values of column properties with overloaded versions of these methods that index into the ProDataObject property list. For more information on determining the index of a column property and other information about column properties of a ProDataObject, see the “Using Java SDO classes to access Property meta data” section on page 5–63.

3. Add each ProDataObject, `dataObj`, filled with data in Step 2 to the ProDataGraph using the following ProDataGraph method:

**Syntax**

```java
void addProDataObject(ProDataObject dataObj)
void addProDataObject(int index, ProDataObject dataObj)
```

This method adds the ProDataObject to the collection (ProDataObject list) identified by its defined table name. Using an index, you can insert the ProDataObject at a location in the list. Otherwise, the object is added to the end of the list.

4. Once all the tables (ProDataObject lists) have been populated with rows of data in Step 3, generate the parent-child ProDataObject references specified by all the ProDataRelationMetaData objects contained in the ProDataGraph using the following ProDataGraph method:

**Syntax**

```java
void setChildTableReferences()
```

You can also generate these references one table at a time using overloads of this method. However, this is the most efficient method to generate data-relation references for a fully populated ProDataGraph.

For more information on methods for building ProDataGraphs, see the “ProDataGraph class” section on page 5–30.
Updating a ProDataSet

The previous section describes how to read ProDataSet data passed from the AppServer using a ProDataGraph and how to use a ProDataGraph to write ProDataSet data back to the AppServer. However, to manage updates to a data source on the AppServer, you must typically interact further with the ProDataGraph, as described in the following sections.

Typical cycle for ProDataSet updates

ProDataSet updates generally follow a typical pattern in the round trip from AppServer to client:

1. The Java client initially obtains the ProDataSet from the AppServer as a working ProDataGraph.

2. The client makes changes to the working ProDataGraph, and sends those changes back to the AppServer.

3. The AppServer matches the records in its changed ProDataSet back to the original data source, verifies if records were changed by other users, and resolves any conflicts.

4. The AppServer then applies the changes to the data source and passes back to the Java client the final versions of records that may have been further changed in another ProDataSet, along with any errors.

5. The Java client then handles any errors in the returned ProDataGraph and merges the final versions of data in this ProDataGraph into its working ProDataGraph.

6. The client continues with Step 2 and the cycle repeats until done.

The rest of this section describes how you can manage the ProDataGraph throughout this update cycle.

Managing a ProDataGraph through the update cycle

The OpenEdge ProDataGraph provides built-in support for tracking updates with its ProChangeSummary object. This object keeps track of what rows are changed (modified, added, or deleted) in the ProDataGraph and also keeps track of the original versions of modified or deleted rows so you can compare those rows against the current rows. For more information on the ProChangeSummary object, see the “ProChangeSummary class” section on page 5–48. In the procedure that follows, this and other OpenEdge support facilitates management of the ProDataSet update cycle in a Java Open Client application.

To manage a ProDataGraph through an update cycle:

1. Receive your working ProDataGraph as an OUTPUT parameter of the application service. For more information, see the “Passing a ProDataGraph as OUTPUT” section on page 5–12.

**Note:** While the actual parameter passing mode is application service dependent, an OUTPUT parameter is typical for an OERA-compliant application.
2. Make changes to the ProDataGraph on the client, as follows:

- For information on finding rows in a ProDataGraph, see the “Finding a row in a ProDataGraph” section on page 5–27.
- For information on modifying rows, see the “Adding data to a ProDataGraph” section on page 5–22.
- For information on adding rows, see the “Adding a row to a ProDataGraph” section on page 5–28.
- For information on deleting rows, see the “Deleting a row from a ProDataGraph” section on page 5–28.

**Note:** Because the application service sees the changes in its input ProDataSet using before-tables, all temp-tables subject to update are defined on the AppServer using the BEFORE-TABLE option. To verify that a temp-table whose data you plan to update is defined with this option, you can check the boolean value returned from the getBImageFlag() method on the ProDataObjectMetaData that corresponds to this temp-table. If the value is true, you can update the corresponding ProDataObject collection and the application service can handle the updates.

3. Return the ProChangeSummary object from your working ProDataGraph using the following ProDataGraph method:

**Syntax**

```
ProChangeSummary getProChangeSummary()
```

4. Extract a separate changes-only ProDataGraph from your working ProDataGraph using the following ProChangeSummary method:

**Syntax**

```
ProDataGraph getChanges()
```

5. Pass the changes-only ProDataGraph back to the application service using an INPUT-OUTPUT parameter. This allows the results to be passed back to the client in the same parameter.

**Note:** While the actual parameter passing mode is application service dependent, an INPUT-OUTPUT parameter is typical for an OERA-compliant application.
6. Process the results from the application service in the output of the changes-only ProDataGraph that you passed to the AppServer in Step 5. This ProDataGraph now contains any additional changes, including errors, returned from the AppServer, as follows:

   a. Handle any errors that might be returned for each ProDataObject as well as the ProDataGraph as a whole. For more information on checking errors, see the “Checking for errors” section on page 5–26.

   b. Check the final changes in the output changes-only ProDataGraph using the methods of its ProChangeSummary object. For more information, see the “ProChangeSummary class” section on page 5–48. Compare the changed rows to the corresponding row in your working output ProDataGraph. Update the original row and add or delete additional required rows as described in Step 2.

7. Once you have merged all final changes into your working ProDataGraph in Step 6, use the following method to accept all changes in the ProDataGraph:

   Syntax

   ```java
   void acceptChanges()
   ```

   This clears out the list of changes in the ProChangeSummary of your ProDataGraph, allowing it to accept more changes.

8. Delete the changes-only ProDataGraph from Step 5 and Step 6.

9. Update any UI appropriately for your updated working ProDataGraph.

10. You can now accept more changes to the working ProDataGraph, as in Step 2, restarting the update cycle, again.

### Checking for errors

On return from a proxy method call to an application service, you can check for errors in any updated output ProDataGraph. You can check for errors at the ProDataGraph, table, and row levels, and return error strings for the table and row levels. The error checking methods return true if there is an error at the specified level.

#### ProDataGraph level

You can check if any errors are returned in the output ProDataGraph using the following ProDataGraph method:

Syntax

```java
boolean hasError()
```
Table level

You can check if an error is returned for a table and get the error string using the following ProDataObject methods:

Syntax

```java
boolean hasTableError()
String getTableErrorString()
```

You only need to invoke these methods on one ProDataObject in a given table collection. These methods return the same information, which applies to the entire table, for any ProDataObject in the list.

Row level

You can check if an error is returned for a row and get the error string using the following ProDataObject methods:

Syntax

```java
boolean hasRowError()
String getRowErrorString()
```

Thus, these methods can return different information for each ProDataObject in a given table collection.

Finding a row in a ProDataGraph

You can find a row (ProDataObject instance) in a ProDataGraph in different ways, depending on how the tables (ProDataObject collections) are related (what ProDataRelationMetaData is associated with the ProDataGraph). If a table does not participate in any data-relations, the only way to find a row is to search through the entire ProDataObject list that you can get using the ProDataGraph getProDataObjects() method for a given table. If you know that a table you are searching participates in a data-relation, you can use the data-relation to find related parent and child rows for each ProDataObject in the table.

Finding parent rows

With the name of a data-relation that identifies a parent table, you can use the ProDataObject getParentRow() method to return the parent row for a ProDataObject instance.

Finding child rows

With the name of a data-relation that identifies a child table, you can use the ProDataObject getChildRows() method to return a list of child rows for a ProDataObject instance. You can then search this filtered list of child rows to find the appropriate ProDataObject.

For more information on retrieving data from an output ProDataGraph, see the “Passing a ProDataGraph as OUTPUT” section on page 5–12.
Adding a row to a ProDataGraph

A change to a ProDataGraph can include adding additional rows (ProDataObject instances).

To add a row to a ProDataGraph:

1. Create a ProDataObject for a specific table, using the ProDataGraph createProDataObject() method.
2. Add data to the new ProDataObject using the various ProDataObject methods for setting column properties.
3. Add the row to the specified table (ProDataObject collection) using the ProDataGraph addProDataObject() method.
4. If the ProDataObject participates in any data-relations (ProDataRelationMetaData), generate the required parent-child references to and from the ProDataObject, using the ProDataGraph setChildTableReferences() method.

For more information on adding a row to a ProDataGraph, see the “Adding data to a ProDataGraph” section on page 5–22.

Deleting a row from a ProDataGraph

A change to a ProDataGraph can include deleting rows (ProDataObject instances).

To delete a row from a ProDataGraph, invoke the standard delete() method on the ProDataObject. If the ProDataObject is involved in any data-relations (ProDataRelationMetaData), ABL automatically deletes all parent-child references to and from the ProDataObject.
Passing temp-tables as ProDataGraph parameters

OpenEdge supports passing an individual ABL temp-table parameter as a ProDataGraph, similar to a ProDataSet. This temp-table ProDataGraph must contain a single ProDataObject collection specified by a single ProDataObjectMetaData that maps to the schema of the temp-table. Of course, there is no ProDataRelationMetaData in a temp-table ProDataGraph and it provides no change tracking.

Similar to a ProDataSet, OpenEdge automatically builds the appropriately formed ProDataGraph for a TABLE or TABLE-HANDLE parameter passed as OUTPUT. You must also prepare a ProDataGraph mapped to temp-table parameters passed as INPUT or INPUT-OUTPUT in a manner similar to a ProDataGraph mapped to ProDataSet parameters. In all other ways, programming with a temp-table ProDataGraph is identical to programming with a ProDataSet ProDataGraph that has no data-relations and no change tracking enabled (getBImageFlag() on the ProDataObjectMetaData returns false).

For information on passing a temp-table ProDataGraph as OUTPUT, see the “Passing a ProDataGraph as OUTPUT” section on page 5–12, and for information on passing a temp-table ProDataGraph as INPUT or INPUT-OUTPUT, see the “Passing a ProDataGraph as INPUT or INPUT-OUTPUT” section on page 5–17.

Note: The ProDataGraph is the default mechanism for passing temp-table parameters to a Java Open Client. An alternative mechanism for passing temp-table parameters is available using the SQL ResultSet interface. For a comparison of these mechanisms for passing temp-table parameters, see the sections on passing temp-tables and ProDataSets in Chapter 4, “Passing Parameters.”
OpenEdge provides a Java class, `com.progress.open4gl.ProDataGraph`, that maps to an ABL ProDataSet. The `ProDataGraph` class extends the Java SDO `DataGraph` class which contains the root `DataObject`. The root `DataObject` contains a reference property for each temp-table of the ProDataSet. Thus, the `ProDataGraph` encapsulates the data for all temp-tables in the ProDataSet. A temp-table reference property maps to a `ProDataObject` collection, where an individual `ProDataObject` represents a single row (record) of a temp-table. (See the “`ProDataObject` class” section on page 5–36.)

The `com.progress.open4gl.ProDataGraph` class contains the following constructors and methods.

### Constructors

The `ProDataGraph` class has two constructors:

- To create a `ProDataGraph` from meta data
- To create a `ProDataGraph` from an initial Java SDO `DataGraph`

#### ProDataGraph created from meta data

The following constructor creates a `ProDataGraph` object with the specified meta data:

**Syntax**

```java
ProDataGraph(ProDataGraphMetaData dgmd)
```

*`dgmd`*

Specifies the meta data as a `ProDataGraphMetaData` object, which provides all the information necessary to map the `ProDataGraph` to an ABL ProDataSet. (See the “`ProDataGraphMetaData` class” section on page 5–50.)

#### ProDataGraph created from an initial DataGraph

The following constructor creates a `ProDataGraph` object from a specified initial Java SDO `DataGraph` object:

**Syntax**

```java
ProDataGraph(DataGraph dg, String dgName, ProDataRelationMetaData[] drList)
```

*`dg`*

Specifies the initial Java SDO `DataGraph` object. This `DataGraph` object must conform to the `ProDataGraph` format: the root `DataObject` must contain only reference properties (to `DataObject` lists), where each reference property represents a temp-table of an ABL ProDataSet. If the root `DataObject` does not have this format, the constructor throws an `Open4GLException`. 
ProDataGraph class

\[dgName\]

Specifies the name of the ProDataGraph, which is typically identical to the ABL name of a corresponding ProDataSet.

\[drList\]

Specifies an array of ProDataRelationMetaobject objects. In the DataObject hierarchy of the initial DataGraph, if a parent-child relationship exists between two temp-tables (DataObject lists), you must add a corresponding ProDataRelationMetaobject object that describes this relationship to the ProDataRelationMetaobject array. (See the “ProDataRelationMetaobject class” section on page 5–59.)

**Inherited methods**

The following methods are inherited from the Java SDO DataGraph implementation. For a complete list, see the documentation for the Service Data Objects in EMF 2.0.1 at the following location:

http://eclipse.org/emf/docs.php

**getChangeSummary( )**

Returns the ChangeSummary object associated with this DataGraph. For example:

**Syntax**

```
ChangeSummary getChangeSummary()
```

**getRootObject( )**

Returns the root DataObject of this DataGraph. For example:

**Syntax**

```
DataObject getRootObject()
```

**Extended methods**

OpenEdge provides the following methods to extend the Java SDO DataGraph and implement the ProDataGraph class.

**acceptChanges( )**

Clears out the list of changes in the ProChangeSummary of the ProDataGraph, enabling the ProDataGraph to accept a new set of changes. For example:

**Syntax**

```
void acceptChanges()
```
This method has the same behavior as the `ChangeSummary.beginLogging()` method. For more information, see the “ProChangeSummary class” section on page 5–48.

**addProDataObject()**

Adds the `ProDataObject` instance to its `ProDataObject` collection within the `ProDataGraph`, as specified by the overloaded methods. For example:

**Syntax**

```java
void addProDataObject(ProDataObject dataObj)
void addProDataObject(int index, ProDataObject dataObj)
```

*dataObj*

Specifies the `ProDataObject` to add.

*index*

Specifies the location in the `ProDataObject` collection to add the `ProDataObject`. If you do not specify *index*, the method adds the `ProDataObject` to the end of the collection.

**createProDataObject()**

Returns a new `ProDataObject` instance for the specified temp-table within the `ProDataGraph`. A single `ProDataObject` corresponds to an individual row (record) in the temp-table. For example:

**Syntax**

```java
ProDataObject createProDataObject(String tableName)
```

*tableName*

Specifies the ABL name of a temp-table that identifies the meta data used to create the `ProDataObject`. If there is not a `ProDataObjectMetaData` object contained within this `ProDataGraph` for the specified temp-table name, the method throws an `Exception`. For more information on the `ProDataObjectMetaData` class, see the “ProDataObjectMetaData class” section on page 5–53.

**getMetaData()**

Returns the `ProDataGraphMetaData` object contained by this `ProDataGraph`. For example:

**Syntax**

```java
ProDataGraphMetaData getMetaData()
```

For more information on the `ProDataGraphMetaData` class, see the “ProDataGraphMetaData class” section on page 5–50.
**getNumTables()**

Returns the number of ProDataObject collections (number of temp-tables) in this ProDataGraph object. For example:

**Syntax**

```java
int getNumTables()
```

**getProChangeSummary()**

Returns the ProChangeSummary object associated with this ProDataGraph. For example:

**Syntax**

```java
ProChangeSummary getProChangeSummary()
```

For more information on the ProChangeSummary class, see the “ProChangeSummary class” section on page 5–48.

**getProDataObjects()**

Returns the list of ProDataObject instances for the specified temp-table within the ProDataGraph, as specified by the overloaded methods. For example:

**Syntax**

```java
List getProDataObjects(String tableName)
List getProDataObjects(int tableIdx)
```

**tableName**

Specifies the ABL name of a temp-table that identifies this ProDataObject list.

**tableIdx**

Specifies the 0-based index into the list of temp-table names returned by the `getTableNames()` method and that corresponds to the specified temp-table. (See the “getTableNames()” section on page 5–34.)

For more information on the ProDataObject class, see the “ProDataObject class” section on page 5–36.

**getProDataGraphName()**

Returns the name of this ProDataGraph object. For example:

**Syntax**

```java
java.lang.String getProDataGraphName()
```

This name is typically identical to the ABL name of a ProDataSet to which this ProDataGraph object maps as a parameter of a proxy application service method.
**getTableNames( )**

Returns a `String` array of the table names that identify the `ProDataObject` collections in this `ProDataGraph` object. For example:

**Syntax**

```java
java.lang.String[] getTableNames()
```

Each table name is typically identical to the ABL name of a temp-table contained by a corresponding `ProDataSet` to which this `ProDataGraph` maps as a parameter of a proxy application service method.

**hasError( )**

Returns `true` if the AppServer returned an error message for this `ProDataGraph` passed as an application service parameter. For example:

**Syntax**

```java
boolean hasError()
```

It is set on return from the AppServer when an update to a data source of the corresponding `ProDataSet` parameter is not successful.

**setChildTableReferences( )**

Populates (child) reference property lists for specified `ProDataObject` collections contained in this `ProDataGraph`, as specified by the overloaded methods. For example:

**Syntax**

```java
void setChildTableReferences()
void setChildTableReferences(int tableIdx)
void setChildTableReferences(String tableName)
```

**tableIdx**

Specifies the 0-based index into the list of temp-table names returned by the `getTableNames()` method. (See the “getTableNames( )” section on page 5–34.) The specified temp-table (`ProDataObject` collection) corresponds to the parent table of a data-relation (`ProDataRelationMetaData` object) contained in the `ProDataGraph`.

**tableName**

Specifies the ABL name of a temp-table (`ProDataObject` collection) contained in the `ProDataGraph`. The specified `ProDataObject` collection corresponds to the parent table of a data-relation (`ProDataRelationMetaData` object) contained in the `ProDataGraph`.

If you do not specify `tableIdx` or `tableName`, the method populates the reference property lists for all `ProDataObject` collections involved as parent tables in data-relations contained in the `ProDataGraph`. 
You only need to call this method when building a ProDataGraph object that contains data-relations, typically after adding one or more ProDataObject instances that participate in data-relations contained in the ProDataGraph. A reference property list is a list of references in a ProDataObject that point to other ProDataObject instances contained by a single ProDataObject collection that is a child table of the ProDataObject containing the references. This parent ProDataObject contains one reference property list for each ProDataRelationMetaData that the ProDataObject is involved in as a parent. For more information on the ProDataRelationMetaData class, see the “ProDataRelationMetaData class” section on page 5–59.
ProDataObject class

OpenEdge provides a Java class, `com.progress.open4gl.ProDataObject`, that maps to a row (record) in an ABL temp-table. The ProDataObject class extends the Java SDO `DataObject` class. It maps to an ABL temp-table row within a ProDataSet. (A `ProDataObject` list maps to the entire ABL temp-table). A `ProDataObject` holds its actual data as a set of column properties, where each `Property` object maps to a column (field) of the temp-table. Java SDO `DataObject` and extended `ProDataObject` access methods allow the client to get the `Property` object values (column data).

In ABL, fields in a temp-table can be defined with an extent value that creates a one-dimensional array of the specified data type, with the number of elements specified by the extent. JDBC `ResultSet` objects (also supported to map temp-table parameters) do not support arrays as columns for temp-table parameters. But in a Java SDO `DataObject`, a property's type (specified by a `Type` object) can be many-valued and this can be mapped to an array field.

For more information on the Java SDO `Property` and `Type` interfaces, see the “Using Java SDO classes to access Property meta data” section on page 5–63.

The `com.progress.open4gl.ProDataObject` class provides no constructor, because it is created by the `ProDataGraph.createProDataObject()` factory method (see the “ProDataGraph class” section on page 5–30). It does provide the following inherited and extended methods. Among these include methods to get and set the values of column properties according to the Java data type. To identify the Java data type that matches the ABL data type of a corresponding temp-table field, see the data type mappings in Table 5–3. If you do not already know the ABL data type of a mapped column property, you can identify it using the `getProType()` method of the corresponding `ProDataObjectMetaData` object. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

Inherited methods

The following methods are inherited from the Java SDO `DataObject` implementation. For a complete list, see the documentation for the Service Data Objects in EMF 2.0.1 at the following location:

http://eclipse.org/emf/docs.php

get()

Returns the specified column property value as an `Object`, as specified by the overloaded methods. For example:

Syntax

```java
java.lang.Object get(int propertyIndex)
java.lang.Object get(String propertyName)
```

`propertyIndex`

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMetaData` methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.
**proDataObject class**

`propertyName`

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**getBigDecimal()**

Returns the value of a `BigDecimal` column property, as specified by the overloaded methods. For example:

**Syntax**

```java
java.math.BigDecimal getBigDecimal(int propertyIndex)
java.math.BigDecimal getBigDecimal(String propertyName)
```

`propertyIndex`

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMeta`Data methods. For more information, see the “`ProDataObjectMeta`Data class” section on page 5–53.

`propertyName`

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**getBoolean()**

Returns the value of a `boolean` column property, as specified by the overloaded methods. For example:

**Syntax**

```java
boolean getBoolean(int propertyIndex)
boolean getBoolean(String propertyName)
```

`propertyIndex`

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMeta`Data methods. For more information, see the “`ProDataObjectMeta`Data class” section on page 5–53.

`propertyName`

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.
getBytes( )

Returns the value of a byte[] column property, as specified by the overloaded methods. For example:

Syntax

<table>
<thead>
<tr>
<th>byte[] getBytes(int propertyIndex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[] getBytes(String propertyName)</td>
</tr>
</tbody>
</table>

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

getInt( )

Returns the value of an int column property, as specified by the overloaded methods. For example:

Syntax

<table>
<thead>
<tr>
<th>int getInt(int propertyIndex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getInt(String propertyName)</td>
</tr>
</tbody>
</table>

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.
ProDataObject class

getList( )

Returns the value of a List column property, as specified by the overloaded methods. For example:

**Syntax**

```
java.util.List getList(int propertyIndex)
java.util.List getList(String propertyName)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

You can use this method to access the many-valued column property mapped to a temp-table array field. To identify the Java data type of each Property in the list, match the ABL data type of the corresponding temp-table field to the Java data type specified in Table 5–3.

getLong( )

Returns the value of a long column property, as specified by the overloaded methods. For example:

**Syntax**

```
long getLong(String propertyName)
long getLong(int propertyIndex)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.
getString( )

Returns the value of a String column property, as specified by the overloaded methods. For example:

**Syntax**

```
String getString(int propertyIndex)
String getString(String propertyName)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

getype( )

Returns the type of the DataObject. For example:

**Syntax**

```
commonj.sdo.Type getType()
```

For more information on DataObject types, see the “Using Java SDO classes to access Property meta data” section on page 5–63.

isSet( )

Returns true if the column property, as specified by the method overloading, is considered to be set. For example:

**Syntax**

```
boolean isSet(int propertyIndex)
boolean isSet(String propertyName)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.
set( )

Sets the value of the column property as an `Object`, as specified by the overloaded methods. For example:

**Syntax**

```java
void set(int propertyIndex, java.lang.Object value)
void set(String propertyName, java.lang.Object value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMetaData` methods. For more information, see the “`ProDataObjectMetaData` class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.

**setBigDecimal( )**

Sets the value of a `BigDecimal` column property, as specified by the overloaded methods. For example:

**Syntax**

```java
void setBigDecimal(int propertyIndex, java.math.BigDecimal value)
void setBigDecimal(String propertyName, java.math.BigDecimal value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMetaData` methods. For more information, see the “`ProDataObjectMetaData` class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.
setBoolean()

Sets the value of a boolean column property, as specified by the overloaded methods. For example:

**Syntax**

```java
void setBoolean(int propertyIndex, boolean value)
void setBoolean(String propertyName, boolean value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMetaData` methods. For more information, see the “`ProDataObjectMetaData class`” section on page 5–53.

**propertyName**

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.

setBytes()

Sets the value of a `byte[]` column property, as specified by the overloaded methods. For example:

**Syntax**

```java
void setBytes(int propertyIndex, byte[] value)
void setBytes(String propertyName, byte[] value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the `ProDataObject`. You can identify the index of a particular column property that corresponds to a mapped temp-table field using `ProDataObjectMetaData` methods. For more information, see the “`ProDataObjectMetaData class`” section on page 5–53.

**propertyName**

Specifies the name of a column property in the `Property` list contained by the `ProDataObject`. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

 Specifies the value to set for this property.
setInt()

Sets the value of an int column property, as specified by the overloaded methods. For example:

Syntax

```java
void setInt(int propertyIndex, int value)
void setInt(String propertyName, int value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.

setList()

Sets the value of a List column property, as specified by the overloaded methods. For example:

Syntax

```java
void setList(int propertyIndex, java.util.List value)
void setList(String propertyName, java.util.List value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.

You can use this method to set the value of a many-valued column property mapped to a temp-table array field. To identify the Java data type of each Property in the list, match the ABL data type of the corresponding temp-table field to the Java data type specified in Table 5–3.
setLong( )

Sets the value of a long column property, as specified by the overloaded methods. For example:

Syntax

```
void setLong(int propertyIndex, long value)
void setLong(String propertyName, long value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.

setString( )

Sets the value of a String column property, as specified by the overloaded methods. For example:

Syntax

```
void setString(int propertyIndex, String value)
void setString(String propertyName, String value)
```

Extended methods

OpenEdge provides the following methods to extend the Java SDO dataObject and implement the ProDataObject class.

**getChildRows( )**

Returns a list of the specified child ProDataObject instances of the current parent ProDataObject. For example:

Syntax

```
java.util.List getChildRows(java.lang.String relationName)
```

**relationName**

Specifies the name of a data-relation (ProDataRelationMetaData) in which the current ProDataObject participates as a parent row. For more information, see the “ProDataRelationMetaData class” section on page 5–59.
**getFieldCount()**

Returns the number of column properties that this ProDataObject represents. For example:

**Syntax**

```java
int getFieldCount()
```

A ProDataObject holds its data as a property list—one property per column. However, a ProDataObject might contain extra properties (reference properties) if it is involved in a data-relation (ProDataRelationMetaData). This method returns the number of properties that map to a column. You can access these properties in sequence using the various ProDataObject set and get methods with the 0-based `propertyIndex` value up to the limit specified by this method.

**getGregorianCalendar()**

Returns the value of a GregorianCalendar column property, as specified by the following overloaded methods:

**Syntax**

```java
java.util.GregorianCalendar getGregorianCalendar(int propertyIndex)
java.util.GregorianCalendar getGregorianCalendar(String propertyName)
```

**Note:** The Java SDO API does not currently support GregorianCalendar for a DataObject. However, ABL supports this data type for use with the ProDataObject class.

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**getParentRow()**

Returns the parent ProDataObject of the current child ProDataObject. For example:

**Syntax**

```java
ProDataObject getParentRow(java.lang.String relationName)
```

**relationName**

Specifies the name of a data-relation (ProDataRelationMetaData) in which the current ProDataObject participates as a child row. For more information, see the “ProDataRelationMetaData class” section on page 5–59.
getRowErrorString()  
Returns the error string for this ProDataObject instance (row). For example:

**Syntax**

```java
String getRowErrorString()
```

This error string is set on return from the AppServer when the update to a data source using this row (mapped to a record in a temp-table) is not successful.

getTableErrorString()  
Returns the error string for the temp-table (ProDataObject collection) that contains the data from this ProDataObject. For example:

**Syntax**

```java
String getTableErrorString()
```

This error string is set on return from the AppServer when the update to a data source using this temp-table is not successful.

gTableName()  
Returns the ABL name of the temp-table that this ProDataObject maps to. For example:

**Syntax**

```java
java.lang.String getTableName()
```

hasRowError()  
Returns true if the AppServer returned an error message for the temp-table record (row) mapped to this ProDataObject. For example:

**Syntax**

```java
boolean hasRowError()
```

It is set on return from the AppServer when an update to a data source using this row is not successful.

hasTableError()  
Returns true if the AppServer returned an error message for the temp-table that contains the data from this ProDataObject row. For example:

**Syntax**

```java
boolean hasTableError()
```

It is set on return from the AppServer when an update to a data source using any row in this temp-table is not successful.
setGregorianCalendar()

Sets the value of a GregorianCalendar column property, as specified by the overloaded methods:

**Syntax**

```java
void setGregorianCalendar (int propertyIndex,
                        java.util.GregorianCalendar value)
void setGregorianCalendar (String propertyName, java.util.GregorianCalendar value)
```

**propertyIndex**

Specifies a 0-based index into the list of column properties contained by the ProDataObject. You can identify the index of a particular column property that corresponds to a mapped temp-table field using ProDataObjectMetaData methods. For more information, see the “ProDataObjectMetaData class” section on page 5–53.

**propertyName**

Specifies the name of a column property in the Property list contained by the ProDataObject. This name is typically identical to the name of an ABL temp-table field to which this column property is mapped.

**value**

Specifies the value to set for this property.

**Note:** The Java SDO API does not currently support GregorianCalendar for a DataObject. However, ABL supports this data type for use with the ProDataObject class.
Accessing ABL ProDataSets

ProChangeSummary class

OpenEdge provides a Java class, `com.progress.open4gl.ProChangeSummary`, that tracks changes to a ProDataGraph. The ProChangeSummary class extends the Java SDO ChangeSummary class, which is contained within a DataGraph. It records changes to any of the underlying DataObject instances in a DataGraph, including modifications, deletions, and additions of DataObject instances from given starting state. At a later point in the update cycle of a ProDataGraph, when you invoke the ProDataGraph.acceptChanges() method, the contained ProChangeSummary object resets to begin tracking changes from a new starting state of the ProDataGraph.

The `com.progress.open4gl.ProChangeSummary` class provides no constructor, because it is internally managed entirely by the ProDataGraph that contains it. It does provide the following inherited and extended methods.

Inherited methods

The following methods are inherited from the Java SDO ChangeSummary implementation. For a complete list, see the documentation for the Service Data Objects in EMF 2.0.1 at the following location:


**beginLogging( )**

Clears the list of changes and turns change logging on. For example:

**Syntax**

```java
void beginLogging()
```

This method has the same behavior as the ProDataGraph.acceptChanges() method. For more information, see the “ProDataGraph class” section on page 5–30.

**endLogging( )**

Turns off change logging. For example:

**Syntax**

```java
void endLogging()
```

**getChangedDataObjects( )**

Returns a list consisting of all the DataObject instances that have changed while logging. For example:

**Syntax**

```java
java.util.List getChangedDataObjects()
```
getOldValues()  
Returns a list of settings that represents the property values of the specified DataObject at the point when logging began. For example:

Syntax

```java
java.util.List getOldValues(DataObject dataObj)
```

* dataObj  
  Specifies the DataObject instance whose old values you want to get.

isCreated()  
This method returns true if the specified DataObject was created while logging. For example:

Syntax

```java
boolean isCreated(DataObject dataObj)
```

* dataObj  
  Specifies a DataObject instance whose update status you want to test.

isDeleted()  
Returns true if given DataObject was deleted while logging. For example:

Syntax

```java
boolean isDeleted(DataObject dataObj)
```

* dataObj  
  Specifies a DataObject instance whose update status you want to test.

Extended method

OpenEdge provides the following method to extend the Java SDO ChangeSummary and implement the ProChangeSummary class.

getChanges()  
Returns a ProDataGraph consisting of all the ProDataObject instances that have been changed while logging. For example:

Syntax

```java
ProDataGraph getChanges()
```
OpenEdge provides a Java class, `com.progress.open4gl.ProDataGraphMetaData`, that allows you to specify and access the schema (meta data) for a ProDataGraph object. Before you can instantiate a ProDataGraph object, you must first define its meta data. You can derive the meta data from an existing Java SDO DataGraph or specify the meta data directly using a ProDataGraphMetaData object. For more information on instantiating ProDataGraph objects, see the “ProDataGraph class” section on page 5–30.

The ProDataGraphMetaData must contain one or more ProDataObjectMetaData objects to specify the meta data for one or more temp-tables (ProDataObject collections) defined for the ProDataGraph (see the “ProDataObjectMetaData class” section on page 5–53).

If the ProDataGraph contains more than one ProDataObjectMetaData object, the ProDataGraphMetaData can also contain zero or more ProDataRelationMetaData objects. A ProDataRelationMetaData object defines a relationship between a parent and child ProDataObject collection (see the “ProDataRelationMetaData class” section on page 5–59).

The `com.progress.open4g1.ProDataGraphMetaData` class contains the following constructor and methods.

### Constructor

The ProDataGraphMetaData class has the following one constructor:

**Syntax**

```java
ProDataGraphMetaData(String dataSetName)
```

*dataSetName*

Specifies the name of the specified ProDataGraph. Typically, this is identical to the ABL name of a ProDataSet to which the specified ProDataGraph maps as an application service parameter.

### Methods

The ProDataGraphMetaData class provides the methods that follow.

**addDataRelation()**

Adds the specified data-relation definition to the ProDataGraphMetaData object. For example:

**Syntax**

```java
void addDataRelation (ProDataRelationMetaData drmd)
```

*drmd*

Specifies a ProDataRelationMetaData object that defines a data-relation for a ProDataGraph.
You can only call this method after you have already called the addTable() method to define the ProDataObject collections that participate in the specified data-relation. You must also call this method to define each required data-relation before attaching this ProDataGraphMetaData object to the ProDataGraph. This method throws an Exception if any table participating in the specified data-relation is not already a member of the ProDataGraphMetaData.

**addTable()**

Adds the specified temp-table schema definition (ProDataObject meta data) to the ProDataGraphMetaData object. For example:

**Syntax**

```java
void addTable(ProDataObjectMetaData tablemd)
```

**tablemd**

Specifies a ProDataObjectMetaData object that defines the schema of a temp-table in a ProDataGraph.

You must call this method to define each required ProDataObject collection before attaching this ProDataGraphMetaData object to the ProDataGraph. Each ProDataObjectMetaData object describes the meta data for ProDataObject instances in a collection that maps to a temp-table in the corresponding ABL ProDataSet. When you call ProDataGraph.createProDataObject(), ABL returns a ProDataObject that contains column and type data as described by the corresponding ProDataObjectMetaData object.

**getNumRelations()**

Returns the number of ProDataRelationMetaData objects in this ProDataGraphMetaData object. For example:

**Syntax**

```java
int getNumRelations()
```

**getNumTables()**

Returns the number of ProDataObjectMetaData objects defined in this ProDataGraphMetaData object. For example:

**Syntax**

```java
int getNumTables()
```
getRelationMetaData()

Returns the specified ProDataRelationMetaData object in this ProDataGraphMetaData object. For example:

Syntax

```java
ProDataRelationMetaData getRelationMetaData(int idx)
```

**idx**

Specifies a 0-based index of data-relations up to the limit specified by the `getNumRelations()` method.

getTableMetaData()

Returns the specified ProDataObjectMetaData object in this ProDataGraphMetaData object. For example:

Syntax

```java
ProDataObjectMetaData getTableMetaData(int idx)
```

**idx**

Specifies a 0-based index of tables that can also reference an item in the array of table names returned by the `getTableNames()` method up to the limit specified by the `getNumTables()` method.

getTableNames()

Returns a String array of the table names of the ProDataObjectMetaData objects defined in this ProDataGraphMetaData object. For example:

Syntax

```java
java.lang.String[] getTableNames()
```
ProDataObjectMetaData class

OpenEdge provides a Java class, com.progress.open4gl.ProDataObjectMetaData, that allows you to specify and access the schema (meta data) for a ProDataObject collection in terms of a corresponding ABL temp-table. Before you can instantiate a ProDataObject for any collection, you must define its schema (meta data or type) using this class. For more information on instantiating ProDataObject instances, see the “ProDataObject class” section on page 5–36.

A ProDataObject has a commonj.sdo.Type object associated with it that defines the schema (data model) for all ProDataObject instances of that type. When you define a ProDataObjectMetaData, OpenEdge builds this Type object to reflect the specified data model. Once the definition of the ProDataObjectMetaData is complete, you must add it to the ProDataGraphMetaData used to instantiate a ProDataGraph so you can then create ProDataObject instances with the corresponding type information. For more information on the commonj.sdo.Type interface, see the “Using Java SDO classes to access Property meta data” section on page 5–63.

The com.progress.open4gl.ProDataObjectMetaData class contains the following constructor and methods.

Constructor

The ProDataObjectMetaData class has the following one constructor:

Syntax

ProDataObjectMetaData(String tableName, int numFields, boolean bimageFlag,
int numIndexes, String multiIxCols,
String XMLNamespace, String XMLPrefix)

tableName

Specifies a name for the specified ProDataObject type (and collection). This name is typically identical to any ABL temp-table to which this collection is mapped.

numFields

Specifies the number of fields (column properties) in the specified ProDataObject type.

bimageFlag

Specifies true if the corresponding ABL temp-table is defined with the BEFORE-TABLE option, indicating that the temp-table (and hence the ProDataObject collection) can be modified. Otherwise, this value must be false and you cannot modify the data contained in the collection.

numIndexes

Specifies the number of indexes on the table.
multiIxCols

Specifies null if there are no indexes or a formatted string that contains all the index info for this temp-table, as follows:

**Syntax**

```
"[primeUniqueFlag,primeFld1[,primeFldn]].\ldots:primeIdxName.]
[uniqueIdxfld1[,uniqueIdxfldn]].\ldots:uniqueIdxName.].\ldots"
```

**primeUniqueFlag**

Specifies a primary index with a value of 1 if the index is unique and a value of 0 if the index is not unique.

**primeFld1[,primeFldn].\ldots**

Specifies the names of one or more fields involved in the primary index.

**primeIdxName**

Specifies the primary index name.

**uniqueIdxfld1[,uniqueIdxfldn].\ldots**

Specifies names of one or more fields involved in a unique secondary index.

**uniqueIdxName**

Specifies a unique secondary index name.

Thus, a table can have no indexes, a single primary index followed by zero or more secondary unique indexes, or it can have a single secondary unique index followed by zero or more additional secondary unique indexes. Examples of these tables follow:

- **Table with a single primary index:**

  "1,custNum:CustIndex"

  This is a primary unique index named CustIndex with one column named custNum.

- **Table with a single secondary index:**

  "orderDate:OrderIndex"

  This is a secondary unique index named OrderIndex with one column named orderDate.
• Table with two indexes:

```
"1,custNum:CustIndex.orderDate,shipDate:OrderIndex"
```

The two indexes are:

- A primary unique index named CustIndex with one column named custNum.
- A secondary unique index named OrderIndex with two columns named orderDate and shipDate

• Table with three indexes:

```
"0,custNum,custName:CustIndex.orderDate:OrderIndex.
   itemNum:ItemIndex"
```

The three indexes are:

- A primary non-unique index named CustIndex with two columns named custNum and custName
- A secondary unique index named OrderIndex with one column named orderDate
- A secondary unique index named ItemIndex with one column named itemNum

**XMLNamespace**

Specifies the namespace for XML serialization or `null`.

**XMLPrefix**

Specifies the prefix for XML serialization or `null`.

**Methods**

The `ProDataObjectMetaData` class provides the following methods.

**getBImageFlag()**

Returns the `bImageFlag` value defined for this `ProDataObjectMetaData` object (corresponding to the `BEFORE-TABLE` attribute of the corresponding ABL temp-table). For example:

**Syntax**

```
boolean getBImageFlag()
```
getExtent()

Returns the extent value of the specified column property. For example:

Syntax

```java
int getExtent(int propertyIndex)
```

propertyIndex

Specifies a 0-based index into the ProDataObject property list, limited by the value of the getFieldCount() method. It returns a value that corresponds to the relative position specified by FieldNumber in the setFieldMetaData() method.

If the column property does not represent an array field, the method always returns 0.

getFieldCount()

Returns the number of column properties defined for the specified ProDataObject type. For example:

Syntax

```java
int getFieldCount()
```

The value is identical to the setting of numFields in the constructor that instantiates this ProDataObjectMetaData.

getFieldName()

Returns the name of the specified column property. For example:

Syntax

```java
int getFieldName(int propertyIndex)
```

propertyIndex

Specifies a 0-based index into the ProDataObject property list, limited by the value of the getFieldCount() method. It returns a value that corresponds to the relative position specified by FieldNumber in the setFieldMetaData() method.

getNoSchemaMarshal()

Returns true if the most recent invocation of the setNoSchemaMarshal() method indicated that the specified ProDataObject collection is to be marshalled to the AppServer without schema information. For example:

Syntax

```java
boolean getNoSchemaMarshal()
```
**getProType()**

Returns the ABL data type of the specified column property as a class constant value found in `com.progress.open4gl.Parameter`. For example:

**Syntax**

```java
int getProType(int propertyIndex)
```

**propertyIndex**

Specifies a 0-based index into the `ProDataObject` property list, limited by the value of the `getFieldCount()` method. It returns a value that corresponds to the relative position specified by `fieldNumber` in the `setFieldMetaData()` method.

For more information on the class constant values provided by the `com.progress.open4gl.Parameter` class, see the information on specifying data type metadata for temp-tables in Chapter 4, “Passing Parameters.”

**getTableName()**

Returns the table name associated with this `ProDataObjectMetaData` object. For example:

**Syntax**

```java
String getTableName()
```

**getUserOrder()**

Returns the user order position of the specified column property. For example:

**Syntax**

```java
int getUserOrder(int propertyIndex)
```

**propertyIndex**

Specifies a 0-based index into the `ProDataObject` property list, limited by the value of the `getFieldCount()` method. It returns a value that corresponds to the relative position specified by `fieldNumber` in the `setFieldMetaData()` method.

**setFieldMetaData()**

Adds the specified field (column property) to the property list of the specified `ProDataObject` type. For example:

**Syntax**

```java
void setFieldMetaData(int fieldNumber, String fieldName, int extent, int proType, int userOrder, int xmlMapping)
```

**fieldNumber**

Specifies a 1-based position for this column property that corresponds to the position of a mapped field in an ABL temp-table.
**fieldName**

Specifies a name for the column property that is typically identical to a mapped field in the corresponding ABL temp-table. The value cannot be null and must be unique among column properties in the specified ProDataObject type.

**extent**

Specifies if and how the column property represents an array field in the corresponding temp-table. The value must be 0 or greater. If the value is greater than 1, this column property is many-valued (represents an array field) and the value is its extent. If the property represents a BLOB or CLOB field, the value must be 0 or 1.

**proType**

Specifies the value of a class constant defined in the `com.progress.open4gl.Parameter` class. The specified class constant indicates the ABL data type of the mapped temp-table field. For more information on these class constants, see the information on specifying data type meta data for temp-tables in Chapter 4, “Passing Parameters.” To identify the Java data type that the column property assumes for the specified ABL data type, see Table 5–3.

**userOrder**

Specifies a 0-based user order position for the column property.

**xmlMapping**

Reserved for future use. Always specify 0.

For more information on column properties, see the “Using Java SDO classes to access Property meta data” section on page 5–63.

**setNoSchemaMarshal( )**

Indicates, as a run-time only setting, whether schema information for the specified ProDataObject collection is sent to the AppServer along with the data. For example:

**Syntax**

```java
void setNoSchemaMarshal(boolean flag)
```

**flag**

When set to true, any application service method that passes a ProDataGraph parameter containing this ProDataObjectMetaData marshals only the data to the AppServer from the specified ProDataObject collection. This setting suppresses transmission of index descriptions and all field information, which helps to speed the transmission of data. You can change this setting at any point where you invoke application service methods.
ProDataRelationMetaData class

OpenEdge provides a Java class, com.progress.open4gl.ProDataRelationMetaData, that allows you to specify and access the definition of a given data-relation. A single data-relation defines a relationship between a parent ProDataObject collection and a child ProDataObject collection within a ProDataGraph. Thus, a ProDataRelationMetaData object represents schema information that is defined as part of a corresponding ProDataGraphMetaData object. To specify parent-child relationships, each data-relation names the column properties (fields) in parent and child ProDataObject types that form a primary-foreign key relationship between a given parent ProDataObject instance and one or more child ProDataObject instances.

You can use a data-relation to retrieve child records related to the parent. A parent ProDataObject instance (representing a single table row) contains a list of references to the related child ProDataObject instances (representing one or more table rows). For each data-relation, these references are represented in every parent row as a many-valued reference property in order to support multiple child rows. After you instantiate a ProDataGraph and add all ProDataObject instances, you can then call the ProDataGraph setChildTableReferences() method, which uses the specified key relationships to create all the child reference property lists in each parent ProDataObject as provided by the data-relations defined for the ProDataGraph.

A ProDataGraphMetaData object can have multiple ProDataRelationMetaData objects. Each ProDataRelationMetaData object typically maps to a particular data-relation object in a corresponding ABL ProDataSet.

Note: The requirement to map data-relations in a ProDataGraph to data-relation objects in a ProDataSet depends on whether the ProDataSet parameter is static or dynamic and what the application service expects. For more information, see the information on preparing to pass parameters using different modes in the “Preparing and passing ProDataSets as ProDataGraph parameters” section on page 5–9.

The com.progress.open4gl.ProDataRelationMetaData class contains with the following constructors and methods.

Constructors

The ProDataRelationMetaData class has two constructors for:

- Creating a data-relation from parent and child ProDataObjectMetaData
- Creating a data-relation from selected parent and child key fields

Creating a data-relation from parent and child ProDataObjectMetaData

The following constructor creates a ProDataRelationMetaData object with the specified ProDataObjectMetaData objects:

Syntax

ProDataRelationMetaData(String name, ProDataObjectMetaData parent, ProDataObjectMetaData child)

name

Specifies the name of the ProDataRelationMetaData object.
parent

Specifies the parent ProDataObject type.

child

Specifies the child ProDataObject type.

After creating a ProDataRelationMetaData object using this constructor, use the object’s setColumns() methods to set the key fields for the parent and child table relationships. For more information, see the “setColumns( )” section on page 5–62.

Creating a data-relation from selected parent and child key fields

The following constructor creates a ProDataRelationMetaData object from column properties selected as key fields in specified parent and child tables (ProDataObject collections):

Syntax

```java
ProDataRelationMetaData(String name, int parentIx, int childIx, int numPairs, String pairsList, )
```

name

Specifies the name of the ProDataRelationMetaData object.

parentIx

Specifies a 0-based index to a parent ProDataObject collection that corresponds to an index into the array of table names returned by the ProDataGraphMetaData getTableNames( ) method.

childIx

Specifies a 0-based index to a child ProDataObject collection that corresponds to an index into the array of table names returned by the ProDataGraphMetaData getTableNames( ) method.

numPairs

Specifies the number of column property pairs (key field pairs) that represent this relationship.

pairsList

Specifies a String containing a comma-separated list of column property names. The list consists of numPairs column pairs, where the parent’s column property name is followed by its matching child column property name. For example:

- If numPairs = 1, pairsList might contain: "pCustName,cOrderCustName" as one pair of parent and child property names
- If numPairs = 2, pairsList might contain: "pCustName,cOrderCustName,pCustBranch,cOrderBranch" as two pair of parent and child property names

The data types of the named parent and child column property pairs must be comparable.
Methods

The `ProDataRelationMetaData` class provides the following methods.

**getChildColumns( )**

Returns the 0-based indexes of the child table column properties for the following `ProDataRelationMetaData` object:

**Syntax**

```
int[] getChildColumns()
```

**getChildTable( )**

Returns the name of the child table for this `ProDataRelationMetaData` object. For example:

**Syntax**

```
String getChildTable()
```

**getParentColumns( )**

Returns the 0-based indexes of the parent table column properties for this `ProDataRelationMetaData` object. For example:

**Syntax**

```
int[] getParentColumns()
```

**getParentTable( )**

Returns the name of the parent table for this `ProDataRelationMetaData` object. For example:

**Syntax**

```
String getParentTable()
```

**getRelationName( )**

Returns the name of this `ProDataRelationMetaData` object. For example:

**Syntax**

```
String getRelationName()
```
Accessing ABL ProDataSets

setColumns()  
Specifies the column properties that are to participate in primary-foreign key relationships between the parent and child tables of this ProDataRelationMetaDataSet object, as specified by the overloaded methods. For example:

Syntax

```java
void setColumns(int parentColIdx, int childColIdx)
void setColumns(int[] parentColIdxs, int[] childColIdxs)
```

parentColIdx

Specifies the 0-based index of a single column property in the parent ProDataObject type to use as the primary key.

childColIdx

Specifies the 0-based index of a single column property in the child ProDataObject type to use as the foreign key.

parentColIdxs

Specifies an array of 0-based indexes to column properties in the parent ProDataObject type to use as the primary key.

childColIdxs

Specifies an array of 0-based indexes to column properties in the child ProDataObject type to use as the foreign key.

When specifying parentColIdxs and childColIdxs, the arrays must be the same length. For either overloading, the corresponding parent and child column properties must be comparable.
Using Java SDO classes to access Property meta data

The OpenEdge ProDataGraph relies on the Java SDO Type and Property interfaces to encapsulate the base DataObject interface meta data for the columns of a ProDataObject. This meta data specifies both the list of Property objects (properties) that define a DataObject and specifies the Java data types that define the individual column properties and whether the properties are single- or many-valued. For a ProDataObject, the ProDataObjectMetaData then maps the schema information for temp-table fields to the corresponding Property objects.

This section describes the Type and Property interfaces in terms of base DataObject meta data support with reference to the OpenEdge ProDataGraph, as appropriate.

Type interface

The Java SDO Type interface represents a common view of a data type. A Type has a Class object associated with it. For example, a Type representing a simple data type might have the java.lang.Integer class associated with it. A Type representing a complex data type might have a Customer class associated with it. Thus, java.lang.Class implements the Java SDO Type interface.

So, both a DataObject and a Property object has a Type associated with it. Typically, the Type of a DataObject is a complex data type like the Customer class, and the type of a Property is a simple data type like the java.lang.Integer class. A Type that represents a complex data type also has a Property list associated with it. So, for example, a Type representing the Customer class has a list of Property objects corresponding to the Customer columns or fields. A Type that corresponds to a simple data type defines no Property lists.

In the ProDataGraph model, a temp-table defined by a ProDataObjectMetaData object has a Type with a Property list that includes column properties, which map to the temp-table fields, and reference properties, which implement any data-relations specified by a ProDataRelationMetaData object. Each of the properties in this list also has a Type. So, both the DataObject (or ProDataObject for a ProDataGraph) and its list of Property objects have a getType() method that returns the Type of the respective object.

Methods

The Java SDO Type interface provides the following methods:

- **getClass()**, getInstanceClass() — These methods return the Java class that this Type represents. If this is the Type of a reference property, getInstanceClass() returns null. For example:

  Syntax

  ```
  java.lang.Class getClass()
  java.lang.Class getInstanceClass()
  ```

- **getProperties()** — This method returns the list of the properties for the Type of a DataObject. For a Type representing the simple data type of a Property (for example, java.lang.Integer), it returns null. For example:

  Syntax

  ```
  java.util.List getProperties()
  ```
• **getProperty()** — This method returns the property with the specified name from the property list of a DataObject Type. For a column property of a ProDataObject, `propName` is also the ABL name of a corresponding temp-table field. For example:

```java
Property getProperty(String propName)
```

• **isInstance()** — This method returns whether the specified object is an instance of this Type. For example, if Type represents the Customer class and `obj` is a DataObject whose Type represents the Customer class, this method returns true. For example:

```java
boolean isInstance(java.lang.Object obj)
```

### Property interface

A Java SDO Property is contained in a property list associated with a DataObject, where each property in the list has a well defined index. In the ProDataGraph model, a Property corresponds to either a temp-table field (as a column property) or references another DataObject (as a reference property). A column property has a Type that is either a primitive type, such as int, or a commonly used class, such as java.lang.String. A reference property has a Type that refers to the Type of another DataObject. For example, the reference property of a DataObject of data type Customer might refer to one or more DataObject instances of data type Order.

A column property can be single-valued or many-valued. If an ABL temp-table field is defined with an extent option, its corresponding column property is defined as many-valued.

In the ProDataGraph model, a reference property is used to implement a data-relation defined by an associated ProDataRelationMetaData object. When you invoke the ProDataGraph `setChildReferences()` method, a reference property is added to each DataObject of every parent table that references a child table DataObject. A reference property is defined as many-valued so multiple child DataObject instances of the same type can be referenced in a given child table.

### Methods

The Java SDO Property interface provides the following methods:

• **getDefault()** — This method returns the default value this Property has in a DataObject (as specified by Java SDO) when the property has not yet been set. For example:

```java
Object getDefault()
```
• getName() — This method returns the name of the Property. For a column property, this is the ABL name of the corresponding temp-table field. For example:

Syntax

```java
String getName()
```

• getType() — This method returns the Type of the Property. For example:

Syntax

```java
Type getType()
```

• isMany() — This method returns true if this Property is defined as many-valued, and false if single-valued. If it returns true for a column property, the DataObject get*() methods return a java.lang.List. It always returns true for a reference property. For example:

Syntax

```java
boolean isMany()
```
Sample application

The Documentation and Samples directory contains a sample Java Open Client application that accesses and updates an ABL ProDataSet as a ProDataGraph. You can find it in the following directory:

```
Doc_and_Samples_Install/src/samples/open4gl/java/UpdateDataSet/
```

For more information on accessing and installing these samples, see the information on sample Java client applications in Chapter 1, “Configuring and Deploying Java Open Client Applications.”
Extending Proxy Objects

Extending proxy objects is a step you may need to do. This chapter provides an example to demonstrate the technique in the following section:

- Example of extending proxy objects
Example of extending proxy objects

For the Java client, ProxyGen generates two class files for each object. One contains the implementation, and the other is a delegating class that just calls this implementation class. The Java client code accesses the delegating classes. The delegating classes are created so the Java client is not exposed to implementation details of the proxy. These delegating classes are available for inheritance in Java while the implementation classes are final and cannot be extended.

Caution: Direct editing of any delegating class generated by ProxyGen is not supported. To modify a delegating class, you must extend (subclass) it.

For example, if we have an Account AppObject and a Tax SubAppObject, ProxyGen generates the implementation classes AccountImpl.java and TaxImpl.java and the delegating classes Account.java and Tax.java.

Example 6–1 shows a skeleton of the Account delegating class.

Example 6–1: Delegating class

```java
public class Account {
    protected AccountImpl m_accountImpl;
    
    public Account(connect-parameters) {
        m_accountImpl = new AccountImpl(connect-parameters);
    }
    
    public void Add(add-parameters) {
        m_accountImpl.Add(add-parameters);
    }
    
    public Tax createAO_Tax() {
        return new Tax(m_accountImpl);
    }
    
    public AccountInfo createPO_AccountInfo(AccountInfo-parameters) {
        return new AccountInfo(m_accountImpl, AccountInfo-parameters);
    }
    
    public _release() {
        m_accountImpl._release();
    }
    ...
}
```

Note the protected member variable m_accountImpl. If a SubAppObject or a ProcObject is created, this variable must be passed to its constructor.
Example 6–2 shows the constructor in the Tax class.

**Example 6–2: SubAppObject/ProcObject constructor**

```java
Public class Tax
{
    protected TaxImpl m_taxImpl;

    public Tax(AccountImpl accountImpl)
    {
        m_taxImpl = new TaxImpl(accountImpl);
    }
}
```

The member variables are protected rather than private, to allow a Java client to extend these classes.

Example 6–3 shows how a client might extend the Account and Tax classes.

**Example 6–3: Class extensions**

```java
Public class MyAccount extends Account
{
    public MyAccount(parameters)
    {
        super(parameters);
    }
    public MyTax createAO_MyTax()
    {
        return new MyTax(m_accountImpl);
    }
    public sendMail() // a new method
    {
        /*** Do my own thing ***/
    }
}

Public class MyTax extends Tax
{
    public MyTax(AccountImpl accountImpl)
    {
        super(accountImpl);
    }
    public SetStatus(int status) // override Tax's method
    {
        /*** Do my own thing and then defer to superclass ***/
        super.SetStatus(status);
    }
}
```

**Note:** To extend a SubAppObject or ProcObject, you also must extend the associated AppObject.
Accessing Proxy Properties

The Open Client Runtime provides several properties that determine the behavior of the client application. These properties are called *proxy properties*. You can specify proxy properties default behavior with an application configuration file and/or by calling methods on the OpenEdge-provided classes, `com.progress.open4gl.RunTimeProperties` and `com.progress.open4gl.javaproxy.Connection`.

There are several properties that govern behavior across an entire application. These properties do not affect any particular object, but they affect the behavior of all objects created by the application. They are referred to as *proxy run-time properties*.

Other properties govern behavior across a single application service. These properties are called *proxy connection properties*.

The following sections describe the different ways you can access proxy, run-time, and connection properties:

- Setting properties on the command line
- RunTimeProperties and Connection classes
- Accessing properties with methods
- Relationship between the RunTimeProperties class and the Connection class
- Modifying an instantiated Connection object’s properties
- Alphabetical listing of properties
Setting properties on the command line

You can set any global property on the command line you use to run your Java client application. Use the following syntax:

**Syntax**

```
-Dpropertyname = value
```

For example:

```
-DPROGRESS.Session.initialConnections = 5
```

For a list of the properties you can set on the command line, see the “Alphabetical listing of properties” section on page 7–15.
RunTimeProperties and Connection classes

Java Open Client provides two classes that you can use to access the properties of a proxy object:

- RunTimeProperties class
- Connection class

**RunTimeProperties class**

The `com.progress.open4gl.RunTimeProperties` class contains static methods and global property values. You can call the methods on this class to change the value of the proxy properties. You can also access the properties directly.

The RunTimeProperties class has a set of default values. Any properties set using an application configuration file are represented by the RunTimeProperties class.

**Connection class**

The `com.progress.open4gl.javaproxy.Connection` class stores AppServer connection information, which can be passed to the AppObject constructor when connecting to the AppServer. It contains methods and property values; however, it does not contain any static methods. You must create an instance of the class and call methods to change the value of the proxy properties.

When you create an instance of the Connection class (a Connection object), it inherits the initial values of its properties from the RunTimeProperties class. You can then override these values programatically. These changes only affect the proxy objects that are associated with this Connection object.

If you change the value of a property in a Connection object after it has been instantiated, the Connection object might not recognize the change, depending on the session model. For more information, see the “Modifying an instantiated Connection object’s properties” section on page 7–14.

**Note:** Changes to the properties of a Connection object have no effect on the global values of the RunTimeProperties class.
Accessing properties with methods

There are two types of methods you can use to access properties on the RunTimeProperties class and on instances of the Connection class:

- General property accessor methods that perform either get or set operations according to the type of method (get or set) and the property name
- Purposed property accessor methods that perform operations on properties for a specified functional purpose

The following sections describe how to access proxy properties with methods:

- General property accessor methods for the RunTimeProperties class
- General property accessor methods for Connection objects
- Available properties
- Accessing properties using purposed accessor methods

General property accessor methods for the RunTimeProperties class

You can access global properties by property name using the get and set general purpose methods. When you use these methods, you must call the correct accessor based on the data type of the property. The following is the syntax for these general methods:

**Get property methods syntax**

```java
public static String getStringProperty(String propName)
public static int getIntProperty(String propName)
public static long getLongProperty(String propName)
public static boolean getBooleanProperty(String propName)
```

**Set property methods syntax**

```java
public static void setStringProperty(String propName, String propValue)
public static void setIntProperty(String propName, int propValue)
public static void setLongProperty(String propName, long propValue)
public static void setBooleanProperty(String propName, boolean propValue)
```

**propName**

The property name. This can be any of the properties listed in the tables in the “Available properties” section on page 7–6.

**propValue**

The value for the property name.
For example, to set the PROGRESS.Session.initialConnections property to 5, use the following method to set an integer property:

```java
RunTimeProperties.setIntProperty("PROGRESS.Session.initialConnections", 5);
```

### General property accessor methods for Connection objects

You can access Connection object properties by property name using the get and set general purpose methods. When you use these methods, you must call the correct accessor based on the data type of the property. The following is the syntax for these general methods:

#### Get property methods

```java
public String getStringProperty(String propName)
public int getIntProperty(String propName)
public long getLongProperty(String propName)
public boolean getBooleanProperty(String propName)
```

#### Set property methods

```java
public void setStringProperty(String propName, String propValue)
public void setIntProperty(String propName, int propValue)
public void setLongProperty(String propName, long propValue)
public void setBooleanProperty(String propName, boolean propValue)
```

- **propName**
  
  The property name. This can be any of the properties listed in the tables in the “Available properties” section on page 7–6.

- **propValue**
  
  The value for the property name.

For example, to set the PROGRESS.Session.initialConnections property to 5, use the following method to set an integer property:

```java
ConnectObj.setIntProperty("PROGRESS.Session.initialConnections", 5);
```

Where `ConnectObj` is an instance of the `com.progress.open4gl.proxy.Connection` class.
Available properties

Table 7–1 lists all of the available properties according to their functional category. For more information on these property categories, see the “Accessing properties using purposed accessor methods” section on page 7–6. For a detailed description of each property, see the “Alphabetical listing of properties” section on page 7–15.

Table 7–1: Available properties

<table>
<thead>
<tr>
<th>Category</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>• PROGRESS.Session.sessionModel</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.initialConnections</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.maxConnections</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.minConnections</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.idleConnectionTimeout</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.connectionLifetime</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.nsClientMinPort</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.nsClientMaxPort</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.nsClientPortRetry</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.nsClientPortRetryInterval</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.nsClientPicklistSize</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.nsClientPicklistExpiration</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.requestWaitTimeout</td>
</tr>
<tr>
<td>Thread control</td>
<td>• PROGRESS.Session.waitIfBusy</td>
</tr>
<tr>
<td>Proxy server</td>
<td>• PROGRESS.Session.proxyHost</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.proxyPort</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.proxyUserId</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.proxyPassword</td>
</tr>
<tr>
<td>Secure Sockets Layer</td>
<td>• PROGRESS.Session.certificateStore</td>
</tr>
<tr>
<td>management</td>
<td>• PROGRESS.Session.noHostVerify</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.noSslSessionReuse</td>
</tr>
<tr>
<td>Tracing</td>
<td>• PROGRESS.Session.enableTracing</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.logFileName</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.loggingLevel</td>
</tr>
</tbody>
</table>

Note: These tracing properties apply to the RunTimeProperties class only. (They do not apply to Connection objects.)

Accessing properties using purposed accessor methods

In addition to the general property accessor methods, you can access properties using individual property accessor methods. Every property has a pair of methods you can use to set and get the value of the property.

When calling these methods on the RunTimeProperties class, use the class name to call the method. For example:

```java
RunTimeProperties.setInitialConnections(5);
```
When calling these methods on a Connection object, use the instance of the object to call the method. For example:

```
ConnectObj.setInitialConnections(5);
```

Where `ConnectObj` is an instance of the `com.progress.open4gl.javaprox.Connection` class.

**Session methods**

Table 7–2 lists the session property methods. For a detailed description of each property, see the “Alphabetical listing of properties” section on page 7–15.

### Table 7–2: Session methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setSessionMode(int sessionModel)</code></td>
<td>Sets <code>PROGRESS.Session.sessionModel</code></td>
</tr>
<tr>
<td><code>getSessionModel()</code></td>
<td>Gets <code>PROGRESS.Session.sessionModel</code></td>
</tr>
<tr>
<td><code>setInitialConnections(int nInitialConnections)</code></td>
<td>Sets <code>PROGRESS.Session.initialConnections</code></td>
</tr>
<tr>
<td><code>getInitialConnections()</code></td>
<td>Gets <code>PROGRESS.Session.initialConnections</code></td>
</tr>
<tr>
<td><code>setMaxConnections(int nMaxConnections)</code></td>
<td>Sets <code>PROGRESS.Session.maxConnections</code></td>
</tr>
<tr>
<td><code>getMaxConnections()</code></td>
<td>Gets <code>PROGRESS.Session.maxConnections</code></td>
</tr>
<tr>
<td><code>setMinConnections(int nMinConnections)</code></td>
<td>Sets <code>PROGRESS.Session.minConnections</code></td>
</tr>
<tr>
<td><code>getMinConnections()</code></td>
<td>Gets <code>PROGRESS.Session.minConnections</code></td>
</tr>
<tr>
<td><code>setIdleConnectionTimeout(int nTimeoutSec)</code></td>
<td>Sets <code>PROGRESS.Session.idleConnectionTimeout</code></td>
</tr>
<tr>
<td><code>getIdleConnectionTimeout()</code></td>
<td>Gets <code>PROGRESS.Session.idleConnectionTimeout</code></td>
</tr>
<tr>
<td><code>setConnectionLifetime(int nConnectionLifetimeSec)</code></td>
<td>Sets <code>PROGRESS.Session.connectionLifetime</code></td>
</tr>
<tr>
<td><code>getConnectionLifetime()</code></td>
<td>Gets <code>PROGRESS.Session.connectionLifetime</code></td>
</tr>
<tr>
<td><code>setNsClientMinPort(int nMinPortNum)</code></td>
<td>Sets <code>PROGRESS.Session.nsClientMinPort</code></td>
</tr>
<tr>
<td><code>getNsClientMinPort()</code></td>
<td>Gets <code>PROGRESS.Session.nsClientMinPort</code></td>
</tr>
<tr>
<td><code>setNsClientMaxPort(int nMaxPortNum)</code></td>
<td>Sets <code>PROGRESS.Session.nsClientMaxPort</code></td>
</tr>
<tr>
<td><code>getNsClientMaxPort()</code></td>
<td>Gets <code>PROGRESS.Session.nsClientMaxPort</code></td>
</tr>
<tr>
<td><code>setNsClientPortRetry(int nRetries)</code></td>
<td>Sets <code>PROGRESS.Session.nsClientPortRetry</code></td>
</tr>
<tr>
<td><code>getNsClientPortRetry()</code></td>
<td>Gets <code>PROGRESS.Session.nsClientPortRetry</code></td>
</tr>
<tr>
<td><code>setNsClientPortRetryInterval(int nDelayMs)</code></td>
<td>Sets <code>PROGRESS.Session.nsClientPortRetryInterval</code></td>
</tr>
</tbody>
</table>
Thread control methods

The AppServer can be accessed from a multi-threaded client. However, the AppServer itself is not multi-threaded. As a result, only one request is serviced at a time. If a method call is made from the client while another one is still running on the AppServer, the application will decide which of the following should happen:

- The second call can be queued and executed when the AppServer becomes available
- The client can get a Java exception

The application can set the run-time properties to do either and can switch the setting at any time. The default is to throw an exception (WaitIfBusy=false).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNsClientPortRetryInterval()</td>
<td>Gets PROGRESS.Session.nsClientPortRetryInterval</td>
</tr>
<tr>
<td>setNsClientPicklistSize(int nPickListSize)</td>
<td>Sets PROGRESS.Session.nsClientPicklistSize</td>
</tr>
<tr>
<td>getNsClientPicklistSize()</td>
<td>Gets PROGRESS.Session.nsClientPicklistSize</td>
</tr>
<tr>
<td>setNsClientPicklistSize(int nPickListExpiration)</td>
<td>Sets PROGRESS.Session.nsClientPicklistExpiration</td>
</tr>
<tr>
<td>getNsClientPicklistExpiration()</td>
<td>Gets PROGRESS.Session.nsClientPicklistExpiration</td>
</tr>
<tr>
<td>setRequestWaitTimeout(int nRqWaitSec)</td>
<td>Sets PROGRESS.Session.requestWaitTimeout</td>
</tr>
<tr>
<td>getRequestWaitTimeout()</td>
<td>Sets PROGRESS.Session.requestWaitTimeout</td>
</tr>
</tbody>
</table>
Accessing properties with methods

The methods in Table 7–3 provide thread control to the Open Client Runtime. For a detailed description of each property, see the “Alphabetical listing of properties” section on page 7–15.

Table 7–3: Thread control methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description/Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>setWaitIfBusy()</td>
<td>Sets <code>PROGRESS.Session.waitIfBusy</code> to true</td>
</tr>
<tr>
<td>setNoWaitIfBusy()</td>
<td>Sets <code>PROGRESS.Session.waitIfBusy</code> to false</td>
</tr>
<tr>
<td>getWaitIfBusy()</td>
<td>Gets <code>PROGRESS.Session.waitIfBusy</code></td>
</tr>
<tr>
<td></td>
<td>Returns TRUE if <code>setWaitIfBusy()</code> was called; FALSE, if</td>
</tr>
<tr>
<td></td>
<td><code>setNoWaitIfBusy()</code> was called</td>
</tr>
</tbody>
</table>

Proxy server specification methods

You use proxy server specification methods if a Proxy Web server exists between the client and the Web server hosting the AppServer Internet Adapter (AIA).

The methods in Table 7–4 provide Proxy Web server support to a Java Open Client application. For a detailed description of each property, see the “Alphabetical listing of properties” section on page 7–15.

Table 7–4: Proxy server specification methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setProxyHost(String hostname)</td>
<td>Sets <code>PROGRESS.Session.proxyHost</code></td>
</tr>
<tr>
<td>getProxyHost()</td>
<td>Gets <code>PROGRESS.Session.proxyHost</code></td>
</tr>
<tr>
<td>setProxyPort(int port)</td>
<td>Sets <code>PROGRESS.Session.proxyPort</code></td>
</tr>
<tr>
<td>getProxyPort()</td>
<td>Gets <code>PROGRESS.Session.proxyPort</code></td>
</tr>
<tr>
<td>setProxyUserId(String userid)</td>
<td>Sets <code>PROGRESS.Session.proxyUserId</code></td>
</tr>
<tr>
<td>getProxyUserId()</td>
<td>Gets <code>PROGRESS.Session.proxyUserId</code></td>
</tr>
<tr>
<td>setProxyPassword(String password)</td>
<td>Sets <code>PROGRESS.Session.proxyPassword</code></td>
</tr>
<tr>
<td>getProxyPassword()</td>
<td>Gets <code>PROGRESS.Session.proxyPassword</code></td>
</tr>
</tbody>
</table>
Secure Sockets Layer management methods

When using HTTPS to access the AppServer, you must supply digital certificates with your client application.

The methods in Table 7–5 provide digital certificate management to the Open Client application. For a detailed description of each property, see the “Alphabetical listing of properties” section on page 7–15.

Table 7–5: Secure Sockets Layer management methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCertificateStore(String certStoreList)</td>
<td>Sets PROGRESS.Session.certificateStore</td>
</tr>
<tr>
<td>getCertificateStore()</td>
<td>Gets PROGRESS.Session.certificateStore</td>
</tr>
<tr>
<td>setNoHostVerify(boolean verify)</td>
<td>Sets PROGRESS.Session.noHostVerify</td>
</tr>
<tr>
<td>getNoHostVerify()</td>
<td>Gets PROGRESS.Session.noHostVerify</td>
</tr>
<tr>
<td>setnoSslSessionReuse(boolean verify)</td>
<td>Sets PROGRESS.Session.noSslSessionReuse</td>
</tr>
<tr>
<td>getnoSslSessionReuse()</td>
<td>Gets PROGRESS.Session.noSslSessionReuse</td>
</tr>
</tbody>
</table>

Tracing methods (RunTimeProperties only)

Tracing logs highlights of proxy execution, including data received by the client and data passed to the AppServer. The methods in Table 7–6 provide tracing functionality to the Open Client application. These methods apply to the RunTimeProperties class only. (They do not apply to Connection objects.) For a detailed description of each property, see the “Relationship between the RunTimeProperties class and the Connection class” section on page 7–12.

Table 7–6: Tracing methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceOn(String filename, int level)</td>
<td>Sets these properties:</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.enableTracing</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.logFileName</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.loggingLevel</td>
</tr>
<tr>
<td>traceOn(int level)</td>
<td>Sets these properties:</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.enableTracing</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.loggingLevel</td>
</tr>
<tr>
<td>traceOn(String filename)</td>
<td>Sets these properties:</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.enableTracing</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.logFileName</td>
</tr>
<tr>
<td>traceOn()</td>
<td>Sets PROGRESS.Session.enableTracing</td>
</tr>
</tbody>
</table>
### Table 7–6: Tracing methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceOff()</td>
<td>Sets <code>PROGRESS.Session.enableTracing</code></td>
</tr>
<tr>
<td>isTracing()</td>
<td>Gets <code>PROGRESS.Session.enableTracing</code></td>
</tr>
</tbody>
</table>
Relationship between the RunTimeProperties class and the Connection class

The values set in a Connection object supersede the static RunTimeProperties class values. This relationship is particularly important if a static RunTimeProperties property is updated after a Connection object has been instantiated. In this situation, the effect on a Connection object instance depends on whether that specific property was overridden in that object after it was instantiated, as described in Table 7–7.

Table 7–7: Relationship between RunTimeProperties and Connection object properties

<table>
<thead>
<tr>
<th>If the RunTimeProperties property has . . .</th>
<th>And the property in the Connection object has . . .</th>
<th>Then the value of the property is returned from the . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not changed</td>
<td>Not changed</td>
<td>RunTimeProperties class</td>
</tr>
<tr>
<td>Changed</td>
<td>Not changed</td>
<td>RunTimeProperties class</td>
</tr>
<tr>
<td>Not changed</td>
<td>Changed</td>
<td>Connection object</td>
</tr>
<tr>
<td>Changed</td>
<td>Changed</td>
<td>Connection object</td>
</tr>
</tbody>
</table>

Note: The converse of this relationship is not true. That is, changing the value of a property for a Connection object does not affect the static value of that property, nor does it affect the value of that property in any other object instance.

The following example uses the waitIfBusy property to illustrate the relationship between RunTimeProperties and Connection object properties:

```java
Connection conn;
boolean ret;
RunTimeProperties.setWaitIfBusy();
conn = new Connection(url, userid, password, Info);
ret = conn.getBooleanProperty("PROGRESS.Session.WaitIfBusy");
System.out.println("(A) WaitIfBusy= " + ret);
RunTimeProperties.setNoWaitIfBusy(); // change static property
ret = conn.getWaitIfBusy(); // equivalent accessor method
System.out.println("(B) WaitIfBusy= " + ret);
conn.setWaitIfBusy(); // change conn object property
ret = conn.getBooleanProperty("PROGRESS.Session.WaitIfBusy");
System.out.println("(C) WaitIfBusy= " + ret);
RunTimeProperties.setBooleanProperty("PROGRESS.Session.WaitIfBusy", false);
ret = conn.getWaitIfBusy();
System.out.println("(D) WaitIfBusy= " + ret);
RunTimeProperties.setWaitIfBusy();
ret = conn.getBooleanProperty("PROGRESS.Session.WaitIfBusy");
System.out.println("(E) WaitIfBusy= " + ret);
```
When executed, this program prints the following result:

(A) WaitIfBusy= true
(B) WaitIfBusy= false
(C) WaitIfBusy= true
(D) WaitIfBusy= true
(E) WaitIfBusy= true
Modifying an instantiated Connection object’s properties

If you change the value of a property in a Connection object after it has been instantiated, the Connection object might not recognize the change, depending on the session model as described in Table 7–8.

Table 7–8: Relationship between Connection object properties and session model

<table>
<thead>
<tr>
<th>If this Connection object property is updated . . .</th>
<th>The update is . . .</th>
<th>Session-managed</th>
<th>Session-free</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRESS.Session.certificateStore</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.connectionLifetime</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.idleConnectionTimeout</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.initialConnections</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.maxConnections</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.minConnections</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.noHostVerify</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.nsClientMaxPort</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.nsClientMinPort</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.nsClientPicklistExpiration</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Recognized once pick list is refreshed</td>
</tr>
<tr>
<td>PROGRESS.Session.nsClientPicklistSize</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Recognized when pick list is refreshed</td>
</tr>
<tr>
<td>PROGRESS.Session.nsClientPortRetry</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.nsClientPortRetryInterval</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.proxyHost</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.proxyPassword</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.proxyPort</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.proxyUserId</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.requestWaitTimeout</td>
<td></td>
<td><strong>Not applicable</strong></td>
<td>Recognized for new requests</td>
</tr>
<tr>
<td>PROGRESS.Session.sessionModel</td>
<td></td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.waitIfBusy</td>
<td></td>
<td>Recognized</td>
<td>Recognized for new requests</td>
</tr>
</tbody>
</table>
Alphabetical listing of properties

This section contains a reference of the properties that you can set for a `RuntimeProperties` class or a `Connection` object, including the applicable session model, a complete description of usage, and the default value for each property.

**PROGRESS.Session.AppServerKeepalive**

**Data Type:** String

**Session model:** All

The `AppServerKeepalive (ASK)` property indicates whether or not the client will participate in a “keep alive” relationship with the AppServer. If configured for the AppServer, the Keepalive feature allows the AppServer to ping the client at set intervals to determine if the client is still viable. Valid values are:

- `AllowServerASK`
- `DenyServerASK`

The absence of this property indicates that the default value for the AppServer Keepalive protocol will be used on this connection.

**Default:** `AllowServerASK`

**PROGRESS.Session.certificateStore**

**Data Type:** String

**Session model:** Session-free and session-managed

The location of the files that contain the root digital certificates that you supply with your client application, when using HTTPS, S, or DCS protocols to access the AppServer. The value for the `PROGRESS.Session.certificateStore` property can be a single filename, a semi-colon delimited list of filenames, or the name of a directory that contains multiple certificates.

**Default:** `psccerts.jar`

**PROGRESS.Session.connectionLifetime**

**Data Type:** Integer

**Session model:** Session-free only

The maximum number of seconds that a given connection can be used before it is destroyed. Connections whose lifetime exceeds the specified value are destroyed as they become available. (This might cause the number of connections in the pool to temporarily fall below the `PROGRESS.Session.minConnections` value.)
An available connection is one that is not currently reserved to run a request. Bound connections associated with remote persistent procedures are not available for reuse until the persistent procedure is deleted. As a result, bound connections remain available as long as necessary, even if they exceed the specified value.

**Default**: 300

**PROGRESS.Session.enableTracing**

**Data Type**: Integer

**Session model**: Session-free and session-managed

Determines whether tracing is on (1) or off (0). Tracing logs highlights of proxy execution, including data received by the client and data passed to the AppServer.

This property applies to the `RunTimeProperties` class only (it does not apply to `Connection` objects).

**Default**: 0

**PROGRESS.Session.idleConnectionTimeout**

**Data Type**: Integer

**Session model**: Session-free only

The amount of time, in seconds, that the client waits before it attempts to shut down idle network connections to the AppServer, based on client demand. However, the number of connections does not fall below the `PROGRESS.Session.minConnections` value.

If this value is zero, idle network connections remain connected.

**Default**: 0

**PROGRESS.Session.initialConnections**

**Data Type**: Integer

**Session model**: Session-free only

The number of socket connections established when the AppObject is instantiated. If the specified number of connections cannot be created, the AppObject constructor fails and any connections that were successfully created are closed.

**Default**: 1

**PROGRESS.Session.logFileName**

**Data Type**: String

**Session model**: Session-free and session-managed
The name of the log file to which trace messages are sent. If this property is not set, the tracing output is Standard Error.

This property applies to the RunTimeProperties class only (not to Connection objects).

**Default:** None

**PROGRESS.Session.loggingLevel**

**Data Type:** Integer

**Session model:** Session-free and session-managed

Determines the level of information logged for tracing. 0 is minimum logging; 7, maximum. The lower the value for this property, the greater the level of information. For more information about logging levels, see *OpenEdge Development: Debugging and Troubleshooting*.

This property applies to the RunTimeProperties class only (it does not apply to Connection objects).

**Default:** 2

**PROGRESS.Session.maxConnections**

**Data Type:** Integer

**Session model:** Session-free only

The maximum number of connections that can be established for a given AppObject. The value must be greater than or equal to zero.

If this value is zero, there is no limit to the number of connections that can be created.

**Default:** 0

**PROGRESS.Session.minConnections**

**Data Type:** Integer

**Session model:** Session-free only

The minimum number of connections that can be established for a given AppObject. The value should not be greater than the value of **PROGRESS.Session.maxConnections**.

If the number of connections falls below the **PROGRESS.Session.minConnections** value, then sufficient new connections are created (when the next request is run) to bring the size of the connection pool back up to the **PROGRESS.Session.minConnections** value.

**Default:** 0
PROGRESS.Session.noHostVerify

**Data Type:** Boolean

**Session model:** Session-free and session-managed

Controls domain name checking. Turns off host verification for a Secure Sockets Layer (SSL) connection using HTTPS, AppServerS, or AppServerDCS protocols.

Without this property specified, the client compares the host name specified in the URL with the Common Name specified in the server certificate, and raises an error if they do not match. With this property specified, the client never raises the error. For more information, see the sections on managing the OpenEdge certificate store in *OpenEdge Getting Started: Installation and Configuration*.

Specifying FALSE enables domain name checking. Specifying TRUE disables domain name checking.

**Default:** FALSE

---

PROGRESS.Session.noSslSessionReuse

**Data Type:** Boolean

**Session model:** Session-free and session-managed

If specified, the connection does not reuse the session IDs to shorten the overhead in reconnecting to the same SSL-enabled server.

**Default:** FALSE

---

PROGRESS.Session.nsClientMaxPort

**Data Type:** Integer

**Session model:** Session-free and session-managed

The minimum value for the UDP port number used by the client when communicating with the Name Server. If this value is zero, OpenEdge chooses the NameServer client port randomly.

This value should be greater than or equal to the value of the PROGRESS.Session.nsClientMinPort property.

**Default:** 0

---

PROGRESS.Session.nsClientMinPort

**Data Type:** Integer

**Session model:** Session-free and session-managed

The minimum value for the UDP port number used by the client when communicating with the NameServer. If this value is zero, OpenEdge chooses the NameServer client port randomly.

**Default:** 0
PROGRESS.Session.nsClientPicklistExpiration

Data Type: Integer

Session model: Session-free only

The maximum amount of time, in seconds, that the client retains an AppServer pick list for an application service.

Default: 300

PROGRESS.Session.nsClientPicklistSize

Data Type: Integer

Session model: Session-free only

The number of AppServer picks to request from the NameServer each time it looks up the available AppServer connections for a given application service name.

Default: 8

PROGRESS.Session.nsClientPortRetry

Data Type: Integer

Session model: Session-free and session-managed

The maximum number of attempts that the client makes to get a valid local UDP port number when attempting to communicate with the NameServer.

Default: 0

PROGRESS.Session.nsClientPortRetryInterval

Data Type: Integer

Session model: Session-free and session-managed

The interval, in milliseconds, that the client waits between attempts to get a valid UDP port number when attempting to communicate with the NameServer.

Default: 0

PROGRESS.Session.proxyHost

Data Type: String

Session model: Session-free and session-managed

The name of the host or the IP address of the host at which the Proxy Web server is located. This value can be a string with either the DNS name of the Proxy Web server or its dot-formatted IP address.
All connections made using the HTTP or HTTPS protocol connect to the proxy server at the specified host and port.

When you specify this parameter, all connections made by an Java client using the HTTP or HTTPS protocol connect to an AppServer Internet Adapter (AIA) instance using the proxy server at the specified host and port.

**Note:** If you specify the `PROGRESS.Session.proxyHost` property, you must also specify the `PROGRESS.Session.proxyPort` property.

**Default:** None

### PROGRESS.Session.proxyPassword

**Data Type:** String

**Session model:** Session-free and session-managed

Use to authenticate the AppServer client to the Proxy Web server. The value can be a string of up to 512 printable ASCII characters.

You should use `PROGRESS.Session.proxyPassword` when `PROGRESS.Session.proxyHost` and `PROGRESS.Session.proxyPort` are specified and the Proxy server requires authentication.

If `PROGRESS.Session.proxyHost` and `PROGRESS.Session.proxyPort` are not specified to the Java Open client, any value for the `PROGRESS.Session.proxyPassword` is ignored.

This property is validated during the connection. If the password is invalid, the connection fails and OpenEdge issues an error message. If `PROGRESS.Session.proxyUserId` is not specified, the connection ignores any value for `PROGRESS.Session.proxyPassword`.

**Default:** None

### PROGRESS.Session.proxyPort

**Data Type:** Integer

**Session model:** Session-free and session-managed

The port number on which the Proxy Web server is listening. You can set the Proxy server port to an integer value from 1 to 65536.

When you specify this property, all connections made by an AppServer client using the HTTP or HTTPS protocol connect to an AppServer Internet Adapter (AIA) instance using the proxy server at the specified host and port.

All connections made using the HTTP or HTTPS protocol connect to the proxy server at the specified host and port.

**Note:** If you specify the `PROGRESS.Session.proxyPort` property, you must also specify the `PROGRESS.Session.proxyHost` property.

**Default:** None
**PROGRESS.Session.proxyUserId**

**Data Type:** String

**Session model:** Session-free and session-managed

Use to authenticate an AppServer client to the Proxy Web server. The user-id can be a string of up to 512 printable ASCII characters, including the space character.

If `PROGRESS.Session.proxyHost` and `PROGRESS.Session.proxyPort` are not specified on the command-line to the Java Open client, any value for the `PROGRESS.Session.proxyUserId` is ignored.

This property is validated during the connection (for an AppServer and a Web Service). If `PROGRESS.Session.proxyUserId` is invalid, the connection fails and OpenEdge issues an error message.

**Default:** None

**PROGRESS.Session.requestWaitTimeout**

**Data Type:** Integer

**Session model:** Session-free only

The way requests are handled when no connections are available and the connection pool is full, specified by one of the values shown in Table 7–9.

<table>
<thead>
<tr>
<th>If the value is . . .</th>
<th>The Open Client Runtime . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Queues the request indefinitely until an AppServer session becomes available</td>
</tr>
<tr>
<td>0</td>
<td>Rejects the request and returns an error message to the client indicating that there are too many concurrent requests</td>
</tr>
<tr>
<td>N</td>
<td>Queues the request the number of seconds specified by the value (N must be greater than zero) until the AppServer becomes available. If no session becomes available in that time, an error message is returned to the client</td>
</tr>
</tbody>
</table>

**Default:** -1
PROGRESS.Session.sessionModel

Data Type: Integer

Session model: Session-free and session-managed

Session model supported by the AppServer operating mode, specified by one of the values shown in Table 7–10.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Session-managed</td>
</tr>
<tr>
<td>1</td>
<td>Session-free</td>
</tr>
</tbody>
</table>

This property must be set for session-free applications. It is optional for session-managed applications.

This value must match the AppServer operating mode or the connection fails.

Default: 0

PROGRESS.Session.waitIfBusy

Data Type: Integer

Session model: Session-managed only

An integer value that determines how to handle client requests to a service that is busy processing a prior request. If the value is 1, the Open Client Runtime queues multiple requests for this service and executes them one at a time until the queue is empty. If the value is 0 and the Open Client Runtime is executing a prior request for the service, each subsequent request for the same service fails until the adapter completes the request it is currently executing.

Default: 0
This chapter provides details on handling the different kinds of exceptions you may encounter. This chapter contains the following sections:

- Exceptions
- General ABL exceptions
- Output ResultSet exceptions
Exceptions

Java clients handle errors by catching exceptions. To enable Java proxies to throw exceptions, OpenEdge establishes two exception class hierarchies:

- **General ABL exceptions** — A hierarchy of exception classes that extend the `Open4GLEXception` class. `Open4GLEXception`, in turn, extends the `com.progress.common.exception.ProException` class.

- **Output ResultSet exceptions** — A `ProSQLException` class, which extends the standard `java.sql.SQLException` class.

In addition to the standard methods provided by `java.lang.Exception` to get the message string, OpenEdge provides the following method to obtain the error number:

```java
long getMessageID()
```

For information about additional methods provided by these exception classes, see the Open Client Toolkit documentation in `OpenEdge-install-directory/java/doc/`.

The following sections describe each `Exception` class in detail.
General ABL exceptions

This section presents the hierarchy of ABL exception classes and then describes each one.

Exception class hierarchy

The ABL exceptions classes are shown hierarchically, in the following example:

```
java.lang.Exception
   com.progress.common.exception.ProException
      com.progress.open4gl.Open4GLException
         com.progress.open4gl.BusySessionException
         com.progress.open4gl.ConnectException
            com.progress.open4gl.BadURLException
            com.progress.open4gl.ConnectFailedException
            com.progress.open4gl.HostUnknownException
            com.progress.open4gl.InvalidNameServerPortException
            com.progress.open4gl.MsgVersionInvalidException
            com.progress.open4gl.NameServerCommunicationsException
            com.progress.open4gl.NameServerInterruptException
            com.progress.open4gl.NameServerMessageFormatException
            com.progress.open4gl.NoSuchAppServiceException
            com.progress.open4gl.OutputSetException
            com.progress.open4gl.RunTime4GLException
               com.progress.open4gl.RunTime4GLErrorException
               com.progress.open4gl.RunTime4GLQuitException
               com.progress.open4gl.RunTime4GLStopException
            com.progress.open4gl.SystemErrorException
```

Exception descriptions

Table 8–1 shows the general ABL exception classes, in alphabetical order.

<table>
<thead>
<tr>
<th>Exception class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadURLException</td>
<td>Invalid URL format in connection parameter.</td>
</tr>
<tr>
<td>BusySessionException</td>
<td>Busy servicing another request.</td>
</tr>
<tr>
<td>ConnectException</td>
<td>Error connecting to AppServer.</td>
</tr>
<tr>
<td>ConnectFailedException</td>
<td>Failed AppServer connection attempt. The message contains the specified reason for the failure.</td>
</tr>
<tr>
<td>HostUnknownException</td>
<td>Invalid NameServer host specified in the URL.</td>
</tr>
<tr>
<td>InvalidNameServerPortException</td>
<td>Invalid NameServer port specified in the URL.</td>
</tr>
<tr>
<td>MsgVersionInvalidException</td>
<td>Incorrect version for message received from NameServer.</td>
</tr>
</tbody>
</table>
NameServerCommunicationsException
NameServer communications error. The message contains the specified reason for the failure.

NameServerInterruptException
NameServer communications broken. Either no response was received from the NameServer or the thread was interrupted before the response was received.

NameServerMessageFormatException
Invalid format for message received from NameServer.

NoSuchAppServiceException
Application Service specified in URL unknown to NameServer.

Open4GLErrorException
The root exception for all Open Client errors. Used for miscellaneous client-side errors.

OutputSetException
Output ResultSet error, for example, trying to get a column value out of column order or trying to make an illegal type conversion.

RunTime4GLErrorException
ABL ERROR condition.

RunTime4GLEXception
ABL run-time errors.

RunTime4GLQuitException
ABL QUIT condition.

RunTime4GLStopException
ABL STOP condition.

SystemErrorException
An unexpected error that indicates an ABL bug or an unusual environmental problem (such as running out of disk space).

<table>
<thead>
<tr>
<th>Exception class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NameServerCommunicationsException</td>
<td>NameServer communications error. The message contains the specified reason for the failure.</td>
</tr>
<tr>
<td>NameServerInterruptException</td>
<td>NameServer communications broken. Either no response was received from the NameServer or the thread was interrupted before the response was received.</td>
</tr>
<tr>
<td>NameServerMessageFormatException</td>
<td>Invalid format for message received from NameServer.</td>
</tr>
<tr>
<td>NoSuchAppServiceException</td>
<td>Application Service specified in URL unknown to NameServer.</td>
</tr>
<tr>
<td>Open4GLErrorException</td>
<td>The root exception for all Open Client errors. Used for miscellaneous client-side errors.</td>
</tr>
<tr>
<td>OutputSetException</td>
<td>Output ResultSet error, for example, trying to get a column value out of column order or trying to make an illegal type conversion.</td>
</tr>
<tr>
<td>RunTime4GLErrorException</td>
<td>ABL ERROR condition.</td>
</tr>
<tr>
<td>RunTime4GLEXception</td>
<td>ABL run-time errors.</td>
</tr>
<tr>
<td>RunTime4GLQuitException</td>
<td>ABL QUIT condition.</td>
</tr>
<tr>
<td>RunTime4GLStopException</td>
<td>ABL STOP condition.</td>
</tr>
<tr>
<td>SystemErrorException</td>
<td>An unexpected error that indicates an ABL bug or an unusual environmental problem (such as running out of disk space).</td>
</tr>
</tbody>
</table>
OpenEdge defines ProSQLException, which is returned from all methods on an OUTPUT TABLE or TABLE-HANDLE ResultSet, including the SDOResultSet object. For example:

```
ProSQLException extends java.sql.SQLException
```

ProSQLException adds a method, getProException(), which returns an OutputSetException.

The SDOResultSet is an interface that maps to ABL SmartDataObjects running on the AppServer. For more information on SDOResultSet objects and how to handle exceptions for them, see Chapter 9, “Using SmartDataObjects from Java Clients.”
Using SmartDataObjects from Java Clients

Any Java application can access a remote ABL SmartDataObject using an interface based on the Java Database Connectivity (JDBC) 2 ResultSet. The supported interface includes many of the standard ResultSet methods and adds a number of extensions. This chapter describes how to access and use this interface, as outlined in the following sections:

- What is a SmartDataObject?
- Accessing remote SmartDataObjects from Java
- Connecting to an AppServer using an SDOAppObject
- Creating an SDOResultSet object
- Working with SDOResultSet objects
- Multi-threading
- Errors and exceptions
- Choosing JDBC ResultSets or SDOResultSets to access OpenEdge data
- Accessing the sample application
What is a SmartDataObject?

A SmartDataObject is an ABL persistent procedure created using the SmartObject technology of the OpenEdge AppBuilder. As such, it provides a number of SmartObject methods (implemented as ABL internal procedures and functions) that allow an ABL application to dynamically query and update OpenEdge databases (and DataServers). It also implements the patented SmartObject messaging technology that allows it to communicate and exchange data with other types of ABL SmartObjects (also implemented in ABL). For more information on SmartDataObjects, see OpenEdge Development: AppBuilder, OpenEdge Development: ADM and SmartObjects, and OpenEdge Development: ADM Reference.

Using the OpenEdge AppBuilder, an ABL programmer can modify and extend the standard SmartDataObject methods to encapsulate any business logic needed to filter data into or out of an OpenEdge-managed database. When the SmartDataObject is deployed remotely (running on an AppServer), this same business logic becomes available to manage data from your Java Open Client application.
Accessing remote SmartDataObjects from Java

Figure 9–1 shows how a Java Open Client accesses a SmartDataObject running on an AppServer. The Java application creates an AppObject that connects to an AppServer running a SmartDataObject. The AppObject, or one of its SubAppObjects, provides an SDOResultSet object that makes the remote SmartDataObject available to the application as an extended JDBC 2 ResultSet.

Once an SDOResultSet is created, the client application can work on it locally without maintaining the connection to the AppServer. Later, the application can establish an AppServer connection again to update rows, open the query again, and so on.

**Requirements for accessing SmartDataObjects**

To enable any Java client to access a remote SmartDataObject as a JDBC 2 ResultSet, you must:

- Have at least one remote SmartDataObject deployed on the connected AppServer
- Call the AppObject or SubAppObject class factory method, `createSDOResultSet()`, to create an SDOResultSet object that can access the specified SmartDataObject

Only one thread can access a single SDOResultSet or any other single output ResultSet. If more than one thread tries to access the same SDOResultSet or output ResultSet, the result is unpredictable.
Example Java access to a SmartDataObject

The following example application connects to the default AppServer and Application Service using an SDOAppObject, which is a prebuilt AppObject installed with Open Client Toolkit. It then accesses the SmartDataObject, CustSDO.w, as an SDOResultSet and changes the name of customer number 1 to “John”, as shown:

```java
import com.progress.open4gl.*;

public class Sample1{
    public static void main(String args[]) throws Exception {
        // Create an SDOAppObject and connect to the AppServer
        SDOAppObject appObj = new SDOAppObject();

        // Create an SDOResultSet object
        com.progress.open4gl.SDOResultSet rSet =
        appObj._createSDOResultSet("CustSDO.w");

        rSet.first(); // Position on the first customer
        rSet.updateString("Name", "John"); // Modify the name to "John"
        rSet.updateRow(); // Send the new value to the AppServer

        // Close the SDOResultSet object.
        rSet.close();

        // release the AppObject
        appObj._release();
    }
}
```

This SDOResultSet object extends the JDBC 1 ResultSet interface to include JDBC 2 functionality. Thus, using standard JDBC 2 ResultSet methods on an SDOResultSet object, you can access ABL data provided by the specified SmartDataObject in the same way as a standard JDBC 2 ResultSet. SDOResultSet also further extends JDBC 2 functionality to access the unique capabilities of SmartDataObjects.
SmartDataObject access tools and documentation

Complete support for Java access to remote SmartDataObjects includes ProxyGen (the Open Client Proxy Generator) and a small set of classes. These classes, together with SDOResultSet, are installed as part of the Open Client Runtime. For reference information on each class and interface, see the documentation in `OpenEdge-install-directory\java\doc`. For information on ProxyGen, see *OpenEdge Development: Open Client Introduction and Programming*.

Objects required to access SmartDataObjects

The Open Client Runtime provides three Java classes that you need to access a remote SmartDataObject from a Java application. These, and other supporting classes, are provided in the `com.progress.open4gl` package:

- **SDOResultSet** — This is an extended subset of the JDBC 2 ResultSet interface, which provides access to a SmartDataObject. Because all instances of a SmartDataObject present the same ABL API, the one SDOResultSet class allows you to access any such instance.

  SDOResultSet allows you to access only the default SmartDataObject API generated by the OpenEdge AppBuilder. If you need to access custom programmer extensions to this API, you must access the SmartDataObject API directly. For more information, see *OpenEdge Development: Open Client Introduction and Programming*.

- **AppObject** or **SubAppObject** — These objects allow you to create an SDOResultSet (using `_createSDOResultSet()`) to access any remote SmartDataObject. You can access the SDOResultSet object from the following proxy objects:

  - Any standard AppObject or SubAppObject you generate in ProxyGen
  - A prebuilt `com.progress.open4gl.SDOAppObject` proxy that is installed as part of Open Client Runtime

Using a standard AppObject or SubAppObject allows you to map other AppServer procedures besides SmartDataObjects into the generated proxy. The SDOAppObject installed with OpenEdge is a complete proxy that allows you to access any SmartDataObject without having to build a proxy in ProxyGen. Essentially, it is a complete AppObject built to access only SDOResultSet objects.

Thus, if you need to access only SmartDataObjects on the AppServer and no other AppServer procedures, you can use this SDOAppObject as is. If you need to access other ABL procedures on the AppServer in addition to the SmartDataObject, you also can use any AppObject or SubAppObject generated with ProxyGen to access any SmartDataObject.

- **SDOParameters** — This is an object you can pass to the SDOResultSet constructor through the `_createSDOResultSet()` method. SDOParameters allows you to modify the default behavior of the SDOResultSet instance. You can create an SDOParameters object using its constructor with no parameters (`SDOParameters()`) and initialize it using its methods.
Developing and deploying a SmartDataObject-aware application

The procedure below will take you through the process of making your application SmartDataObject aware.

To develop and deploy Java applications that access SmartDataObjects:

1. Use the AppBuilder to create and compile a SmartDataObject.

2. Deploy the compiled SmartDataObject (the *.r file) on the AppServer side. (The compiled SmartDataObject must be on the PROPATH of the AppServer.)

3. Copy the ABL-compiled ADM2 (Application Development Model, Version 2) r-code files to a directory in the AppServer PROPATH. You can find these files in the OpenEdge-install-directory\gui\adm2 directory, wherever you have the OpenEdge AppBuilder installed. To complete this step, copy these files to your AppServer’s OpenEdge-install-directory\tty\adm2\ directory.

4. Deploy the Open Client Runtime package on the client side. For more information, see Chapter 1, “Configuring and Deploying Java Open Client Applications.”

5. If your Java application needs only SmartDataObject access—it does not access other remote ABL procedures—you are ready to write Java applications that access the SmartDataObject.

   If your Java application needs to access ABL procedures other than SmartDataObjects, use ProxyGen to create the proxies and deploy them on the client side (see OpenEdge Development: Open Client Introduction and Programming). You do not have to map the SmartDataObject as a ProcObject, however; ProxyGen provides built-in access to SmartDataObjects in Java.

Extending a SmartDataObject

If you extend (add new ABL methods to) a SmartDataObject, to access these extensions you must access the SmartDataObject as a separate ProcObject you create in ProxyGen. This allows you to directly access the SmartDataObject as an ABL persistent procedure with complete access to its public internal procedures and user-defined functions. For more information on accessing a SmartDataObject as a ProcObject, see OpenEdge Development: Open Client Introduction and Programming.
Writing a SmartDataObject-aware Java client

The procedure below will take you through the process of writing your application SmartDataObject-aware Java client.

To write a Java client that accesses a SmartDataObject:

1. Connect to the AppServer by instantiating an AppObject or SDOAppObject class.
2. Create an SDOResultSet object based on the specified SmartDataObject, using the `AppObject._createSDOResultSet()` class factory method, where `AppObject` is the SDOAppObject, AppObject, or an associated SubAppObject instance.
3. Invoke standard JDBC 2 and extended methods on the SDOResultSet, to query and modify data accessed through the SmartDataObject. See the JDBC documentation.
4. Close the SDOResultSet using the `SDOResultSet.close()` method.
5. Release the SDOAppObject, AppObject, or SubAppObject using the `AppObject._release()` method.
Connecting to an AppServer using an SDOAppObject

To establish a connection to an AppServer, you must instantiate an SDOAppObject.

The following constructors for the SDOAppObject class take the same parameters and connect to an AppServer in the same way as any AppObject constructor:

**Syntax**

```java
public SDOAppObject(com.progress.open4gl.javaproxy.Connection connectObj)
```

**Syntax**

```java
public SDOAppObject(String urlString, String userId, String password, String AppServer-info)
```

**Syntax**

```java
public SDOAppObject(String userId, String password, String AppServer-info)
```

**Syntax**

```java
public SDOAppObject()
```

**ConnectObj**

Specifies the Connection object that contains the AppServer connection information. A Connection object can be instantiated with one of the constructors described in the “Connection class” section on page 3–2.

**urlString**

Specifies the AppServer connection information.

**userId, password, AppServer-info**

Specify any information required by the AppServer application at connection time.

For more information on how to specify these parameters, see Chapter 3, “Connecting to an AppServer.”
Creating an SDOResultSet object

Once you instantiate an SDOAppObject, AppObject, or SubAppObject, you can use the object’s _createSDOResultSet() factory method to create an SDOResultSet object and associate it with a specified SmartDataObject.

You cannot create an SDOResultSet instance by calling its constructor. The _createSDOResultSet() factory method prepares and calls the SDOResultSet constructor based on the AppServer connection maintained by the associated AppObject.

The capabilities of SmartDataObjects provide several options for presenting data. You can pass parameters to the _createSDOResultSet() method that enable one or more of these options.

The _createSDOResultSet() method

You can call one of three overloaded versions of _createSDOResultSet(), which supports different combinations of the four factory method parameters.

The following version uses the KEEP_ROWS scrolling mode and the default where and sort expressions:

```java
public SDOResultSet _createSDOResultSet(String sdoName)
```

The following version uses the KEEP_ROWS scrolling mode:

```java
public SDOResultSet _createSDOResultSet(String sdoName, String where,
String sortBy)
```

The following version uses whatever scrolling mode is specified by parameters:

```java
public SDOResultSet _createSDOResultSet(String sdoName, String where,
String sortBy, SDOParameters parameters)
```

For more information, see the “Understanding SDOResultSet scrolling modes” section on page 9–13.

The following parameters apply to _createSDOResultSet():

- **sdoName**
  
  (Required) Specifies the filename of the SmartDataObject (custSDO.w, for example).

- **where**
  
  Allows the application to open a query with a where expression that is different than the default one specified when the SmartDataObject was created. For more information, see the description of the setQueryWhere method in OpenEdge Development: ADM Reference.
**sortBy**

Allows the application to open a query with a sort expression that is different from the default one specified when the SmartDataObject was created. You can specify `sortBy` using one of the following syntax styles:

**Syntax**

```
[ field | field ASCEND | field DESCEND
 [ , { field | field ASCEND | field DESCEND } ] ... ]
```

```
[ field | field ASCEND | field DESCEND
 [ BY { field | field ASCEND | field DESCEND } ] ... ]
```

**field**

A valid database field referenced by the SmartDataObject. Specifying no fields (""") is equivalent to null. Examples include:

"Name"
"Name, City"
"Name DESCEND, City, Zip"
"Name BY City"
"Name DESCEND BY City BY Zip"

For more information on specifying `field`, see the description of the `setQuerySort` method in *OpenEdge Development: ADM and SmartObjects*.

**parameters**

Used for passing any additional initial parameters to the ResultSet object, such as the scrolling mode. If `parameters` does not specify a scrolling mode, the default is used.

**Setting and getting values on the SDOParameters object**

An SDOParameters object in `com.progress.open4g1` can be supplied as a parameter to the SDOResultSet factory method that allows you to control various ResultSet properties. You need to set only the values that differ from the defaults. The methods to set and get these values on the SDOParameters object follow.

The following method sets the row ID of the first row to be fetched from the created and opened SDOResultSet:

```java
void setRowIdentity(String rowId)
```

A null `rowId` is equivalent to not calling `setRowIdentity()` (also see the `reOpenQuery(String rowId)` extended method in the “Miscellaneous management methods” section on page 9–24). Usually, it is more efficient to use `setRowIdentity()` than to create the SDOResultSet and then call the extended method, `absolute(String rowId)`, to reposition the cursor on the specified row.
If you create the SDOResultSet in PREFETCH scrolling mode, you cannot access rows before the rowId that you set with setRowIdentity(). For more information on PREFETCH, see the “Understanding SDOResultSet scrolling modes” section on page 9–13.

The rowId value is equivalent to the value returned by the ABL ROWID function and the SDOResultSet.getRowIdentity() method. For more information on ROWID, see OpenEdge Getting Started: ABL Essentials or the appropriate OpenEdge DataServer Guide for any DataServer accessed by the SmartDataObject. For more information on the SDOResultSet.getRowIdentity() method, see the “Miscellaneous management methods” section on page 9–24.

The following method gets the row ID of the first row to be fetched from the created and opened SDOResultSet:

**Syntax**

```
String getRowIdentity()
```

The following method sets the number of rows the scrolling mechanism fetches each time it accesses the AppServer:

**Syntax**

```
void setFetchSize(int fetchsize)
```

The default is 200.

The following method gets the number of rows the scrolling mechanism fetches each time it accesses the AppServer:

**Syntax**

```
int getFetchSize()
```

The following method sets the Stateless mode for the SDOResultSet:

**Syntax**

```
void setStateless(boolean state)
```

Specify state as true for a stateless SDOResultSet and false for a non-stateless SDOResultSet. For more information, see the “Understanding SDOResultSet stateless mode” section on page 9–14.

The following method gets the Stateless mode for the SDOResultSet:

**Syntax**

```
boolean getStateless()
```

For more information, see the “Understanding SDOResultSet stateless mode” section on page 9–14.
The following method sets the scrolling mode for the SDOResultSet specified as a class constant in com.progress.open4gl.SDOScrollingMode:

**Syntax**

```java
void setScrollingMode(SDOScrollingMode constant)
```

If the SDOResultSet is Stateless, the default is SDOScrollingMode.PREFETCH; otherwise, the default is SDOScrollingMode.KEEP_ROWS. The other supported scrolling mode is SDOScrollingMode.FORWARD_ONLY. For more information, see the “Understanding SDOResultSet scrolling modes” section on page 9–13.

The following method gets the scrolling mode specified for the SDOResultSet, returned as a class constant in com.progress.open4gl.SDOScrollingMode:

**Syntax**

```java
SDOScrollingMode getScrollingMode()
```

For more information, see the “Understanding SDOResultSet scrolling modes” section on page 9–13.

The following method sets the maximum number of rows to be fetched when the scrolling mode is SDOScrollingMode.PREFETCH:

**Syntax**

```java
void setPrefetchMaxRows(int maxRows)
```

The setPrefetchMaxRows() method has no effect for scrolling modes other than SDOScrollingMode.PREFETCH. The default for this mode is to get all the rows of the SDOResultSet. For more information, see the “Understanding SDOResultSet scrolling modes” section on page 9–13 and the “Understanding SDOResultSet stateless mode” section on page 9–14.

The following method gets the value for the maximum number of rows to be fetched when the scrolling mode is SDOScrollingMode.PREFETCH:

**Syntax**

```java
int getPrefetchMaxRows()
```

For more information, see the “Understanding SDOResultSet scrolling modes” section on page 9–13 and the “Understanding SDOResultSet stateless mode” section on page 9–14.
Understanding SDOResultSet scrolling modes

Different scrolling modes allow you to control response time, memory requirements, and the ResultSet isolation level. The isolation level (in this context) is the visibility of modifications made to the ResultSet by applications other than the current Open Client. Each mode represents tradeoffs between these requirements and capabilities. You can set the scrolling mode on the SDOParameters object using the setScrollingMode(SDOScrollingMode constant) method and pass the object to the _createSDOResultSet() factory method. The value of constant is a class constant of the SDOScrollingMode class provided by com.progress.open4gl. The values you can pass for constant are as follows:

- **SDOScrollingMode.PREFETCH** — This is the default and only allowable mode for a stateless SDOResultSet. In this mode, the whole ResultSet is fetched into memory when the SDOResultSet object is created. The response time and amount of memory required is directly related to the size of the ResultSet. This mode creates the highest level of isolation. (For more information, see the information on visibility of updates in the “Updating SDOResultSet objects” section on page 9–19.) The size of the ResultSet is known from the beginning and does not change until the ResultSet is closed or the query is reopened. Also, navigation in this mode is guaranteed to be fast since all the rows are in memory. This mode is ideal for small ResultSets in combination with slow networks.

**Note:** If you are accessing a large ResultSet, you might want to use the setPrefetchMaxRows() method to limit the number of rows you fetch. Otherwise, your Java application can run with an excessively long response time and large memory consumption. This can be a special problem for applications accessing stateless SDOResultSets, which always are in the PREFETCH scrolling mode. For more information, see the “Understanding SDOResultSet stateless mode” section on page 9–14.

- **SDOScrollingMode.KEEP_ROWS** — This is the default mode for an SDOResultSet that is not stateless. The rows are not prefetched but retrieved from the AppServer on demand. Rows that were retrieved are kept in memory until the ResultSet is closed or a row is refreshed or updated. The response time to get the first rows in KEEP_ROWS mode is shorter than in PREFETCH mode. Stateless SDOResultSets do not support this mode (see the “Understanding SDOResultSet stateless mode” section on page 9–14).

- **SDOScrollingMode.FORWARD_ONLY** — In this mode, only the next() navigation method is supported; the cursor cannot be randomly repositioned. Since there is no need to maintain an in-memory cache of rows, the amount of required memory does not depend on the size of the ResultSet. This scrolling mode should be used, therefore, when a large ResultSet is accessed and the next() method is sufficient; no other navigation methods are required.
Understanding SDOResultSet stateless mode

The SDOResultSet Stateless mode allows the client to scroll through and update data without maintaining an active persistent procedure on the Application Server process. Its primary value is in support of an AppServer running in the stateless operating mode. In this operating mode, maintaining a persistent procedure on the Application Server process binds the process to the client, making it unable to serve other clients.

In Stateless mode, the underlying ProcObject (mapped to the SmartDataObject) is held only for the short duration when more data is fetched from the AppServer and updates are sent to the AppServer. After that interaction, the ProcObject is released and the SmartDataObject persistent procedure is deleted.

**Note:** Although the most useful application of Stateless mode is with a stateless AppServer, the Stateless mode for SDOResultSet is orthogonal to the operating mode of the AppServer. The client can set the SDOResultSet to Stateless mode even if the AppServer is not running in stateless operating mode. The client also can create a non-stateless SDOResultSet to access a stateless AppServer.

An SDOResultSet is in Stateless mode if `setStateless(true)` is called on the SDOParameters of the `_createSDOResultSet()` class factory method.

The SDOResultSet Stateless mode has the following limitations:

- Only batch updates are allowed. For more information, see the “Using batch mode [extension]” section on page 9–22.
- Stateless mode is only allowed with the PREFETCH scrolling mode. For more information, see the “Understanding SDOResultSet scrolling modes” section on page 9–13.

As with any SDOResultSet opened in PREFETCH mode, Stateless mode causes the SDOResultSet to return all its rows to the client application at one time. This can pose performance problems for very large ResultSets. To manage an SDOResultSet created in PREFETCH mode, you can explicitly set the maximum number of rows fetched for a query by using the SDOParameters.setPrefetchMaxRows() method. Typically, you use this method together with the SDOResultSet.reOpenQuery(String rowId) method to limit the number of rows fetched for any one query and fetch the next set of rows for a different instance of the query. (For more information on the reOpenQuery(String rowId) method, see the “Miscellaneous management methods” section on page 9–24.) Thus:

- The SDOParameters.setPrefetchMaxRows() sets the maximum rows to fetch for the query when the SDOResultSet is created
- The SDOResultSet.reOpenQuery(String rowId) fetches a new set of rows, using another instance of the same query and maximum number of rows, but starting from the specified rowId, which identifies one of the rows in the previous instance of the query
The following block of Java code uses these two methods to manage the fetching of rows for a stateless SDOResultSet:

```java
// The code shows the fetching of 19 rows in two batches under the Stateless mode.
SDOParameters params = new SDOParameters();
params.set Stateless(true);

// A maximum of 10 rows is fetched
params.setPrefetchMaxRows(10);
com.progress.open4gl.SDOResultSet rSet =
    appObj._createSDOResultSet("CustSDO.w",null,null,params);
String lastRowid = null;

// Displays the first 10 rows
while (rSet.next())
{
    System.out.println(
        rSet.getObject(1) + " " +
        rSet.getObject(2) + " " +
        rSet.getObject(3));
    lastRowid = rSet.getRowIdentity();
}

// Get 10 more rows starting at lastRowid
rSet.reOpenQuery(lastRowid);

// Skip the row we already visited
rSet.next();

// Displays the next 9 rows.
while (rSet.next())
{
    System.out.println(
        rSet.getObject(1) + " " +
        rSet.getObject(2) + " " +
        rSet.getObject(3));
}
```

The code sets the maximum number of rows to fetch to 10 and creates the stateless SDOResultSet, rSet, to access the SmartDataObject, CustSDO.w. It then fetches the rows, maintaining the last-fetched row ID in lastRowid. Finally, it uses lastRowid to reopen the query and fetch the next 10 rows.

Because you must use the last-fetched row ID as the starting point, this causes the same row to be returned as the first row in the next query. The code thus uses the next() method to skip the already-visited row, and it fetches the next 9 rows that have not yet been fetched, for a total of 10 returned for the query.
Working with SDOResultSet objects

This section describes how to use the methods supported on SDOResultSet, including both the subset of standard JDBC 2 methods and SDOResultSet extensions to JDBC 2. For information on the full JDBC 2 standard, see the JDBC 2 specification and the JDK API description. This section also describes other objects and methods that affect SDOResultSet functionality and provides information about the following:

- Detaching SDOResultSet objects [extension]
- Navigating SDOResultSet rows
- Getting SDOResultSet column values
- Updating SDOResultSet objects
- Miscellaneous management methods
- Meta data methods

In the following sections, [Extension] appears after the listed syntax for methods that are SDOResultSet extensions to JDBC 2. If a section describes methods that are all SDOResultSet extensions, [Extension] also appears after the section title.

To apply some of these extensions, you must pass parameters to the SDOResultSet factory method (_createSDOResult()), which creates the SDOResultSet object.

Detaching SDOResultSet objects [extension]

A stateless SDOResultSet can be detached from an AppObject (or SDOAppObject) and re-attached to a specified (not necessarily the same) AppObject. By detaching the ResultSet and releasing the AppObject (disconnecting the AppServer), the Open Client can use and manipulate the ResultSet without holding any resources on the AppServer.

Most of the SDOResultSet methods available on a normal stateless SDOResultSet are available on a detached SDOResultSet. The exceptions are these methods, which require an AppServer connection:

- reOpenQuery() — See the “Miscellaneous management methods” section on page 9–24.
- sendBatch() — See the information on Batch mode in the “Updating SDOResultSet objects” section on page 9–19.
- sendBatchAndReOpen() — See the information on Batch mode in the “Updating SDOResultSet objects” section on page 9–19.

Thus, before calling a method that requires an AppServer connection, you must attach the SDOResultSet to the same or a different AppObject with a connection to the AppServer. Here are the methods that provide this functionality.

The following method detaches from the AppObject that created the ResultSet:

Syntax

```
void detachFromAppObj() [Extension]
```
The following method attaches to the AppObject, SubAppObject, or SDOAppObject specified by `ProxyObject`:

**Syntax**

```java
void attachToAppObj(SDOFactory ProxyObject) [Extension]
```

**Note:** ProxyGen-generated AppObjects and SubAppObjects, as well as the SDOAppObject proxy, all implement the SDOFactory interface.

The following method returns `true` if the SDOResultSet is attached to an AppObject; otherwise, it returns `false`:

**Syntax**

```java
boolean isAttached() [Extension]
```

The following is a typical scenario for using the detaching and attaching functionality:

1. The SDOResultSet is created. Because the scrolling mode is `PREFETCH`, all the rows are read and returned to the client.
2. The SDOResultSet is detached from the AppObject.
3. The AppObject is released (the AppServer connection closed).
4. The client accesses data and possibly calls `startBatch()` and updates some data.
5. The SDOResultSet is attached to another open AppServer connection.
6. The client calls `sendBatch()` to apply the modifications.

The SDOResultSet interface extends the `java.io.Serializable` interface. As a result, a detached SDOResultSet can be stored and retrieved from disk using the standard Java serialization mechanism. It also can be passed by value and returned from methods using remote method invocation (RMI). After being restored from disk or being received through an RMI call, the SDOResultSet can be re-attached to an AppObject using the `attachToAppObj()` method.

### Navigating SDOResultSet rows

The methods in Table 9–1 change or report the position of the cursor in the SDOResultSet.

#### Table 9–1: Navigating SDOResultSet rows

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean absolute(int n)</td>
<td>Positions the cursor on the n-th row. A negative number positions the cursor relative to the end of the SDOResultSet. The index (n) is 1-based.</td>
</tr>
<tr>
<td>boolean previous()</td>
<td>Moves the cursor backwards.</td>
</tr>
<tr>
<td>boolean next()</td>
<td>Moves the cursor forward.</td>
</tr>
</tbody>
</table>
Using SmartDataObjects from Java Clients

Table 9–1: Navigating SDOResultSet rows

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean relative(int n)</td>
<td>Moves the cursor n rows forward (backward if n is negative).</td>
</tr>
<tr>
<td>boolean first()</td>
<td>Positions the cursor on the first row.</td>
</tr>
<tr>
<td>void beforeFirst()</td>
<td>Positions the cursor before the first row.</td>
</tr>
<tr>
<td>boolean last()</td>
<td>Positions the cursor on the last row.</td>
</tr>
<tr>
<td>void afterLast()</td>
<td>Positions the cursor after the last row.</td>
</tr>
<tr>
<td>boolean isAuthenticated()</td>
<td>Is the cursor after the last row?</td>
</tr>
<tr>
<td>boolean isFirst()</td>
<td>Is the cursor before the first row?</td>
</tr>
<tr>
<td>boolean isLast()</td>
<td>Is the cursor on the last row?</td>
</tr>
<tr>
<td>int getRow()</td>
<td>Gets the position (1-based) of the cursor.</td>
</tr>
</tbody>
</table>

Getting SDOResultSet column values

These methods return data from a specified column on an SDOResultSet according to the specified data type. All column access methods in SDOResultSet use the flat model (see OpenEdge Development: Open Client Introduction and Programming). The column is specified by name (columnName) or number (columnIndex), as follows:

- BigDecimal getBigDecimal(int columnName)
- BigDecimal getBigDecimal(String columnName)
- boolean getBoolean(int columnName)
- boolean getBoolean(String columnName)
- byte[] getBytes(int columnName)
- byte[] getBytes(String columnName)
- double getDouble(int columnName)
- double getDouble(String columnName)
- int getInt(int columnName)
- int getInt(String columnName)
- java.sql.Blob getBlob(int columnName)
- java.sql.Blob getBlob(String columnName)
- java.sql.Clob getClob(int columnName)
- java.sql.Clob getClob(String columnName)
- java.sql.Date getDate(int columnName)
• `java.sql.Date getDate(String columnName)`
• `java.sql.Timestamp getTimestamp(int columnName)`
• `java.sql.Timestamp getTimestamp(String columnName)`
• `long getLong(int columnName)`
• `long getLong(String columnName)`
• `Object getObject(int columnName)`
• `Object getObject(String columnName)`
• `String getString(int columnName)`
• `String getString(String columnName)`

Some `java.util.Date` and `java.sql.Date` methods are being deprecated by JavaSoft in favor of using the more robust `java.util.GregorianCalendar`. These are the methods that get column values for instances of this class:

• `java.util.GregorianCalendar getGregorianCalendar(String columnName)`
  [Extension]
• `java.util.GregorianCalendar getGregorianCalendar(int columnIndex)`
  [Extension]

For more information on the mapping between ABL and Java data types, see the sections on data type mapping for temp-table fields and passing `OUTPUT TABLE` parameters in Chapter 4, “Passing Parameters.”

**Updating SDOResultSet objects**

The procedure below will take you through the process of updating your SDOResultSet objects.

To update an SDOResultSet object:

1. Make a change locally in the client application. For example, update column values in an existing row or create a new row (`moveToInsertRow()`).

2. Commit the local changes to the database (through the remote SmartDataObject), by executing a row method, such as `updateRow()` or `insertRow()`.

**Caution:** If you move the cursor from a row where you updated columns (for example, using `next()`), before you commit the column updates to the database, your column updates to that row will be lost.

**Transactions and concurrency control**

By default, all updates that change the database occur in a transaction managed by the remote SmartDataObject. Typically, any method that updates or deletes an existing row, or that inserts a new row causes the SmartDataObject to complete a transaction for that row. Note that updating column values does not, in itself, change the database. Only row operations can actually cause a transaction to occur.
You also can extend a transaction to include modifications to multiple rows by using an SDOResultSet in Batch mode or by remotely controlling an automatic transaction, if one is available on the AppServer. For more information on Batch mode, see the “Using batch mode [extension]” section on page 9–22. For more information on automatic transactions, see the “Using extended transactions” section on page 9–22.

SmartDataObjects use optimistic concurrency control, which means records are read without a lock. Locks are acquired only for the short duration of the transaction when modifications are performed. Before updating or deleting a row, the remote SmartDataObject implementation compares the original values of the columns to their values in the database to determine if the columns were modified by another user. If a column was modified, the modification does not succeed and the transaction is rolled back. A failed update then throws an SDOModificationException. If the SmartDataObject implementation tries to modify or delete a row that is locked, it also throws an SDOModificationException. For more information on SDOModificationException, see the “Errors and exceptions” section on page 9–31.

**Updating column values**

The following methods modify a column in an SDOResultSet according to the specified data type. All column access methods in SDOResultSet use the flat model (see *OpenEdge Development: Open Client Introduction and Programming*). The column is specified by name (columnName) or number (columnIndex) and is set to the specified value (value), as follows:

- void updateBigDecimal(int columnIndex, BigDecimal value)
- void updateBigDecimal(String columnName, BigDecimal value)
- void updateBlob(int columnIndex, java.sql.Blob value)
- void updateBlob(String columnName, java.sql.Blob value)
- void updateBoolean(int columnIndex, boolean value)
- void updateBoolean(String columnName, boolean value)
- void updateBytes(int columnIndex, byte x[])
- void updateBytes(String columnName, byte x[])
- void updateClob(int columnIndex, java.sql.Clob value)
- void updateClob(String columnName, java.sql.Clob value)
- void updateDate(int columnIndex, java.sql.Date value)
- void updateDate(String columnName, java.sql.Date value)
- void updateDouble(int columnIndex, double value)
- void updateDouble(String columnName, double value)
- void updateInt(int columnIndex, int value)
- void updateInt(String columnName, int value)
- void updateLong(int columnIndex, long value)
- void updateLong(String columnName, long value)
- void updateNull(int columnIndex)
- void updateNull(String columnName)
- void updateObject(int columnIndex, Object value)
- void updateObject(String columnName, Object value)
- void updateString(int columnIndex, String value)
• void updateString(String columnName, String value)
• void updateTimestamp(int columnIndex, java.sql.Timestamp value)
• void updateTimestamp(String columnName, java.sql.Timestamp value)

Some java.util.Date and java.sql.Date methods are being deprecated by JavaSoft in favor of using the more robust java.util.GregorianCalendar. These are the methods that update column values with instances of this class:

• void updateGregorianCalendar(String columnName, GregorianCalendar date) [Extension]
• void updateGregorianCalendar(int columnIndex, GregorianCalendar date) [Extension]

For more information on the mapping between ABL and Java data types for SQL ResultSet applications, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

Inserting, deleting, and updating rows

The methods in Table 9–2 manage operations on rows in the SDOResultSet.

Table 9–2: Managing operations on SDOResultSet rows

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean rowDeleted()</td>
<td>Returns true if the cursor is positioned on a deleted row, whether or not the deletion is sent to the AppServer because it does not make sense to allow the user to manipulate a row that is logically deleted. The only time the row is not also physically deleted on the AppServer is if you are using a batch update. For more information, see the “Using batch mode [extension]” section on page 9–22.</td>
</tr>
<tr>
<td>boolean rowInserted()</td>
<td>Returns true if the cursor is positioned on an inserted row position and the row is not yet sent to the AppServer.</td>
</tr>
<tr>
<td>boolean rowUpdated()</td>
<td>Returns true if the cursor is positioned on a row that is modified but not yet sent to the AppServer.</td>
</tr>
<tr>
<td>void CancelRowUpdates()</td>
<td>Cancels the updates by rolling back the effects of all method calls that updated column values. To cancel these updates, you must invoke this method before any call to updateRow() or insertRow().</td>
</tr>
<tr>
<td>void deleteRow()</td>
<td>Deletes the row in the current position and sends the delete request to the AppServer.</td>
</tr>
<tr>
<td>void insertRow()</td>
<td>Sends the newly created row to the AppServer.</td>
</tr>
<tr>
<td>void moveToCurrentRow()</td>
<td>Moves the cursor from the insert row back to the current position.</td>
</tr>
<tr>
<td>void moveToInsertRow()</td>
<td>Moves the cursor to a staging position for creating a new row.</td>
</tr>
<tr>
<td>void updateRow()</td>
<td>Sends the updates for the current row to the AppServer.</td>
</tr>
</tbody>
</table>
Using extended transactions

As described earlier, by default, each row modification (row update, insertion, and deletion method) executes as a single transaction in the SmartDataObject. You can extend a transaction to include multiple rows using Batch mode, where a sendBatch() method applies a set of row updates as a single transaction (see the “Using batch mode [extension]” section on page 9–22). You also can explicitly manage a larger transaction on the AppServer remotely, using an automatic transaction.

An automatic transaction allows you to remotely create a larger transaction within which all SmartDataObject transactions become nested as subtransactions. Using an automatic transaction, the Open Client application can start, commit, and roll back a single transaction by executing methods on a special ProcObject defined in the same AppObject that provides access to the corresponding SmartDataObject. The persistent procedure underlying this ProcObject supports specific functionality and must be available for execution on the AppServer to make an automatic transaction possible. Such a transaction can encompass the entire life of the SDOResultSet. For more information on automatic transactions, see OpenEdge Application Server: Developing AppServer Applications.

Note: You should call reOpenQuery() after an explicit transaction roll-back, to maintain a cache that is consistent with the database. The reOpenQuery() method also can be desirable after committing a transaction containing inserted rows, so these rows are visible within the ResultSet. For more information, see the “Visibility of updates” section on page 9–23.

Using batch mode [extension]

Batch mode allows an Open Client to send several updates to the AppServer in one transaction. Depending on your application requirements, this might be a better update strategy than generating a transaction for each update method:

- If your application requires user feedback for each update method applied, you generally must complete a transaction for each update method and Batch mode is not appropriate.

- If your application requires that a group of updates be applied and succeed together, you must group the updates into a single large transaction. You can do this by using an automatic transaction or by using Batch mode. For information on automatic transactions, see OpenEdge Application Server: Developing AppServer Applications.

- If your application has the option of applying several updates individually or as a group, you can generally increase performance dramatically by grouping the updates into a single large transaction using Batch mode.

- If your SDOResultSet is in Stateless mode, you can apply modifications only under Batch mode.
Table 9–3 lists the SDOResultSet methods that support Batch mode.

Table 9–3:  SDOResultSet methods that support Batch mode

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void cancelBatch() [Extension]</td>
<td>Undoes all the modifications of the current batch.</td>
</tr>
<tr>
<td>void sendBatch() [Extension]</td>
<td>Sends the current batch to the AppServer.</td>
</tr>
<tr>
<td>void sendBatchAndReOpen() [Extension]</td>
<td>Sends the current batch to the AppServer, then reopens the ResultSet.</td>
</tr>
<tr>
<td></td>
<td>In Stateless mode, this method is more efficient than separately calling sendBatch() and reOpenQuery(). For more information, see the “Visibility of updates” section on page 9–23.</td>
</tr>
<tr>
<td>void startBatch() [Extension]</td>
<td>Starts a batch session. All the modifications (column updates, deletions and insertions) are maintained locally. Calling this method if you have already called it and have not called sendBatch() or cancelBatch() throws an Exception.</td>
</tr>
<tr>
<td>boolean inBatch() [Extension]</td>
<td>Returns true if the SDOResultSet object is in Batch mode (that is, you have called startBatch() and have not yet called sendBatch() or sendBatchAndReopen()).</td>
</tr>
</tbody>
</table>

Note: A row that is updated or deleted in Batch mode, and has not yet been sent to the AppServer (using sendBatch()), cannot be refreshed using refreshRow(). Also, to save your changes locally in batch mode, you must still call the row update methods (for example, insertRow() or updateRow()) even though, in this mode, your changes are not sent to the AppServer.

Visibility of updates

The visibility, from within the Open Client, of an update to the SDOResultSet object depends on whether the update is applied by:

- The Open Client itself
- An external application working on the same data source that feeds the SDOResultSet

The following rules determine the visibility of updates initiated by the Open Client:

- **Column updates** — Column updates are visible
- **Row inserts** — New rows are invisible until the ResultSet is reopened
- **Row deletion** — Row deletions are visible
Updates by external applications typically can occur on the data source that feeds the SDOResultSet. You can make any such updates visible to the Open Client application by calling reOpenQuery(), which refreshes all data in the SDOResultSet. If you do not call reOpenQuery(), however, the following rules determine the visibility of updates from external applications:

- **Column updates** — The modification of a column by another application can be made visible by calling the refreshRow() method.

- **Row inserts** — Row inserts are not visible in PREFETCH mode. In other scrolling modes, the visibility is determined by a combination of the underlying implementation of SmartDataObject and the ABL query. The guideline is that the application should not make any assumptions unless it makes an explicit call to reOpenQuery().

- **Row deletion** — Row deletions are not visible in PREFETCH scrolling mode. In other scrolling modes, the visibility is determined by a combination of the underlying implementation of the SmartDataObject and the ABL query. The guideline is that the application should not make any assumptions unless it makes an explicit call to refreshRow() or reOpenQuery(). In these cases, the specific row deletion or all the row deletions become visible, respectively.

**Miscellaneous management methods**

These methods facilitate various management functions on an SDOResultSet, and include standard JDBC 2 methods and extensions to manage SmartDataObjects.

**Standard JDBC 2 management methods**

The supported standard JDBC 2 management methods include those listed in Table 9–4.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
<td>Closes the SDOResultSet and releases the underlying SmartDataObject. Any local modifications that have not been sent to the AppServer are lost when this method is called.</td>
</tr>
<tr>
<td>int findColumn(String columnName)</td>
<td>Returns the column index.</td>
</tr>
<tr>
<td>int getFetchSize()</td>
<td>Returns the size (in number of rows) of the buffer the underlying implementation uses to fetch rows from the AppServer.</td>
</tr>
<tr>
<td>void refreshRow()</td>
<td>Gets the latest version of the current row from the database.</td>
</tr>
</tbody>
</table>

Table 9–4: Standard JDBC 2 management methods (1 of 2)
Working with SDOResultSet objects

The nonstandard SDOResultSet methods listed in Table 9–5 support a variety of features unique to SmartDataObject management.

Note: The rowId value referenced in the following methods is equivalent to the value returned by the ABL ROWID function and the SDOResultSet.getRowIdentity() method. For more information on ROWID, see *OpenEdge Getting Started: ABL Essentials* (or the *OpenEdge DataServer Guide* for any DataServer accessed by the SmartDataObject).

### Table 9–4: Standard JDBC 2 management methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDOResultSetMetaData.getMetaData()</td>
<td>Returns the SDOResultSetMetaData object, which is an extension of the JDBC ResultSetMetaData object.</td>
</tr>
<tr>
<td>boolean wasNull()</td>
<td>Indicates if the last fetched value from a getDatatype() method is equal to the Unknown value (?). This method is especially useful for methods that return a primitive data type, as in this example where SDOResults is an SDOResultSet:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>int cValue;</td>
<td></td>
</tr>
<tr>
<td>cValue = SDOResults.getInt(&quot;QUANTITY&quot;);</td>
<td></td>
</tr>
<tr>
<td>if (SDOResults.wasNull())</td>
<td></td>
</tr>
<tr>
<td>return(-1);</td>
<td></td>
</tr>
<tr>
<td>Because a primitive data type cannot hold the null value (the Java object equivalent to the Unknown value (?)), wasNull() provides the equivalent for the int value returned by getInt(). In this example, the method fragment returns the value –1 for a Unknown value (?) because valid integer values for the QUANTITY column must be positive.</td>
<td></td>
</tr>
</tbody>
</table>

### SmartDataObject management methods [extension]

The nonstandard SDOResultSet methods listed in Table 9–5 support a variety of features unique to SmartDataObject management.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean absolute(String rowId) [Extension]</td>
<td>Repositions the cursor at the row specified by rowId.</td>
</tr>
<tr>
<td>String getQuery() [Extension]</td>
<td>Returns the ABL query that populated this SDOResultSet object.</td>
</tr>
<tr>
<td>String getRowIdentity() [Extension]</td>
<td>Gets the string representation of the ROWID value for the current row that can be used as input to reOpenQuery(String rowId), absolute(String rowId), and SDOParameters.setRowIdentity(String rowId).</td>
</tr>
</tbody>
</table>
Table 9–5: Nonstandard SDOResultSet management methods  (2 of 2)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getSDOInterface()</code> [Extension]</td>
<td>Provides direct access to the underlying ABL methods of the SmartDataObject that are efficient for remote SmartDataObject access. The returned SDOInterface is a ProcObject that includes all relevant public methods provided by the SmartDataObject. Thus, you have full access to these SmartDataObject features without any limit imposed by the SDOResultSet interface. If the SDOResultSet is stateless, calling <code>getSDOInterface()</code> creates a remote SmartDataObject procedure you can delete only by calling <code>releaseSDOInterface()</code>. Calling <code>releaseSDOInterface()</code> on a non-stateless SDOResultSet has no effect. See the “Note on SDOInterface method” section on page 9–26.</td>
</tr>
<tr>
<td><code>releaseSDOInterface()</code> [Extension]</td>
<td>On a stateless SDOResultSet, releases the remote SmartDataObject procedure. Calling <code>releaseSDOInterface()</code> on an SDOResultSet that is not Stateless has no effect. This method is used in conjunction with <code>getSDOInterface()</code>.</td>
</tr>
<tr>
<td><code>reOpenQuery()</code> [Extension]</td>
<td>Reopens the SmartDataObject query and positions the cursor before the first row. This method refreshes all the data in the SDOResultSet, makes all of the newly inserted rows visible, and removes all of the deleted rows. Any local modifications that you have not sent to the AppServer before you call this method are lost.</td>
</tr>
<tr>
<td><code>reOpenQuery(String rowId)</code> [Extension]</td>
<td>Works like <code>reOpenQuery()</code>, but the first row returned is the one specified by <code>rowId</code>. A null <code>rowId</code> is equivalent to <code>reOpenQuery()</code>. Usually it is more efficient to call <code>reOpenQuery(String rowId)</code> than to first call <code>reOpenQuery()</code> and then call <code>absolute(String rowId)</code> to reposition on the specified row. <strong>Note</strong>: If the SDOResultSet is in <code>SDOScrollingMode.PREFETCH</code> mode, you cannot get the rows before this <code>rowId</code>. For more information on <code>SDOScrollingMode.PREFETCH</code>, see the “Understanding SDOResultSet scrolling modes” section on page 9–13.</td>
</tr>
</tbody>
</table>

Note on SDOInterface method
The actual SmartDataObject methods supported through SDOInterface include:

- `serverSendRows`
- `serverCommit`
- `initializeObject`
- `setQueryWhere`
- `getQueryWhere`
- `setQuerySort`
- `openQuery`
- `closeQuery`
- `columnProps`
- `fetchMessages`
- `getUpdatableColumns`
- `addQueryWhere`
- `assignQuerySelection`
- `getTables`
- `getObjectVersion`
- `batchServices`

For more information on these SmartDataObject methods, see *OpenEdge Development: ADM and SmartObjects* and *OpenEdge Development: ADM Reference*.

**Note:** A typical application does not require direct access to the `SDOInterface`. Direct access to SmartDataObject methods requires expert knowledge of SmartDataObject internals. In some cases, mixing direct calls to the SmartDataObject with calls through the `SDOResultSet` interface can leave the `SDOResultSet` in an unexpected state. **Use the `SDOInterface` directly only after carefully analyzing the alternatives.**

### Meta data methods

Meta data about the `SDOResultSet` is exposed through the `com.progress.open4gl.SDOResultSetMetaData` interface. An object that implements `SDOResultSetMetaData` is returned using the `SDOResultSet.getMetaData()` method. The interface, `com.progress.open4gl.SDOResultSetMetaData`, is an extended subset of the standard JDBC 2 `java.sql.ResultSetMetaData` interface.

`SDOResultSetMetaData` works with both JDK 1.1.x and JDK 1.2; therefore, much of the Java 1.2 functionality (standard JDBC ResultSetMetaData interface) is supported even with JDK 1.1.x.

#### JDBC 2 meta data methods

The supported standard JDBC 2 methods include the ones listed in Table 9–6.

**Table 9–6: Standard JDBC meta data methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getColumnCount()</code></td>
<td>The number of columns in the <code>SDOResultSet</code></td>
</tr>
<tr>
<td><code>int getColumnDisplaySize(int columnIndex)</code></td>
<td>The maximum display size width of this column in characters</td>
</tr>
<tr>
<td><code>String getColumnType(int columnIndex)</code></td>
<td>The SQL type of the column</td>
</tr>
<tr>
<td><code>String getColumnType(int columnIndex)</code></td>
<td>The ABL data type name of the column</td>
</tr>
<tr>
<td><code>String getColumnName(int columnIndex)</code></td>
<td>The name of the column</td>
</tr>
<tr>
<td><code>String getColumnLabel(int columnIndex)</code></td>
<td>The ABL dictionary label for this column</td>
</tr>
<tr>
<td><code>String getColumnTypeName(int columnIndex)</code></td>
<td>The ABL data type name of the column</td>
</tr>
</tbody>
</table>
Nonstandard meta data methods [extension]

SDOResultSetMetaData extensions to the standard interface include the following features:

- Methods that expose meta data information that exists in the schema of the OpenEdge database but does not have an equivalent standard (for example, getColumnInitialValue())
- Methods that expose ABL-to-Java mapping information (for example, getColumnJavaTypeName())

**Note:** The SDOResultSetMetaData interface also hides SmartDataObject system fields. SmartDataObject system fields are fields (such as changedFieldsList) that are used only by the SmartDataObject implementation code. They have no meaning for the Open Client and should not be accessed directly.

Table 9–7 lists the supported nonstandard SDOResultSetMetaData methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getColumnFormat (int column) [Extension]</td>
<td>The ABL dictionary display format</td>
</tr>
<tr>
<td>String getColumnInitialValue (int column) [Extension]</td>
<td>The ABL dictionary initial value as a string</td>
</tr>
<tr>
<td>String getColumnJavaTypeName (int column) [Extension]</td>
<td>The Java type name</td>
</tr>
<tr>
<td>int getColumnProType (int column) [Extension]</td>
<td>The ABL data type number. The return value corresponds to one of the class constants defined in com.progress.open4gl.Parameter. For more information, see Chapter 4, “Passing Parameters.”</td>
</tr>
</tbody>
</table>
Table 9–7: Nonstandard SDOResultSetMetaData methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getColumnValExp(int column) [Extension]</td>
<td>The ABL dictionary validation expression</td>
</tr>
<tr>
<td>String getColumnValMsg(int column) [Extension]</td>
<td>The ABL dictionary validation message</td>
</tr>
</tbody>
</table>
Multi-threading

You cannot access a single SDOResultSet from more than one thread at a time. The results of concurrent access to a single SDOResultSet by more than one thread are undetermined.
Errors and exceptions

Two types of errors can occur working with SDOResultSet, depending on the category of method that is called:

- Errors that are caused by an application bug, such as an attempt to position the cursor at a negative row number. This type of error causes a java.sql.Exception to be thrown, with different messages for different errors.

- Errors that can occur during an otherwise normal session, such as an update failure due to a concurrency control collision on the AppServer. This type of error causes a specific SDOResultSet Exception to be thrown, which allows you to handle the error programmatically.

This section describes the SDOResultSet exception hierarchy and typical errors by method category.

SDOResultSet exceptions

All SDOResultSet and SDOResultSetMetaData methods throw com.progress.open4gl.ProSQLException, which is the root of more specific exceptions. Table 9–8 shows the full exception hierarchy. Indentation in the table indicates the level of the hierarchy.

Table 9–8: Output ResultSet exceptions for SDOResultSet

<table>
<thead>
<tr>
<th>Exception class hierarchy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.SQLException</td>
<td>Root exception for all JDBC 2 exceptions</td>
</tr>
<tr>
<td>com.progress.open4gl.ProSQLException</td>
<td>Root exception for all SDOResultSet exceptions</td>
</tr>
<tr>
<td>com.progress.open4gl.DeletedRowAccessException</td>
<td>Attempted to access a deleted row</td>
</tr>
<tr>
<td>com.progress.open4gl.SDOModificationException</td>
<td>Failed attempt to modify an existing row</td>
</tr>
</tbody>
</table>

Two exceptions from this hierarchy allow you to handle normal program conditions:

- **DeletedRowAccessException** — This exception is thrown when there is an attempt to access a deleted row. To prevent it, the application can use the boolean rowDeleted() method before trying to access a row.

- **SDOModificationException** — This exception is thrown when the back-end SmartDataObject fails to insert a new row, update an existing row, or delete an existing row. If the row operation is one of many in a larger transaction, each update, insert, or delete failure causes a message to be added to a list of messages held by the thrown SDOModificationException.
You can access all the error information provided by SDOModificationException using the following methods.

This method returns the list of fields (if any) related to this error message:

Syntax

```java
String getFields()
```

This method returns the name of the table (if any) related to this error message:

Syntax

```java
String getTable()
```

This method returns the text of the current error message:

Syntax

```java
String getText()
```

This method positions the cursor on the next error message:

Syntax

```java
void nextMessage()
```

For more information on the SDOModificationException class and its methods, see the SDOModificationException.html file in OpenEdge-install-directory\java\doc.

Typical errors by method category

Calling the following categories of methods can result in a corresponding set of errors:

- **Navigation methods** — The typical errors in this category include attempts to navigate to rows that do not exist. If the row has been deleted, this error throws the DeletedRowAccessException.

- **Insert, delete, and update row methods** — The typical errors in this category include:
  - Concurrency control errors (the row was updated by another user, for example)
  - Validation errors (the update was rejected by some AppServer validation routine)
  - Trying to update a row while the cursor is not in an appropriate position (for example, calling insertRow() while the cursor is not in an insert row position)

The first two types of errors throw the SDOModificationException.
• **Get and update column value methods** — The typical errors in this category include:
  – Trying to convert a value to the wrong data type (for example, trying to get an integer column using a `getDateTime()` method)
  – Trying to update a column with the wrong data type (for example, trying to set a date object to an integer column)
  – Trying to get or set data when the cursor does not point to a valid row

• **Meta data methods** — The typical errors in this category occur when trying to get meta data information about a column that does not exist.
Choosing JDBC ResultSets or SDOResultSets to access OpenEdge data

Progress Software Corporation supports a JDBC interface directly to the OpenEdge database. Some applications might benefit from using the direct JDBC interface to OpenEdge rather than SDOResultSets, and others might benefit more from using SDOResultSets.

JDBC access to the OpenEdge database should be used in the following cases:

- There is no ABL business logic (such as ABL triggers) that must be used
- Two-tier access (client/server) is sufficient
- The client side application requires full JDBC support

SmartDataObject ResultSets should be used in the following cases:

- ABL business logic is part of the AppServer application
- N-tier configuration is required for performance or other reasons
- Higher than JDBC 1 level of access is required. SDOResultSet, which is based on JDBC 2, is easier to use and does not require knowledge of SQL
Accessing the sample application

The Documentation and Samples directory contains a sample SmartDataObject application, including a small Java Open Client application and the generated SmartDataObject ready to deploy. For more information, see the readme.htm file in the following samples directory:

```
Doc_and_Samples_Install/src/samples/open4gl/SmartDataObject/
```

For more information on accessing and installing these samples, see the information on sample Java client applications in Chapter 1, “Configuring and Deploying Java Open Client Applications.”
You can build a Java applet that runs as an Open Client; that is, the applet can make proxy method calls to remote AppServer procedures from the Web browser environment. However, you must include additional code to allow the Web browser to accept the Open Client applet for execution. The Open Client supports Java applets by using an HTML page to distribute the appropriate files. There are a couple of factors you must consider:

- The Open Client applet obtains the HTTP Basic authentication and Proxy Web server information from the end user and then passes the information to the Open Client. You cannot, however, rely on the browser to provide the user interaction; your applet must perform this task. You also might have to access the Proxy Web server configuration information from the browser. In this case, the applet also needs to obtain the Proxy Web server information from the end user.

- Open Client applets can run on both a company intranet and the Internet. As a result, you need to sign your Java code with a digital certificate, and your applet must get permission to perform a network connection.

This chapter contains the following sections:

- Asking for permission in an applet
- Signing an Open Client applet
- Adding your applet to an HTML page
- Sample applet
Asking for permission in an applet

After creating and digitally signing your applet distribution files, you must write code that requests permission to establish network connections to servers other than the Web server from which your applet was downloaded. This is necessary because often the OpenEdge servers you are connecting to are on systems other than the ones from which the applet was loaded.

Although the procedure for asking for permission is different for the Microsoft Internet Explorer and Netscape browsers, it is possible to support both browsers in the same applet, due to Java's ability to dynamically load classes.

Asking for permission in Microsoft Internet Explorer

Asking for permission in Internet Explorer requires that you add specific code to your applet.

To add a code to your applet:

1. Down load the security classes for Microsoft, by putting the security classes on your CLASSPATH and adding the following import statement to your applet code:

   ```java
   import com.ms.security.*;
   ```

   **Note:** The Microsoft security classes are not available as a standalone package. You can get these security classes by downloading the latest Microsoft Java SDK from [http://www.microsoft.com](http://www.microsoft.com).

2. Insert the following code into your applet, just before creating your Java proxy AppObject:

   ```java
   // Assert Net IO privileges
   try
   {
     if (Class.forName("com.ms.security.PolicyEngine") != null)
       PolicyEngine.assertPermission(PermissionID.NETIO);
   }catch(ClassNotFoundException e)
   {
     System.out.println("IE Security Manager not found");
   }
   ```

This code searches for the class `com.ms.security.PolicyEngine`. When the class is found, the code calls the static method `assertPermission`, asking for NETIO permission. This method verifies that the user selected yes in the security dialog box when the applet was downloaded. If yes was selected, the user can connect to the AppServer.

With Microsoft Internet Explorer, a user is prompted only once when the code is first downloaded.
3. If you use HTTPS, you also might need to obtain file I/O privileges from the Microsoft VM security system to read the files containing the root digital certificates. If this is the case, add the following line to your code, after the line of code that request NETIO privileges:

```java
PolicyEngine.assertPermission(PermissionID.FILEIO);
```

Permissions are valid only for a particular method call; that is why you should insert this code just before constructing your Java proxy AppObject. For instance, if you put this code in its own method, you would no longer have the permission when your method returns.

For more information on Microsoft’s trust based security model, see the Microsoft SDK for Java documentation.

**Asking for permission in Netscape version 6**

Asking for permissions in Netscape version 6 or when using a Java plug-in in your Microsoft Internet Explorer requires that you change the Java 2 security policy. The Java 2 security policy file is located in the JRE's `lib\security` subdirectory. It might be combined with a per-user policy file located in the user’s home directory.

The JRE’s file is named `java.policy`, and the version in a user’s home directory is named `.java.policy`. Changing the JRE policy file grants permissions to every user on the system, while changing the policy file in a user’s home directory affects only that user. Progress Software Corporation recommends changing the user’s home directory on individual machines. For more information about Java 2 security policy files, refer to the Java 2 Security guides.

To support HTTP and HTTPS, the Open Client applet requires additional network connection privileges. The Java permission is named `java.net.SocketPermission`, and it takes these arguments:

- The fully qualified DNS address and port (usually port 80) of the Web server for the AppServer Internet Adapter (AIA)
- The network actions `connect`, `resolve`

The full entry is:

```java
permission java.net.SocketPermission host:80, connect, resolve
```

When using HTTPS in your Open Client applet, you also might need an additional Java permission to read the root digital certificate files, to validate the identity of the Web server for the AIA. If this is the case, you must add file read permissions that provide full read ability as shown in the following entry:

```java
permission java.io.FilePermission *, read;
```

The Java permissions should be placed in the Java 2 policy file grant blocks. One permission grants access to only your Open Client applet code; the other permission, to only the Open Client Runtime.
The following shows some sample code for these permission grants:

```java
grant codebase http://applicationhost.acme.com/aia/jars-{
    permission java.net.SocketPermission
    https://aia.acme.com/aia/Aia1:443,connet,resolve;
    permission java.io.FilePermission *,read;
};
grant codebase http://applicationhost.acme.com/aia/applets/-{
    permission java.net.SocketPermission *, read;
};
```

In this sample:
- The o4g1rths1.jar runtime and root digital certificates are located in the applicationhost Web server aia/jar path
- The Open Client applet is located in the applicationhost Web server aia/applets path
- The AIA is located in the AIA Web server aia/Aia1 path

**Supporting both browsers in one applet**

The following code works for both Microsoft Internet Explorer and Navigator in a single applet:

```java
import com.ms.security.*;
import netscape.security.PrivilegeManager;
try
{
    // Assert Net IO privileges
    System.out.println("Checking for Internet Explorer Security Manager");
    if (Class.forName("com.ms.security.PolicyEngine") != null)
    PolicyEngine.assertPermission(PermissionID.NETIO);
}
catch(ClassNotFoundException e)
{
    System.out.println("IE Security Manager not found");
}
try
{
    System.out.println("Checking for Netscape Security Manager");
    if (Class.forName("netscape.security.PrivilegeManager") != null) {
        try {
            PrivilegeManager.enablePrivilege("UniversalConnect");
        } catch(Exception ex) {
            System.out.println("Netscape threw exception: " +
            ex.toString());
        }
    }
}
catch(ClassNotFoundException e)
{
    System.out.println("Netscape Security Manager not found");
}
```

To test this code, make sure both security manager classes are in your CLASSPATH.
Signing an Open Client applet

When preparing your Java applet for download with a digital certificate, you must sign your code.

To sign the code:

1. Generate your Java proxy. For more information, see OpenEdge Development: Open Client Introduction and Programming.

2. Code your applet.

3. Create a .cab file for Internet Explorer browsers and a .jar file for Netscape browsers for the Java applet code and proxy you created that includes a digital signature to download the applet. For instructions about completing these tasks, see the Microsoft SDK for Java and the Netscape Object Signing Tools documentation.

4. To sign your code, you need a code signing certificate. While both browsers allow you to create a test digital certificate, you need a real digital certificate to distribute your application. If you distribute your application over the Internet, you usually get a digital certificate from a public PKI provider like VeriSign. If you distribute your application on a company intranet, you can set up a digital certificate server and create your own certificates.

The steps for setting up and operating your own certificate server are not covered in this document. You can get information about this from the Microsoft or Netscape Web sites.

Note: Although both Microsoft and Netscape use a standard digital certificate format, the disk storage formats used in the respective signing tools are not interoperable. As a result, you must get separate digital certificates if you want to support both Microsoft and Netscape browsers.

OpenEdge provides the Open Client Runtime packages in digitally signed .cab and .jar files, so you can distribute these directly without additional work. These .cab and .jar files are located in the Java directory under your OpenEdge installation directory and have filenames beginning with o4glrt. For more information about which of these files to choose for your Java applet see the information on selecting an Open Client Runtime package in OpenEdge Development: Open Client Introduction and Programming.

When you embed an Open Client Runtime package in a Web page, a verification dialog box automatically opens when a user accesses the page. The user has the option of accepting or not accepting the installation of the Open Client Runtime. If the user does not accept, the Open Client is not installed, and your Open Client applet cannot be run.
Adding your applet to an HTML page

To add your applet to a Web page, you must add the correct HTML tags to allow a browser to download and run your applet. Although how you do this differs between Microsoft Internet Explorer and Navigator, it is possible to have one Web page that supports both browsers.

Microsoft Internet Explorer

Internet Explorer allows you to download and install Java libraries on the user's computer; therefore, the Open Client Runtime can be downloaded and installed once. Internet Explorer also supports versioning on these Java libraries. This allows you to update your users automatically when the Open Client Runtime is updated.

To install the Open Client Runtime from a Web page, you add an OBJECT tag to your Web page as shown in the following code:

```html
<!-- This checks for installation of the Open4GL runtime code, and will install it if it isn't there or is an older version -->
<OBJECT CLASSID="CLSID:E79DA910-15D0-11d2-B591-00C04FD4A860"
CODEBASE="/cabs/o4glrt.cab#Version=9,1,1115,0">
<applet code=Open4GLDSO.class width=0 height=0 id=datactl MAYSCRIPT=true>
<PARAM NAME=cabbase VALUE="/cabs/o4gldso.cab"
</applet>
```

When you do this, you must modify the CODEBASE attribute to the location of the Open Client Runtime packages on the Web server from which the .cab file was distributed. When Internet Explorer encounters this tag, it looks in the user's Windows registry for an entry with the specified class ID and version. Then, it looks to see if the o4glrt.cab file exists. If either of these above conditions is not met, the .cab file is downloaded.

The Microsoft security classes are not available as a standalone package. You can get these security classes by downloading the latest Microsoft Java SDK from http://www.microsoft.com.

Each Open Client Runtime .cab file has a unique class ID. This allows your application to change the types of network protocols it uses dynamically the next time the Web page is run. The version number, which changes for each release, is supplied by OpenEdge in the o4glrt.cab file's open4glrt.ini file.

Once the .cab file is downloaded, a dialog box opens, advising the user that this code needs to be trusted and is signed by Progress Software Corporation. If the user selects OK, the .zip file is installed in the Windows/Java/TrustedLib directory, and all required registry keys are added or updated.

If the user selects OK, the .zip file also is added dynamically to the TrustedClassPath. This entry is volatile, and if the user restarts Internet Explorer, the .zip file is no longer included in its CLASSPATH. Therefore, any time you have a Web page that uses the Open Client Runtime, you should include that OBJECT tag. Once the code is installed, there is no further action required by users based on that OBJECT tag, even if they restart Internet Explorer. Since the runtime is installed on the TrustedClassPath, only trusted code can access it. Since you need to be trusted to open a connection to an OpenEdge server, you are required to package your applet and proxies into a signed .cab file. Once this is done, you can add an applet tag like the following:

```html
<applet code=Open4GLDSO.class width=0 height=0 id=datactl MAYSCRIPT=true>
<PARAM NAME=cabbase VALUE="/cabs/o4gldso.cab" />
</applet>
```
The code attribute specifies your applet's class name that must be contained in the .cab file specified with the cabbase PARAM. The value of the cabbase PARAM should point to the .cab file's path that contains the applet and proxy on the distribution Web server.

Internet Explorer looks for the cabbase, and if it finds it, downloads the specified .cab file. Since this is a signed .cab file, a dialog box opens and displays information from your code signing certificate. Your Java code is saved in Internet Explorer's volatile cache, so if the user restarts the Internet Explorer, your .cab file is downloaded again.

**Navigator**

When adding your applet to a Web page for Navigator, you must consider the following:

- Earlier Navigator browsers did not support downloading and installing Java libraries. Therefore, every time a user accesses the Web page for the first time in a session, the applet, proxy, and the Open Client Runtime packages are downloaded.

- Navigator versions prior to 4.05 did not allow an applet to support multiple .jar files. If you need to support a version of Navigator prior to 4.05, you must repackage everything into a single .jar file and sign it.

For Navigator version 4.05 or later, you can use the following applet tag to embed your applet in a Web page:

```html
<applet code="AppletTest.class"
codebase="/cabs" name="AppletTest" width="320" height="240"
archive="apptest.jar,o4glrt.jar">
</applet>
```

In the above example, the applet code and proxies are in the signed apptest.jar file, and the Open Client Runtime package is in the o4glrt.jar file. When Navigator encounters this tag, it downloads both .jar files. When your applet requests UniversalConnection privileges, a dialog box opens and prompts your user to grant the privilege.
Supporting both browsers with the same page

To support both Internet Explorer and Navigator with the same Web page, include code like the following:

```xml
<Object CLASSID="CLSID:890DFF4B-57E8-11d2-807f-00C04FD4F937" CODEBASE="/cabs/o4glrt.cab#Version=9,0,1066,0"> 
  <applet 
    code="AppletTest.class" 
    codebase="/cabs" name="AppletTest" width="320" height="240" 
    archive="apptest.jar,o4glrt.jar">
    <param NAME=cabbase VALUE="/cabs/apptest.cab"/>
  </applet>
</object>
```

When using the code in this example, be aware of the following issues that arise for the different browsers:

- Navigator ignores and does not support the OBJECT tag and the cabbase PARAM.
- Internet Explorer supports the OBJECT tag and the cabbase PARAM and looks for classes in the .cab file specified in the cabbase PARAM of the OBJECT tag before using the .jar files specified in the applet tag's archive attribute.
- Internet Explorer does not support Netscape's code signing technology, as a result, any code found in .jar files is untrusted and limits the servers to which the browser can connect.
- Navigator does not support .cab files and, as a result, it ignores the cabbase PARAM; therefore, you need to specify both .cab and .jar files.

For more information about supporting both Internet Explorer and Navigator with the same Web page, see the Microsoft Knowledge Base article Q179652.
Sample applet

The Documentation and Samples directory contains a sample Java Open Client applet. You can find it in the following directory:

```
Doc_and_Samples_Install/src/samples/open4gl/java/applet/
```

For more information on accessing and installing these samples, see the information on sample Java client applications in Chapter 1, "Configuring and Deploying Java Open Client Applications."
The Open Client Java OpenAPI is a generic set of Java classes that can be used instead of the generated Java proxies produced by ProxyGen. Without proxies, you must do all of the setup work in your client code to access an application service on the AppServer. You must know the procedure names, including relative path information (for the equivalent ABL RUN statement), the number of parameters along with their type, and the return type for user-defined functions. When using the OpenAPI, there is no compile time checking so all function prototype error checking is done at runtime.

This chapter describes the Java OpenAPI in these sections:

- Overview of Java OpenAPI classes
- Connecting to the AppServer
- Running procedures and user-defined functions
- Setting up parameters
- Handling returned values
- Sample Java OpenAPI code
Overview of Java OpenAPI classes

Table 11–1 shows a summary of the classes that support the Java OpenAPI.

Table 11–1: Java OpenAPI Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.progress.open4gl.javaproxy.OpenAppObject</td>
<td>For accessing external ABL procedures in an application service using the OpenAPI</td>
</tr>
<tr>
<td>com.progress.open4gl.javaproxy.OpenProcObject</td>
<td>For accessing an ABL persistent procedure using the OpenAPI</td>
</tr>
<tr>
<td>com.progress.open4gl.javaproxy.ParamArray</td>
<td>An object containing an array for passing parameters to an ABL procedure or user-defined function</td>
</tr>
<tr>
<td>com.progress.open4gl.javaproxy.ParamArrayMode</td>
<td>Constants for specifying the mode of an ABL procedure or user-defined function parameter (INPUT, INPUT-OUTPUT, or OUTPUT)</td>
</tr>
<tr>
<td>com.progress.open4gl.Parameter</td>
<td>Constants for specifying the ABL data type of a procedure or user-defined function parameter, return type, or temp-table field</td>
</tr>
<tr>
<td>com.progress.open4gl.ProDataGraphMetaData</td>
<td>For defining the schema of an ABL ProDataSet or temp-table mapped to a ProDataGraph</td>
</tr>
<tr>
<td>com.progress.open4gl.ProDataObjectMetaData</td>
<td>For defining the schema of an ABL temp-table contained within a ProDataGraph (mapped to a ProDataObject)</td>
</tr>
<tr>
<td>com.progress.open4gl.ProDataRelationMetaData</td>
<td>For defining a ProDataSet data-relation contained within a ProDataGraph</td>
</tr>
<tr>
<td>com.progress.open4gl.dynamicapi.ProResultSetMetaDataImpl</td>
<td>For defining the schema of an ABL temp-table mapped to an SQL ResultSet</td>
</tr>
</tbody>
</table>

1. For complete description of the ProDataGraph class and its associated classes, see Chapter 5, “Accessing ABL ProDataSets.”

2. For complete information on temp-table parameters mapped to an SQL ResultSet, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

The sections of this chapter describe how to use these classes to access an AppServer using the Java OpenAPI.
Connecting to the AppServer

To establish a connection to an AppServer using the Java OpenAPI, you must instantiate a \texttt{com.progress.open4gl.javaproxy.OpenAppObject}. You can create an \texttt{OpenAppObject} using one of the following constructors:

**Syntax**

\begin{verbatim}
public OpenAppObject(Connection connectObj, String appservice)
public OpenAppObject(String url, String userid, String password,
                     String appserver-info, String appservice)
public OpenAppObject(String userid, String password,
                     String appserver-info, String appservice)
public OpenAppObject(String appservice)
\end{verbatim}

\texttt{connectObj}

Specifies a \texttt{com.progress.open4gl.javaproxy.Connection} object, which defines connection parameters to access the AppServer. For more information on this object, see Chapter 3, “Connecting to an AppServer.”

\texttt{appservice}

Specifies the name of the application service provided by the AppServer.

\texttt{url}

Specifies the URL to connect to an Appserver URL. Defaults to \texttt{AppServer://localhost:5162/appService}. For more information on the AppServer URL connection parameter format and default connection information, see the sections on connecting to an AppServer using a URL in \textit{OpenEdge Application Server: Developing AppServer Applications}.

\texttt{userid}

Specifies a user ID required to access the AppServer.

\texttt{password}

Specifies a password required to access the AppServer.

\texttt{appserver-info}

Specifies information required by the application service at connection time.

You can set a number of session and run-time properties to manage access to an application service. For example, to access a session-free AppServer using the Java OpenAPI, you must set the proxy property \texttt{PROGRESS.Session.sessionModel} to 1. For more information on setting this and other session properties see Chapter 7, “Accessing Proxy Properties.”
Running procedures and user-defined functions

After connecting to an AppServer and creating a com.progress.open4gl.javaproxy.OpenAppObject, you can invoke application service code as follows:

1. Run any external non-persistent or persistent procedure that is available on the OpenAppObject. (Running a persistent procedure instantiates an OpenProcObject.)

2. Run any available internal procedure or user-defined function on an instantiated OpenProcObject.

Before running any procedure or user-defined function, you must set up any parameters and return types required by the procedure or user-defined function. For more information, see the “Setting up parameters” section on page 11–7. You can then pass the parameters to the procedure or user-defined function and manage them according to their modes (INPUT, INPUT-OUTPUT or OUTPUT). For more information, see the “Passing parameters” section on page 11–33.

After running any procedure or user-defined function, you must handle any output (returned values) or error conditions. For more information, see the “Handling returned values” section on page 11–37.

The following sections describe:

- Running a non-persistent procedure on an OpenAppObject
- Running a persistent procedure (OpenProcObject) on an OpenAppObject
- Running an internal procedure or user-defined function on an OpenProcObject

For examples that show how to run application service code, see the “Sample Java OpenAPI code” section on page 11–39.
Running procedures and user-defined functions

Running a non-persistent procedure on an OpenAppObject

Once you create your com.progress.open4gl.javaproxy.OpenAppObject instance, you can run any non-persistent procedure on the connected AppServer using the following method on the OpenAppObject:

Syntax

```java
public void runProc(String procName, ParamArray paramArray)
    throws Open4GLEException, RunTime4GLEException,
             SystemErrorException
```

`procName`

Specifies the name of the procedure to run, including any path relative to the PROPATH setting for the AppServer.

`paramArray`

Specifies a com.progress.open4gl.javaproxy.ParamArray that holds the parameters for the procedure. For more information, see the “Setting up a parameter array” section on page 11–7.

Running a persistent procedure (OpenProcObject) on an OpenAppObject

Once you create your com.progress.open4gl.javaproxy.OpenAppObject instance, you can run any persistent procedure on the connected AppServer using the following method on the OpenAppObject:

Syntax

```java
public OpenProcObject createPO(String procName, ParamArray paramArray)
    throws Open4GLEException, RunTime4GLEException,
             SystemErrorException
```

`procName`

Specifies the name of the procedure to run, including any path relative to the PROPATH setting for the AppServer.

`paramArray`

Specifies a com.progress.open4gl.javaproxy.ParamArray that holds the parameters for the procedure. For more information, see the “Setting up a parameter array” section on page 11–7.

This method returns a com.progress.open4gl.javaproxy.OpenProcObject, which you can use to run internal procedures and user-defined functions provided by the persistent procedure.
Running an internal procedure or user-defined function on an OpenProcObject

Once you obtain your `com.progress.open4gl.javaproxy.OpenProcObject` instance, you can run any internal procedure or user-defined function defined in the object. However, if you are calling a user-defined function, you must identify the return type of the function before calling it. For more information, see the “Defining the return type for a user-defined function” section on page 11–19. For information on accessing the return value after calling the function, see the “Accessing user-defined function return values” section on page 11–38.

Run the internal procedure or user-defined function using the following method on the `OpenProcObject`:

**Syntax**

```java
public void runProc(String procName, ParamArray paramArray)
    throws Open4GLEException, RunTime4GLEException, SystemErrorException
```

*procName*

The name of an internal procedure or user-defined function provided by the `OpenProcObject`.

*paramArray*

Specifies a `com.progress.open4gl.javaproxy.ParamArray` that holds the parameters for the internal procedure or user-defined function. For more information, see the “Setting up a parameter array” section on page 11–7.
Setting up parameters

Before running an application service procedure or user-defined function, it might require parameters that you must pass. Before passing the parameters, you must set them up in two basic steps using the Java OpenAPI.

To pass parameters to an application service procedure or function:

1. Create a variable for each parameter.
2. Add each variable to a parameter array as required for the procedure or function you are calling.

The following sections describe how to complete these steps:

- Creating variables for parameters
- Setting up a parameter array
- Adding parameters using data type-specific methods
- Adding parameters using a generic method
- Defining the return type for a user-defined function
- Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet
- Defining the schema for a ProDataSet parameter
- Defining the schema for temp-tables in a ProDataSet
- Defining the schema for a temp-table parameter mapped to a ProDataGraph

Creating variables for parameters

Before passing a parameter, you must create a variable of the correct Java data type. If you want to pass the ABL Unknown value (?) (null in Java) for intrinsic types, you must use the corresponding object. The Progress to Java mappings are the same as the ones defined for passing parameters to Open Client proxy methods. For more information on the supported mappings between ABL and Java data types, see Chapter 4, “Passing Parameters.”

Setting up a parameter array

You must place all parameters for a procedure (external, persistent, and internal) or user-defined function into a com.progress.open4gl.javaproxy.ParamArray object before running the procedure or user-defined function.

Creating a parameter array

You can create a ParamArray using the following constructor:

Syntax

```
public ParamArray(int numParams)
```
numParams

Specifies the total number of parameters (including all INPUT, INPUT-OUTPUT, and OUTPUT parameters).

You can add each parameter to the ParamArray using a set of methods that add a parameter based on the data type of the parameter or a single generic add parameter method. You must identify the parameter position as well as the mode of the parameter. For temp-tables and ProDataSets, you must identify the meta data. You must also pass in an initial value for input and input-output parameters.

If you want to reuse the ParamArray for another call:

- The number of parameters for the next call must match the number of parameters in the ParamArray object
- You must clear any current values in the ParamArray using the clear() method as follows:

```java
// Create the ParamArray
ParamArray parms = new ParamArray(1);
.
.
// Clear for reuse
parms.clear();
```

**Adding parameters using data type-specific methods**

You can set each parameter using one of the methods on the com.progress.open4gl.javaproxy.ParamArray class that is specified for the corresponding ABL data type. ABL data types that are mapped to intrinsic data types in Java have two sets of methods:

- One for the intrinsic type, which cannot be set to the Unknown value (?) (null)
- One for an Object type that can accept the Unknown value (?)

Methods to add arrays of each data type are also provided.

**General syntax for data type-specific add parameter methods**

The general syntax of these methods is as follows:

**Syntax**

```java
public void addProgressType(int position, DataType value, int mode,
                           [ int extentValue | MetaType metaData ]
   throws Open4GLException
```

*ProgressType*

Indicates the ABL data type of the parameter, such as Decimal in addDecimal(), for adding a parameter of data type DECIMAL.
Position

Specifies a 0-based index indicating the parameter position.

DataType

Specifies the Java data type of the parameter, for example, `java.math.BigDecimal` or `java.math.BigDecimal[]`.

Value

Specifies a variable that contains the value of the parameter, or `null` for an `OUTPUT` parameter.

Mode

Specifies the ABL mode (passing direction) of the parameter. This can be one of the following constant values:

- `ParamArrayMode.INPUT`
- `ParamArrayMode.INPUT_OUTPUT`
- `ParamArrayMode.OUTPUT`

ExtentValue

Specifies the extent of an array for methods that add an array parameter (where `ProgressType` is appended with `Array`, as in `AddDecimalArray()`).

MetaType

Specifies one of the following classes for methods that add a temp-table or ProDataSet parameter:

- `ProResultSetMetaDataImpl` — Provides the schema information for temp-table parameters that are mapped to a `java.sql.ResultSet`. For more information on defining this schema, see the “Defining the schema for a temp-table parameter mapped to a `java.sql.ResultSet`” section on page 11–22.

- `ProDataGraphMetaData` — Provides the schema information for ProDataSet parameters and temp-table parameters that are mapped to a `com.progress.open4gl.ProDataGraph`. For more information on defining the schema for ProDataSet parameters, see the “Defining the schema for a ProDataSet parameter” section on page 11–24. For more information on defining the schema for temp-table parameters mapped to a `ProDataGraph`, see the “Defining the schema for a temp-table parameter mapped to a `ProDataGraph`” section on page 11–31.

Metadata

Specifies an instance of `MetaType`. When `mode` is `ParamArrayMode.OUTPUT` and `ProgressType` is `DatasetHandle` or `TableHandle` (passing a dynamic ProDataSet or temp-table parameter), you can set this parameter to `null`.
Syntax summaries of all data type-specific add parameter methods

The following sections list syntax summaries for all of the data type-specific methods for adding parameters to a ParamArray object:

- CHARACTER
- COM-HANDLE
- DATASET and DATASET-HANDLE
- DATE
- DATETIME
- DATETIME-TZ
- DECIMAL
- HANDLE
- INT64
- INTEGER
- LOGICAL
- LONGCHAR
- MEMPTR
- RAW
- RECID
- ROWID
- TABLE and TABLE-HANDLE

**CHARACTER**

These are the data type-specific methods for adding an ABL CHARACTER parameter:

**Syntax**

```java
public void addCharacter(int position, String value, int mode)
throws Open4GLException

public void addCharacterArray(int position, String[] value, int mode, int extentValue)
throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.
Setting up parameters

COM-HANDLE

These are the data type-specific methods for adding an ABL COM-HANDLE parameter:

Syntax

```java
public void addCOMHandle(int position, com.progress.open4gl.COMHandle value,
                           int mode)
    throws Open4GLEException

public void addCOMHandleArray(int position,
                               com.progress.open4gl.COMHandle[] value,
                               int mode, int extentValue)
    throws Open4GLEException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

DATASET and DATASET-HANDLE

These are the data type-specific methods for adding an ABL ProDataSet (DATASET or DATASET-HANDLE) parameter:

Syntax

```java
public void addDataset
    (int position,
     com.progress.open4gl.ProDataGraph value, int mode,
     com.progress.open4gl.ProDataGraphMetaData metaData)
    throws Open4GLEException

public void addDatasetHandle
    (int position,
     com.progress.open4gl.ProDataGraph value, int mode,
     com.progress.open4gl.ProDataGraphMetaData metaData)
    throws Open4GLEException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

DATE

These are the data type-specific methods for adding an ABL DATE parameter:

Syntax

```java
public void addDate(int position, java.util.GregorianCalendar value,
                    int mode)
    throws Open4GLEException

public void addDateArray(int position, java.util.GregorianCalendar[] value,
                          int mode, int extentValue)
    throws Open4GLEException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.
DATETIME

These are the data type-specific methods for adding an ABL DATETIME parameter:

Syntax

```java
public void addDatetime(int position,
                       java.util.GregorianCalendar value,
                       int mode)
    throws Open4GLException

public void addDatetimeArray(int position,
                             java.util.GregorianCalendar[] value,
                             int mode, int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

DATETIME-TZ

These are the data type-specific methods for adding an ABL DATETIME-TZ parameter:

Syntax

```java
public void addDatetimeTZ(int position,
                          java.util.GregorianCalendar value,
                          int mode)
    throws Open4GLException

public void addDatetimeTZArray(int position,
                               java.util.GregorianCalendar[] value,
                               int mode, int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

DECIMAL

These are the data type-specific methods for adding an ABL DECIMAL parameter:

Syntax

```java
public void addDecimal(int position, java.math.BigDecimal value,
                       int mode)
    throws Open4GLException

public void addDecimalArray(int position, java.math.BigDecimal[] value,
                            int mode, int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.
HANDLE

These are the data type-specific methods for adding an ABL HANDLE parameter:

Syntax

```java
public void addHandle(int position,
        com.progress.open4gl.Handle value,
        int mode)
        throws Open4GLException

public void addHandleArray(int position,
        com.progress.open4gl.Handle[] value,
        int mode, int extentValue)
        throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

INT64

These are the data type-specific methods for adding an ABL INT64 parameter:

Syntax

```java
public void addInt64(int position, Long value, int mode)
    throws Open4GLException

public void addInt64Array(int position, Long[] value, int mode,
        int extentValue)
    throws Open4GLException

public void addInt64(int position, long value, int mode)
    throws Open4GLException

public void addInt64Array(int position, long[] value, int mode,
        int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

INTEGER

These are the data type-specific methods for adding an ABL INTEGER parameter:

Syntax

```java
public void addInteger(int position, Integer value, int mode)
    throws Open4GLException

public void addIntegerArray(int position, Integer[] value, int mode,
        int extentValue)
    throws Open4GLException

public void addInteger(int position, int value, int mode)
    throws Open4GLException

public void addIntegerArray(int position, int[] value, int mode,
        int extentValue)
    throws Open4GLException
```
For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

**LOGICAL**

These are the data type-specific methods for adding an ABL LOGICAL parameter:

**Syntax**

```java
public void addLogical(int position, Boolean value, int mode)
    throws Open4GLException

public void addLogicalArray(int position, Boolean[] value, int mode,
    int extentValue)
    throws Open4GLException

public void addLogical(int position, boolean value, int mode)
    throws Open4GLException

public void addLogicalArray(int position, boolean[] value, int mode,
    int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

**LONGCHAR**

These are the data type-specific methods for adding an ABL LONGCHAR parameter:

**Syntax**

```java
public void addLongchar(int position, String value, int mode)
    throws Open4GLException

public void addLongcharArray(int position, String[] value, int mode,
    int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

**MEMPTR**

These are the data type-specific methods for adding an ABL MEMPTR parameter:

**Syntax**

```java
public void addMemptr(int position, com.progress.open4gl.Memptr value,
    int mode)
    throws Open4GLException

public void addMemptrArray(int position, com.progress.open4gl.Memptr[] value,
    int mode, int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.
These are the data type-specific methods for adding an ABL RAW parameter:

**Syntax**

```java
public void addRaw(int position, byte[] value, int mode)
    throws Open4GLException

public void addRawArray(int position, byte[][] value, int mode,
    int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

These are the data type-specific methods for adding an ABL RECID parameter:

**Syntax**

```java
public void addRecid(int position, Long value, int mode)
    throws Open4GLException

public void addRecidArray(int position, Long[] value, int mode,
    int extentValue)
    throws Open4GLException

public void addRecid(int position, long value, int mode)
    throws Open4GLException

public void addRecidArray(int position, long[] value, int mode,
    int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

These are the data type-specific methods for adding an ABL ROWID parameter:

**Syntax**

```java
public void addRowid(int position, com.progress.open4gl.Rowid value,
    int mode)
    throws Open4GLException

public void addRowidArray(int position, com.progress.open4gl.Rowid[] value,
    int mode, int extentValue)
    throws Open4GLException
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.
**TABLE and TABLE-HANDLE**

These are the data type-specific methods for adding an ABL temp-table (TABLE or TABLE-HANDLE) parameter:

**Syntax**

```java
public void addTable
  (int position, java.sql.ResultSet value, int mode,
   com.progress.open4gl.ProResultSetMetaDataImpl metaData)
  throws Open4GLEexception

public void addTableHandle
  (int position, java.sql.ResultSet value, int mode,
   com.progress.open4gl.ProResultSetMetaDataImpl metaData)
  throws Open4GLEexception

public void addTable
  (int position,
   com.progress.open4gl.ProDataGraph value, int mode,
   com.progress.open4gl.ProDataGraphMetaData metaData)
  throws Open4GLEexception

public void addTableHandle
  (int position,
   com.progress.open4gl.ProDataGraph value, int mode,
   com.progress.open4gl.ProDataGraphMetaData metaData)
  throws Open4GLEexception
```

For information on the parameters to these methods, see the “General syntax for data type-specific add parameter methods” section on page 11–8.

**Adding parameters using a generic method**

You can use the following generic method on the
com.progress.open4gl.javaproxy.ParamArray class to set the parameters in the ParamArray:

**Syntax**

```java
public void addParameter
  (int position, Object value, int mode,
   int proType, int extentValue,
   com.progress.open4gl.ProResultSetMetaDataImpl metaData)
  throws Open4GLEexception

public void addParameter
  (int position, Object value, int mode,
   int proType, int extentValue,
   com.progress.open4gl.ProDataGraphMetaData metaData)
  throws Open4GLEexception
```

**position**

Specifies a 0-based index indicating the parameter position.

**value**

Specifies a variable of the appropriate `Object` data type that contains the value for the parameter, or `null` for an OUTPUT parameter.
Setting up parameters

**mode**

Specifies the ABL mode (passing direction) of the parameter. This can be one of the following constant values:

- `ParamArrayMode.INPUT`
- `ParamArrayMode.INPUT_OUTPUT`
- `ParamArrayMode.OUTPUT`

**proType**

Specifies the ABL data type of the parameter indicated by a class constant defined in the `com.progress.open4gl.Parameter` class, as shown in Table 11–2.

<table>
<thead>
<tr>
<th>Progress data type</th>
<th>com.progress.open4gl.Parameter class constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>PRO_CHARACTER</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>PRO_COMHANDLE</td>
</tr>
<tr>
<td>DATASET</td>
<td>PRO_DATASET</td>
</tr>
<tr>
<td>DATASET-HANDLE</td>
<td>PRO_DATASETHANDLE</td>
</tr>
<tr>
<td>DATE</td>
<td>PRO_DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>PRO_DATETIME</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>PRO_DATETIMETZ</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PRO_DECIMAL</td>
</tr>
<tr>
<td>INT64</td>
<td>PRO_INT64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>PRO_INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>PRO_LOGICAL</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>PRO_LONGCHAR</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>PRO_MEMPTR</td>
</tr>
<tr>
<td>RAW</td>
<td>PRO_RAW</td>
</tr>
<tr>
<td>RECID</td>
<td>PRO_RECID</td>
</tr>
<tr>
<td>ROWID</td>
<td>PRO_ROWID</td>
</tr>
<tr>
<td>TABLE</td>
<td>PRO_TEMPTABLE</td>
</tr>
<tr>
<td>TABLE-HANDLE</td>
<td>PRO_TABLEHANDLE</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
</tbody>
</table>
extentValue

Specifies the extent of an array, zero for a scalar value.

metaData

Specifies the schema for ProDataSet or temp-table parameters, where proType is one of the following:

- Parameter.PRO_DATASET
- Parameter.PRO_DATASETHANDLE
- Parameter.PRO_TEMPTABLE
- Parameter.PRO_TABLEHANDLE

The value can be specified using one of the following classes:

- **ProResultSetMetaDataImpl** — To define the schema information for temp-table parameters that are mapped to a java.sql.ResultSet. For more information on defining this schema, see the “Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet” section on page 11–22.

- **ProDataGraphMetaData** — To define the schema information for ProDataSet parameters and temp-table parameters that are mapped to a com.progress.open4gl.ProDataGraph. For more information on defining the schema for ProDataSet parameters, see the “Defining the schema for a ProDataSet parameter” section on page 11–24. For more information on defining the schema for temp-table parameters mapped to a ProDataGraph, see the “Defining the schema for a temp-table parameter mapped to a ProDataGraph” section on page 11–31.

When mode is ParamArrayMode.OUTPUT and proType is Parameter.PRO_DATASETHANDLE or Parameter.PRO_TABLEHANDLE (passing a dynamic ProDataSet or temp-table parameter), you can set this parameter to null.
Defining the return type for a user-defined function

You must specify the return type for a user-defined function as part of setting up the parameters. You can set this return type using the following method on the com.progress.open4gl.javaproxy.ParamArray object:

Syntax

```java
public void setReturnType(int proType) throws Open4GLError
```

**proType**

Specifies an ABL data type returned by the user-defined function and indicated by a class constant defined in the com.progress.open4gl.Parameter class, as shown in Table 11–3.

**Note:** User-defined functions cannot return LONGCHAR and MEMPTR data across an AppServer.

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>com.progress.open4gl.Parameter class constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>PRO_CHARACTER</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>PRO_COMHANDLE</td>
</tr>
<tr>
<td>DATE</td>
<td>PRO_DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>PRO_DATETIME</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>PRO_DATETIMETZ</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PRO_DECIMAL</td>
</tr>
<tr>
<td>INT64</td>
<td>PRO_INT64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>PRO_INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>PRO_LOGICAL</td>
</tr>
<tr>
<td>RAW</td>
<td>PRO_RAW</td>
</tr>
<tr>
<td>RECID</td>
<td>PRO_RECID</td>
</tr>
<tr>
<td>ROWID</td>
<td>PRO_ROWID</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
</tbody>
</table>

For example, to set the return type for a user-defined function to the ABL INTEGER data type, you can invoke the setReturnType() method on a ParamArray object, parms, as follows:

```java
parms.setReturnType(Parameter.PRO_INTEGER);
```
Arrays as return values

Java Open Clients can return array values (called extents in ABL) from user-defined functions. Table 11–4 provides the data type mappings for array parameters.

Table 11–4: Data type mappings between ABL and Java array types

<table>
<thead>
<tr>
<th>ABL type with EXTENT</th>
<th>Java proxy return type for values</th>
<th>Java proxy return type for unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>int[ ]</td>
<td>Integer[]</td>
</tr>
<tr>
<td>INT64</td>
<td>long[]</td>
<td>Long[]</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal[ ]</td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td>java.lang.String[ ]</td>
<td></td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean[ ]</td>
<td>Boolean[ ]</td>
</tr>
<tr>
<td>DATE</td>
<td>java.util.GregorianCalendar[ ]</td>
<td></td>
</tr>
<tr>
<td>DATETIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECID</td>
<td>long[ ]</td>
<td></td>
</tr>
<tr>
<td>RAW</td>
<td>byte[ ][ ]</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>com.progress.open4gl.Rowid[ ]</td>
<td></td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>com.progress.open4gl.COMHandle[ ]</td>
<td></td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>com.progress.open4gl.Handle[ ]</td>
<td></td>
</tr>
</tbody>
</table>

setIsReturnExtent( ) method

To specify that a return value is an array, call the setIsReturnExtent() method with TRUE on the com.progress.open4gl.javaproxy.ParamArray object. When you get the return value with the getFunctionReturnValue() method, it returns an object which you can then cast to an array.

Syntax

```java
public bool setIsReturnExtent( boolean bIsReturnExtent )
```

The default value is FALSE.

Handling unknown values in arrays.

The ABL Unknown value (?) is mapped to Java null values. A Java array is an object so, it too can be assigned the null value. In this case all array elements are null. Since Java int[], boolean[], and long[] are arrays of intrinsic types, the individual elements of the array cannot be null. The int[], boolean[], or long[] array as a whole can be null and that means that every element of an array return value on the Appserver is the Unknown value (?). If you want to return the Unknown value (?) to individual array elements, use the Allow Unknown... check box in the ProxyGen tool and then the generated proxy will contain Integer[], Boolean[], or Long[] for the return value instead of int[], boolean[], or long[].
setIsReturnUnknown( ) method

This method must be set to TRUE if the return type from a user defined function is an array that can contain the Unknown value (?). Setting the method for scalar return values is not necessary, but it may be a useful convention for code readability.

Syntax

```java
public bool setIsReturnUnknown( boolean bIsReturnUnknown )
```

For scalar return values, when you call `getReturnValue()`, the method returns an object. If that object is null, then you know the ABL user-defined function returned the Unknown value (?). If the object is not null, you can then cast it to the proper type object to get the value:

```java
Integer intRetVal = (Integer)parms.getReturnValue();
```

For array return values, if the method is set to FALSE, the default, the Open Client expects a value and maps the return to the data type shown in the second column of Table 11–4. If the method is TRUE, then the return maps to the alternate data type if one is shown in column three of the table.

For example, if the ABL return type is `INTEGER EXTENT` and this method is set to TRUE, the `getReturnValue()` method returns `Integer[]`. If it is set to FALSE, the method returns `int[]`.

```java
ParamArray parms = new ParamArray(1);
...
// Set up return type
parms.setReturnType (Parameter.PRO_DECIMAL);
parms.setIsReturnExtent (true);
// Run procedure
openPO.RunProc("GetOrderTotalsByDollar", parms);
// Get return value
java.math.BigDecimal [] retValArray = (java.math.BigDecimal [])
 parms.getReturnValue();
```
Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet

Defining the schema for an ABL TABLE or TABLE-HANDLE parameter passed as a java.sql.ResultSet is a multi-step process.

**Note:** If you pass a temp-table as a java.sql.ResultSet parameter you must access the parameter as a data stream. For more information on accessing temp-tables as java.sql.ResultSet parameters, see Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”

To define the schema for a temp-table parameter passed as a java.sql.ResultSet:

2. Add field descriptions to the ProResultSetMetaDataImpl object.
3. If the parameter is for input or input-output, define a java.sql.ResultSet to hold the parameter value.
4. Add the ProResultSetMetaDataImpl object as a temp-table parameter to your ParamArray object using the appropriate set parameter method.

**Note:** You can also pass a temp-table parameter as a ProDataGraph. For more information, see the “Defining the schema for a temp-table parameter mapped to a ProDataGraph” section on page 11–31.

Defining a ProResultSetMetaDataImpl object

For each temp-table parameter you must define a com.progress.open4gl.ProResultSetMetaDataImpl object to hold the schema using the following constructor:

**Syntax**

```
public ProResultSetMetaDataImpl(int numFields)
```

**numFields**

Specifies the number of fields (or columns) in the temp-table.
Adding field descriptions to the ProResultSetMetaDataImpl object

Add the meta data for each field of the temp-table by calling the setFieldMetaData() method on the ProResultSetMetaDataImpl object for each column in the temp-table:

**Syntax**

```java
public void setFieldMetaData(int position, String name, int extentValue, int proType)
```

*position*

Specifies the 1-based position of a mapped field in an ABL temp-table.

*name*

Specifies a field name, typically set to the corresponding ABL field name in the temp-table.

*extentValue*

Specifies the extent of an array, 0 or 1 for a scalar value.

*proType*

Specifies the value of a class constant defined in the com.progress.open4gl.Parameter class. The specified class constant indicates the ABL data type of the mapped temp-table field. For more information on these class constants, see the sections on specifying field data type meta data for temp-tables in Chapter 4, “Passing Parameters.” To identify the Java data type that the column property assumes for the specified ABL data type, see the information on mapping temp-tables to java.sql.ResultSet objects in Appendix C, “Passing Temp-tables as SQL ResultSet Parameters.”
Adding the ProResultSetMetaDataImpl object as a temp-table parameter

Add the ProResultSetMetaDataImpl object by passing it as a parameter of the addTable() or addTableHandle() method that you use to add the temp-table parameter to the ParamArray object. For more information, see the “TABLE and TABLE-HANDLE” section on page 11–16.

The following example adds a temp-table as an input parameter.

Example 11–1: OpenAPI fragment adding a temp-table parameter as an SQL ResultSet

```java
// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Set up the meta data
ProResultSetMetaDataImpl metaData1 = new ProResultSetMetaDataImpl(6);
metaData1.setFieldMetaData(1, "OrderNum", 0, Parameter.PRO_INTEGER);
metaData1.setFieldMetaData(2, "SalesRep", 0, Parameter.PRO_CHARACTER);
metaData1.setFieldMetaData(3, "OrderDate", 0, Parameter.PRO_DATE);
metaData1.setFieldMetaData(4, "ShipDate", 0, Parameter.PRO_DATE);
metaData1.setFieldMetaData(5, "TotalDollars", 0, Parameter.PRO_DECIMAL);
metaData1.setFieldMetaData(6, "OrderStatus", 0, Parameter.PRO_CHARACTER);

// Create result set for an input parameter
ResultSet rs = new ResultSet();
...
// Add the parameter to the ParamArray
parms.addTable(0, rs, ParamArrayMode.INPUT, metaData1);
```

Defining the schema for a ProDataSet parameter

Defining the schema for an ABL ProDataSet (DATASET or DATASET-HANDLE) parameter is a multi-step process.

**Note:** This is a condensed description with reference to the Java OpenAPI. For complete information on defining the schema for a ProDataSet parameter, see Chapter 5, “Accessing ABL ProDataSets.”

To define the schema for a ProDataSet parameter:

1. Define a com.progress.open4gl.ProDataGraphMetaData object to specify the schema.
2. Define the temp-tables for the ProDataGraphMetaData object.
3. Define the data-relations for the ProDataGraphMetaData object.
4. If the parameter is for input or input-output, define a ProDataGraph using the ProDataGraphMetaData object to hold the parameter value.
5. Specify the ProDataGraphMetaData object and any ProDataGraph (for input or
input-output) as a ProDataSet parameter to your ParamArray object using the appropriate
set parameter method.

**Defining a ProDataGraphMetaData object**

For each ProDataSet parameter you must define a
com.progress.open4gl.ProDataGraphMetaData object. You use this object to specify the
schema for the ProDataSet when you set the DATASET or DATASET-HANDLE parameter in the
ParamArray object. You can create an instance of this object using the following constructor:

**Syntax**

```java
public ProDataGraphMetaData(String proDataSetName)
```

*proDataSetName*

Specifies the ProDataSet name in ABL.

**Defining the temp-tables for the ProDataGraphMetaData object**

You must define the meta data for each temp-table defined by the ProDataGraphMetaData. For more information, see the “Defining the schema for temp-tables in a ProDataSet” section on page 11–28. You can then add the temp-table meta data (ProDataObjectMetaData) to the
ProDataGraphMetaData object using the following ProDataGraphMetaData method:

**Syntax**

```java
public void addTable(ProDataObjectMetaData doMetaData)
```

*doMetaData*

Specifies the meta data for a temp-table.

**Defining the data-relations for the ProDataGraphMetaData object**

You must define any data-relations that are defined for the ABL ProDataSet using
com.progress.open4gl.ProDataRelationMetaData objects. You can create a
ProDataRelationMetaData object for each data-relation between a parent and child temp-table
using the following constructor:

**Syntax**

```java
public ProDataRelationMetaData(String dataRelationName, int parentIx, int childIx, int numPairs, String pairsList)
```

*dataRelationName*

Specifies the name of the ProDataRelationMetaData object.

*parentIx*

Specifies a 0-based index to a parent temp-table (ProDataObject collection) that corresponds to an index into the array of table names returned by the
ProDataGraphMetaData getTableNames() method.
Using the Open Client Java OpenAPI to Directly Access the AppServer

childIx

Specifies a 0-based index to a child temp-table (ProDataObject collection) that corresponds to an index into the array of table names returned by the ProDataGraphMetaData.getTableNames() method.

numPairs

Specifies the number of column property pairs (key field pairs) that represent this relationship. This allows multiple fields to represent a key relationship between the parent and child temp-table.

pairsList

Specifies a String containing a comma-separated list of field names. The list consists of numPairs field pairs, where the parent temp-table field name for each pair is followed by its matching child temp-table field name. The data types of the named parent and child temp-table field pairs must be comparable.

**Note:** For the equivalent ProDataRelationMetaData() constructor in the .NET OpenAPI, the order of parent and child field names in the pairsList parameter is reversed. For more information, see *OpenEdge Development: .NET Open Clients.*

You can add each data-relation definition to the ProDataGraphMetaData object using the following method:

**Syntax**

```
public void addDataRelation(ProDataRelationMetaData drMetaData)
```

**drMetaData**

Specifies a data-relation to include in the ProDataGraphMetaData object.

**Define a ProDataGraph for any input parameter**

If the parameter is for input or input-output, define the com.progress.open4gl.ProDataGraph to hold the value using the following constructor:

**Syntax**

```
ProDataGraph(ProDataGraphMetaData dgmd)
```

**dgmd**

Specifies the ProDataGraphMetaData that you have defined for the parameter.
Specifying the ProDataGraphMetaData object for a ProDataSet parameter

Specify the ProDataGraphMetaData object together with any input ProDataGraph by passing them as parameters of the addDataset() or addDatasetHandle() method that you use to add the ProDataSet parameter to the ParamArray object. For more information, see the “DATASET and DATASET-HANDLE” section on page 11–11.

The following example adds a temp-table parameter defined with no indexes.

Example 11–2: OpenAPI fragment setting a ProDataSet parameter

```java
// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Create the ProDataGraphMetaData
ProDataGraphMetaData dgMetaData = new ProDataGraphMetaData("dsCustOrd");

// Create the ProDataObjectMetaData for the Customer table
ProDataObjectMetaData doCustMD = new ProDataObjectMetaData("ttCust", 3, false, 0, null, null, null);

doCustMD.setFieldMetaData
    (1, "CustNum", 0, Parameter.PRO_INTEGER, 0, 0);

doCustMD.setFieldMetaData
    (2, "Name", 0, Parameter.PRO_CHARACTER, 1, 0);

doCustMD.setFieldMetaData
    (3, "SalesRep", 0, Parameter.PRO_CHARACTER, 2, 0);

// Create the ProDataObjectMetaData for the Order table
ProDataObjectMetaData doOrderMD = new ProDataObjectMetaData("OrderDetails", 6, false, 0, null, null, null);

doOrderMD.setFieldMetaData
    (1, "OrderNum", 0, Parameter.PRO_INTEGER, 0, 0);

doOrderMD.setFieldMetaData
    (2, "SalesRep", 0, Parameter.PRO_CHARACTER, 1, 0);

doOrderMD.setFieldMetaData
    (3, "OrderDate", 0, Parameter.PRO_DATE, 2, 0);

doOrderMD.setFieldMetaData
    (4, "ShipDate", 0, Parameter.PRO_DATE, 3, 0);

doOrderMD.setFieldMetaData
    (5, "TotalDollars", 0, Parameter.PRO_DECIMAL, 4, 0);

doOrderMD.setFieldMetaData
    (6, "OrderStatus", 0, Parameter.PRO_CHARACTER, 5, 0);

// Add the tables to the ProDataGraph meta data
dgMetaData.addTable(doCustMD);
dgMetaData.addTable(doOrderMD);

// Create and add the relations to the DataSet meta data
ProDataRelationMetaData relation = new ProDataRelationMetaData("custNum", 0, 1, 1, "CustNum,CustNum");
dgMetaData.addDataRelation(relation);

parms.addDataset(0, null, ParamArrayMode.OUTPUT, dgMetaData);
```

For more information on the objects and methods for defining the schema of a ProDataSet parameter and on working with the ProDataGraph used to map a ProDataSet parameter, see Chapter 5, “Accessing ABL ProDataSets.”
Defining the schema for temp-tables in a ProDataSet

Defining the schema for a temp-table (ProDataObject collection) in a ProDataSet is a multi-step process.

Note: This is a condensed description with reference to the Java OpenAPI. For complete information on defining the schema temp-tables in a ProDataSet parameter, see Chapter 5, “Accessing ABL ProDataSets.”

To define the schema for a temp-table in a ProDataSet:

1. Define a `com.progress.open4gl.ProDataObjectMetaData` object to hold the schema.
2. Add field descriptions to the `ProDataObjectMetaData` object.
3. Add the `ProDataObjectMetaData` object to the `ProDataGraphMetaData` for the ProDataSet.

Defining a `ProDataObjectMetaData` object

For each table (ProDataObject collection) contained within a ProDataGraph, you must define a `com.progress.open4gl.ProDataObjectMetaData` object to hold the schema using the following constructor:

Syntax

```java
public ProDataObjectMetaData(String tableName, int numFields,
        boolean bimageFlag,
        int numIndexes, String multiIxCols,
        String XMLNamespace, String XMLPrefix)
```

`tableName`

Specifies a name for the specified ProDataObject type (and collection). This name is typically identical to any ABL temp-table to which this collection is mapped.

`numFields`

Specifies the number of fields (column properties) in the specified ProDataObject type.

`bimageFlag`

Specifies true if the corresponding ABL temp-table is defined with the BEFORE-TABLE option, indicating that the temp-table (and hence the ProDataObject collection) can be modified. Otherwise, this value must be false and you cannot pass modified data between the Open Client and the AppServer.

`numIndexes`

Specifies the number of indexes on the table.
multiIxCols

Specifies null if there are no indexes or a formatted string that contains all the index info for this temp-table, as follows:

**Syntax**

```
"[primeUniqueFlag,primeFld1[,primeFldn]]...:primeIdxName.]...[uniqueIdxFld1[,uniqueIdxFldn]]...:uniqueIdxName.]..."
```

For more information on this formatted string, see the section on the ProDataObjectMetaData() constructor in Chapter 5, “Accessing ABL ProDataSets.”

XMLNamespace

Specifies the namespace for XML serialization or null.

XMLPrefix

Specifies the prefix for XML serialization or null.

Adding field descriptions to the ProDataObjectMetaData object

For each field (ProDataObject column property) in the temp-table, add its meta data by calling the setFieldMetaData() method on the ProDataObjectMetaData object:

**Syntax**

```
public void setFieldMetaData(int fieldNumber, String fieldName,
    int extentValue, int proType,
    int userOrder, int xmlMapping)
```

**fieldNumber**

Specifies a 1-based position that corresponds to the position of a mapped field in an ABL temp-table.

**fieldName**

Specifies a name that is typically identical to a mapped field in the corresponding ABL temp-table. The value cannot be null and must be unique among fields (column properties) in the specified ProDataObject type.

**extentValue**

Specifies if and how the field represents an array field in the corresponding temp-table. The value must be 0 or greater. If the value is greater than 1, this column property is many-valued (represents an array field) and the value is its extent. If the property represents a BLOB or CLOB field, the value must be 0 or 1.
**proType**

Specifies the value of a class constant defined in the `com.progress.open4gl.Parameter` class. The specified class constant indicates the ABL data type of the mapped temp-table field. For more information on these class constants, see the sections on specifying field data type meta data for temp-tables in Chapter 4, “Passing Parameters.” To identify the Java data type that the column property assumes for the specified ABL data type, see the information on managing `ProDataObject` mappings to temp-tables in Chapter 5, “Accessing ABL ProDataSets.”

**userOrder**

Specifies a 0-based user order position for the column property.

**xmlMapping**

Reserved for future use. Always specify 0.

### Adding the ProDataObjectMetaData object to the ProDataGraphMetaData

Add the `ProDataObjectMetaData` object to the `ProDataGraphMetaData` using the following `ProDataGraphMetaData` object method:

**Syntax**

```java
public void addTable(ProDataObjectMetaData doMetaData)
```

**doMetaData**

Specifies the `ProDataObjectMetaData` for one temp-table in the `ProDataSet` parameter.

The following example adds a temp-table parameter defined with no indexes.

**Example 11–3: OpenAPI fragment adding a temp-table parameter**

```java
// Create the ProDataObjectMetaData for the Customer table
ProDataObjectMetaData doCustMD = new ProDataObjectMetaData
   ("ttCust", 3, false, 0, null, null, null);

dCustMD.setFieldMetaData(1, "CustNum", 0, Parameter.PRO_INTEGER, 0, 0);
dCustMD.setFieldMetaData(2, "Name", 0, Parameter.PRO_CHARACTER, 1, 0);
dCustMD.setFieldMetaData(3, "SalesRep", 0, Parameter.PRO_CHARACTER, 2, 0);

// Add the table to the ProDataGraph meta data
dgMetaData.addTable(doCustMD);
```

For more information on the objects and methods for defining the schema (meta data) of temp-tables in a `ProDataGraph`, see Chapter 5, “Accessing ABL ProDataSets.”
Defining the schema for a temp-table parameter mapped to a ProDataGraph

Defining the schema for an ABL temp-table (TABLE or TABLE-HANDLE) parameter passed as a com.progress.open4gl.ProDataGraph is a multi-step process.

Note: This is a condensed description with reference to the Java OpenAPI. For complete information on defining the schema for a temp-table parameter mapped to a ProDataGraph, see Chapter 5, “Accessing ABL ProDataSets.”

To define the schema for a temp-table parameter passed as a ProDataGraph:

1. Define a com.progress.open4gl.ProDataGraphMetaData object.
2. Add a single com.progress.open4gl.ProDataObjectMetaData object to the ProDataGraphMetaData that describes the schema of the temp-table parameter you are passing.
3. If the parameter is for input or input-output, define a com.progress.open4gl.ProDataGraph using the ProDataGraphMetaData to hold the parameter value.
4. Specify the ProDataGraphMetaData object and any ProDataGraph (for input or input-output) as a ProDataSet parameter to your ParamArray object using the appropriate set parameter method. For example:

Syntax

```java
public void addTableHandle
    (int position,
     com.progress.open4gl.ProDataGraph value, int mode,
     com.progress.open4gl.ProDataGraphMetaData metaData)
    throws Open4GLException
```

Note: You can also pass a temp-table parameter as a java.sql.ResultSet. For more information, see the “Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet” section on page 11–22.

This process is almost identical to defining the schema to set up a ProDataSet parameter. The two differences are that you:

1. Define only one temp-table in the ProDataGraphMetaData to represent the temp-table parameter you are passing.
2. Must add a temp-table parameter to a parameter array using the ProDataGraphMetaData, pass the ProDataGraphMetaData (and any input or input-output ProDataGraph) to the appropriate setTable1e() or setTableHandle() method (instead of a setDataset() or setDatasetHandle() method, as for a ProDataSet parameter).
Thus, the ProDataGraph becomes a wrapper for the actual temp-table parameter value. With these differences in mind, for more information on the steps to pass a temp-table parameter as a ProDataGraph, see the “Defining the schema for a ProDataSet parameter” section on page 11–24.

For more information on working with temp-table parameters mapped to a ProDataGraph, see Chapter 5, “Accessing ABL ProDataSets.”
Passing parameters

After you have set up the required parameter array, you can pass the specified parameters to a procedure or user-defined function. The steps for setting up and passing parameters differs, depending on the parameter mode (INPUT, OUTPUT, or INPUT-OUTPUT).

Passing INPUT parameters

Passing an INPUT parameter requires several steps to provide a value as input to an application service procedure or user-defined function.

To pass an INPUT parameter:

1. Create and initialize a variable for the parameter of the correct Java data type (see the “Creating variables for parameters” section on page 11–7).

2. Add the parameter to a ParamArray object (see the “Setting up a parameter array” section on page 11–7).

3. Run the procedure or user-defined function (see the “Running procedures and user-defined functions” section on page 11–4).

This example passes an INPUT integer parameter that does not support the Unknown value (?).

Example 11–4: Passing an INPUT parameter using the Java OpenAPI

```java
// Connect to the AppServer
OpenAppObject dynAO = new OpenAppObject("asbroker1");

// Define and initialize the variable for the input parameter
int CustomerNumber = 33;

// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Add the input parameter to the ParamArray
parms.addInteger(0, CustomerNumber, ParamArrayMode.INPUT);

// Run the procedure or user-defined function
dynAO.runProc("AddCustomer.p", parms);
...
```
This examples passes an INPUT integer parameter that supports the Unknown value (?).

**Example 11–5: Passing an INPUT parameter as the Unknown value (?) using the Java OpenAPI**

```java
// Connect to the AppServer
OpenAppObject dynAO = new OpenAppObject("asbroker1");

// Define and initialize the variable for the input parameter
Integer CustomerNumber = new Integer(33);

// Code possibly affecting the value of CustomerNumber
...

// Set the Integer value for the input parameter
// to null or an int based on the variable value
if (CustomerNumber.intValue() > 33)
    CustomerNumber = null;

// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Add the input parameter to the ParamArray
parms.addInteger(0, CustomerNumber, ParamArrayMode.INPUT);

// Run the procedure or user-defined function
dynAO.runProc("AddCustomer.p", parms);
...
```

**Passing INPUT-OUTPUT parameters**

Passing an INPUT-OUTPUT parameter requires several steps to provide a value as input to an application service procedure or user-defined function, then to return a value as output using the same parameter.

To pass an INPUT-OUTPUT parameter:

1. Create and initialize a variable for the parameter of the correct Java data type (see the “Creating variables for parameters” section on page 11–7).
2. Add the parameter to a ParamArray object (see the “Setting up a parameter array” section on page 11–7).
3. Run the procedure or user-defined function (see the “Running procedures and user-defined functions” section on page 11–4).
4. Get the output value from the ParamArray (see the “Getting OUTPUT parameter values” section on page 11–37). You might also want to reset the variable for reuse.
This example passes an INPUT-OUTPUT integer parameter that does not support the Unknown value (?).

**Example 11–6: Passing an INPUT-OUTPUT parameter using the Java OpenAPI**

```java
// Define the variable for the input-output parameter
// Using Integer object because the output value
// is always returned as an Object
Integer CustomerNumber = new Integer(3);

// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Add the input-output parameter to the ParamArray
parms.addInteger(0, CustomerNumber, ParamArrayMode.INPUT_OUTPUT);

// Run the procedure
...

// Fill from output parameter - must cast from Object
CustomerNumber = (Integer) parms.getOutputParameter(0);
```

**Passing OUTPUT parameters**

Passing an OUTPUT parameter requires several steps to provide a parameter as output from an application service procedure or user-defined function, then to get the value returned by the parameter.

To pass an OUTPUT parameter:

1. Create a variable of the correct Java data type to hold the returned value for the parameter (see the “Creating variables for parameters” section on page 11–7).
2. Add the parameter to a ParamArray object (see the “Setting up a parameter array” section on page 11–7).
3. Run the procedure or user-defined function (see the “Running procedures and user-defined functions” section on page 11–4).
4. Get the value from the ParamArray (see the “Getting OUTPUT parameter values” section on page 11–37).
This example passes an OUTPUT integer parameter that does not support the Unknown value (?).

**Example 11–7: Passing an OUTPUT parameter using the Java OpenAPI**

```java
// Define the variable for the input-output parameter
// Using Integer object because the output value
// is always returned as an Object
Integer CustomerNumber = new Integer(3);

// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Set up output parameter - notice no variable is needed at this point
parms.addInteger(0, null, ParamArrayMode.OUTPUT);

// Run the procedure...

// Fill output parameter
CustomerNumber = (Integer) parms.getOutputParameter(0);
```
Handling returned values

The Java OpenAPI provides methods to access returned values for:

- Output parameters
- The RETURN-VALUE function set by running an ABL non-persistent, persistent, or internal procedure
- The value returned by an ABL user-defined function

The following sections describe how to return each type of value.

Getting OUTPUT parameter values

After running a procedure or user-defined function you can access the OUTPUT parameters using the following ParamArray method:

**Syntax**

```java
public Object getOutputParameter(int paramNum)
```

*paramNum*

Specifies the 0-based position of the parameter.

The output value is always returned as an `Object`. You need to cast the `Object` and assign it to the output variable you have created. If the value returned can be the Unknown value (?) (null in Java), for intrinsic types, you must use the `Object` type that corresponds to the intrinsic type to handle the return of the actual value.

This example gets an output integer parameter that might be set to the Unknown value (?) by ABL.

**Example 11–8: Getting an OUTPUT parameter using the Java OpenAPI**

```java
// Create the ParamArray and run procedure
...

// Fill output parameter
Integer iCustomerNumber;
iCustomerNumber = (Integer) parms.getOutputParameter(0);
if (iCustomerNumber == null) ...;
else ...;
```
Accessing RETURN-VALUE function output from procedures

You can access the value of the RETURN-VALUE function set by running an ABL procedure using methods available from any of three Java OpenAPI objects:

- `com.progress.open4gl.javaproxy.OpenAppObject` — For returning the value set by the last procedure executed on the associated AppServer, use this method:

  Syntax
  ```java
  public String _getProcReturnString() throws Open4GLEException
  ```

- `com.progress.open4gl.javaproxy.OpenProcObject` — For returning the value set by the last internal procedure executed in the associated persistent procedure, use the following method:

  Syntax
  ```java
  public String _getProcReturnString() throws Open4GLEException
  ```

- `com.progress.open4gl.javaproxy.ParamArray` — Returning the value set by the non-persistent, persistent, or internal procedure executed with parameters passed by the specified ParamArray object, use this method:

  Syntax
  ```java
  public String getProcReturnString()
  ```

For more information on using these objects to execute procedures, see the “Running procedures and user-defined functions” section on page 11–4.

Accessing user-defined function return values

You can access the return value after running any user-defined function using the following method on the `com.progress.open4gl.javaproxy.ParamArray` object:

Syntax
```java
public Object getReturnValue()
```

You need to cast the return `Object` to the correct return type for the function. For example:

```java
String retVal = (String)(parms.getReturnValue());
```

For information on accessing arrays “Arrays as return values” section on page 11–20.
Sample Java OpenAPI code

The following sections describe two types of sample code:

- Non-persistent procedure example
- Persistent procedure example

Note: These samples are not available on the OpenEdge product DVD or the PSDN Web site.

Non-persistent procedure example

Suppose you want to run a non-persistent procedure, AddCustomer.p, with the following definition.

Example 11–9: Non-persistent procedure ABL for the OpenAPI

```ABL
DEFINE INPUT PARAMETER CustName AS CHARACTER.
DEFINE INPUT PARAMETER phone AS CHARACTER.
DEFINE INPUT PARAMETER email AS CHARACTER.
DEFINE OUTPUT PARAMETER CustomerNumber AS INTEGER.
```

You might write the following client code to run it.

Example 11–10: OpenAPI code to run the non-persistent procedure

```java
import com.progress.open4gl.javaproxy.*;
import com.progress.open4gl.Parameter;
public class sampleClient {
    private static final java.lang.String tName = new String( "KM_client" );
    
    public static void nonPersistentProcedure() {
        try // To catch all exceptions
        {
            // Connect to the AppServer
            Connection myConn = new Connection("", "", "");
            OpenAppObject dynAO = new OpenAppObject(myConn, "asbroker2");

            // Create the parameters
            String CustName = "abc";
            String phone = "999-555-1234";
            String email = "me@foo.com";

            Integer CustomerNumber;

            // Create a place for RETURN-VALUE
            String retVal;

            // Create the ParamArray
            ParamArray parms = new ParamArray(4);
```
Persistent procedure example

Suppose you want to run a persistent procedure, CustomerOrder.p, with the following definition, containing a user-defined function.

Example 11–11: Persistent procedure ABL for the OpenAPI

```
DEFINE INPUT PARAMETER custNum AS INTEGER NO-UNDO.
...
FUNCTION GetTotalOrdersByNumber RETURNS INTEGER (threshold AS DECIMAL):
...
END.
...
```

You might write the following client code to run it.

Example 11–12: OpenAPI code to run the persistent procedure (1 of 2)

```
import com.progress.open4gl.javaproxy.*;
import com.progress.open4gl.Parameter;

public class sampleClient
{
    private static final java.lang.String tName = new String( "KM_client" );
```
public static void persistentProcedure()
{
    try // To catch all exceptions
    {
        // Connect to the AppServer
        Connection myConn = new Connection("", ",");
        OpenAppObject dynAO = new OpenAppObject(myConn, "asbroker2");

        // Run the persistent Procedure
        // First set up parameters
        ParamArray parms = new ParamArray(1);
        int custNum = 3;

        // Set up input parameters
        parms.addInteger(0, custNum, ParamArrayMode.INPUT);

        // Run procedure
        OpenProcObject dynPO = dynAO.createPO("OrderInfo/CustomerOrder.p", parms);

        // Call UDF
        // First set up parameters
        Integer retVal;
        java.math.BigDecimal threshold = new java.math.BigDecimal(1000);
        parms.clear(); // Clear for reuse

        // Set up input parameters
        parms.addDecimal(0, threshold, ParamArrayMode.INPUT);

        // Set up return type
        parms.setReturnType(Parameter.PRO_INTEGER);

        // Run procedure
        dynPO.runProc("GetTotalOrdersByNumber", parms);

        // Get return value
        retVal = (Integer)(parms.getReturnValue());

        dynPO._release();
        dynAO._release();
    } // try to catch all unexpected exceptions
    catch ( Exception e )
    {
        System.out.println("Exception in sample2()");
        System.out.println("Exception Message: " + e.getMessage());
        e.printStackTrace();
    }
}
} // class
Accessing a SmartDataObject API Directly

You can directly access the complete ABL API (including custom extensions) for any remote ABL SmartDataObject from an Open Client application, where the SmartDataObject is running in distributed mode on an AppServer. Use this direct access technique only if you need to access custom extensions to the default SmartDataObject API provided by an ABL programmer.

To access only the default API generated by the AppBuilder, and only from a Java Open Client application, use the Java SDOResultSet interface provided by the Open Client Toolkit instead.

The Open Client Toolkit installation includes an SDOAppObject. This is a prebuilt proxy that allows you to access any remote ABL SmartDataObject from a Java application without needing to run ProxyGen. An SDOAppObject essentially is an AppObject that provides the built-in interface to a remote SmartDataObject, but not to any other remote ABL procedure.

To access other remote ABL procedures in addition to SmartDataObjects, you can include the additional ABL procedures you want to access when you build the AppObject or SubAppObject using ProxyGen. You can then use the same built-in interfaces to SmartDataObjects provided for SDOAppObject proxies, to access SmartDataObjects using the AppObject or SubAppObject you built.

If you are generating a Java proxy that accesses an ABL SmartDataObject as an SDOResultSet, you do not have to make the compiled r-code for the SmartDataObject available to ProxyGen. The Java SDOResultSet already understands how to access standard SmartDataObject functionality. For more information, see OpenEdge Development: Java Open Clients. If you are defining a ProcObject in ProxyGen to directly access the SmartDataObject as an ABL persistent procedure; however, you must make the compiled r-code available to ProxyGen, like any other ABL procedure.

This appendix contains the following section:

- Directly accessing the SmartDataObject API in a proxy
Directly accessing the SmartDataObject API in a proxy

Before writing an Open Client application that uses the SmartDataObject API, review all the standard API entry points, to understand the available methods. For more complete information on the available entry points, see the documentation on SmartDataObject methods in *OpenEdge Development: ADM and SmartObjects*, *OpenEdge Development: ADM Reference*, and the AppBuilder online help.

**Defining a ProcObject that maps a SmartDataObject**

To directly access the SmartDataObject API, you need to use API methods that handle all control and communication with the remote SmartDataObject, as required by your application. You can access the SmartDataObject API by defining a ProcObject in ProxyGen that supports this interface. Define the ProcObject to include all the methods (internal procedures and user-defined functions) you want to access in the specified SmartDataObject instance.

To define the methods for this ProcObject and generate the proxy, you first must make the r-code for the corresponding SmartDataObject available to ProxyGen. You do this by setting the ProxyGen Propath Components list appropriately. For more information on locating the r-code for a SmartDataObject, see the information on SmartDataObject support in *OpenEdge Application Server: Developing AppServer Applications*.

To ensure all parameters, internal procedures, and user-defined procedures of the ProcObject allow Progress Unknown value (?), do the following in ProxyGen:

1. In the **AppObject** tab, check **Allow Unknown** for parameters and function return values.
2. In the **Customize Persistent Procedure** dialog box, check **Use AppObject Unknown Setting**.

For more information on defining ProcObjects in ProxyGen, see *OpenEdge Development: Open Client Introduction and Programming*.

---

A–2
If your Java Open Client application uses HTTPS (a direct Secure Sockets Layer (SSL) connection to the AppServer), it also uses (root) digital certificates to validate the identity of the Web Server (AIA) to which you will connect. This appendix describes how to perform basic maintenance operations on these digital certificates, as detailed in the following sections:

- Overview
- Managing certificate store files
- Converting digital certificates
Overview

OpenEdge ships a small set of root digital certificates from leading industry Certificate Authorities (CAs), with the Open Client Toolkit. These digital certificates may be used during development and for redistribution with your Open Client application.

Most likely, there will be times when you need to manage digital certificates. It may be necessary to add, remove, or update the (root) digital certificates contained in the JAR files. For example:

- You might want to use a digital certificate from a Certificate Authority not included with the set in OpenEdge.
- A digital certificate expires, and you might need to replace it with a newer version.
- You might want to create your own set of certificates, using only those certificates you know your application will use.
- You may want to customize your own JAR file with only your own selection of digital certificates.
- The Web Server hosting the AIA may be using a digital certificate issued by a private Certificate Authority whose (root) digital certificate is not distributed with OpenEdge.

The certificate files included with the Open Client Toolkit are stored in .zip and .jar files called certificate store files, in the DLC/certs directory. For more information, see the sections on certificate store management in OpenEdge Development: Open Client Introduction and Programming.

Each certificate store file holds several individual root digital certificates and contains one digital certificate list (.dcl file) that lists all certificate files stored in the file. The individual certificates have one of the following formats:

- **DER** — These file types have extensions of .cer and .crt. There is no difference between these types. One file stores one binary certificate.
- **PEM** — These file types have extensions of .pem, .txt, and .0. There is no difference between these types. One file stores one or more certificates.

To help you view and manage digital certificates, OpenEdge provides a certificate management tool (procertm). The procertm utility runs in Windows and on UNIX platforms and lets you import, export, and remove certificates to and from .jar and .zip files. You also can use this tool to convert digital certificates file formats.
Managing certificate store files

The `procertm` utility is run from a command line and has the following syntax:

**Syntax**

```
procertm [options] cert_store
```

**options**

Any combination of the following, in any order:

- `-v` — Prints verbose information about the progress of the digital certificates import and export. When used with `-l`, additional digital certificate field information is printed.

- `-l` — Lists the contents of the `cert_store` file after all import, export, and remove operations are completed.

- `-p` — Prints the digital certificate list the `cert_store` contents to the file `cert_store.dcl`, after all import, export and remove operations are completed.

- `-i cert` — Imports certificate file(s) matching `cert` to `cert_store` from the working directory. The `cert_store` file is created as required. You can specify this option multiple times. See the definition of `cert` below.

- `-e cert` — Exports the certificate file(s) matching `cert` from `cert_store` to the working directory. Any subdirectories are created if required. You can specify this option multiple times. See the definition of `cert` below.

- `-r cert` — Removes the certificate file(s) matching `cert` from `cert_store`. You can specify this option multiple times. See the definition of `cert` below.

- `-d` — Sets the working directory path where certificates are imported from or exported to. The default is the current working directory.

**cert**

Path to the digital certificate you want to import, export, or remove. This is used with the `-i`, `-e`, and `-r` options.

When importing, the path is a relative path to the **working directory**. When exporting or removing digital certificates from `cert_store`, the path is the full digital certificate path specified in `cert_store`. Subdirectories should be specified with a forward slash (`/`).

You can use multicharacter (`*`) and single-character (`?`) wildcards in the `cert` filename and file extension.

**cert_store**

Path to the `.zip` or `.jar` certificate store file. If the certificate store file does not exist, and you are importing digital certificates, a new file is created.
When you run `procertm`, it performs the options in the following order:

1. Imports any certificates specified with the `-i` option from the working directory into `cert_store`. If a certificate is not found, a warning message displays.

2. Exports any certificates specified with the `-e` option from `cert_store` to the working directory. If a certificate is not found, a warning message displays.

3. Removes any certificates specified with the `-r` option from `cert_store`. If a certificate is not found, a warning message displays.

4. Shows the resulting `cert_store` file contents, if the `-l` option is specified.

5. Prints any digital certificate list information, if the `-p` options is specified.
Converting digital certificates

You can use `procertm` to convert digital certificates between .DER and .PEM file formats. To convert files from one file format to the other, use the following command line syntax:

**Syntax**

```
procertm -c in_cert out_cert
```

`in_cert`

The digital certificate whose file format you want to convert.

`out_cert`

The file format to which you want to convert the digital certificate.

The file utility performs the conversion based on the file-extension type. For example, if `in_cert` has a file extension type of `.crt` and `out_cert` has a file extension type of `.pem`, `in_cert` is converted from the .der to .pem format and written to the file `out_cert`. 
Passing Temp-tables as SQL ResultSet Parameters

The Java Open Client can map a static (TABLE) or dynamic (TABLE-HANDLE) temp-table parameter from an AppServer application service as an SQL ResultSet parameter. The Open Client uses the same Java class to map both static and dynamic temp-table parameters. For more information on static and dynamic temp-tables, see Chapter 4, “Passing Parameters.”

Note: You can also map and pass temp-table parameters as ProDataGraph objects. For more information, see Chapter 5, “Accessing ABL ProDataSets.”

The following appendix sections provide information on:

- Preparing and managing temp-table parameters
- SQL ResultSet streaming and call-back sequence
- Passing a TABLE or TABLE-HANDLE as an INPUT parameter
- Passing a TABLE or TABLE-HANDLE as an OUTPUT parameter
- Passing a TABLE or TABLE-HANDLE as an INPUT-OUTPUT parameter
- Example Java client passing an SQL ResultSet parameter
Preparing and managing temp-table parameters

Java clients can access ABL TABLE or TABLE-HANDLE parameters using the following Java classes and interfaces:

- **INPUT parameters** — Any class that implements the java.sql.ResultSet interface.
- **OUTPUT parameters** — An instance of java.sql.ResultSetHolder, which contains a class that implements the com.progress.open4gl.ProResultSet interface, which extends the java.sql.ResultSet interface.
- **INPUT-OUTPUT parameters** — An instance of the com.progress.open4gl.ResultSetHolder class that contains a reference to a client-supported java.sql.ResultSet object instance. On output, the ResultSetHolder contains an instance of com.progress.open4gl.ProResultSet.

For more information on the java.sql.ResultSet interface, see the relevant JavaSoft publications.

OpenEdge provides the abstract class, com.progress.open4gl.InputResultSet, which provides part of an implementation of java.sql.ResultSet. You can extend this class to implement your own input ResultSet objects. For more information, see the “Passing a TABLE or TABLE-HANDLE as an INPUT parameter” section on page C–9.

The java.sql.ResultSet interface allows you to get the values of columns in the underlying temp-table. The extensions in the ProResultSet interface allow you to get the values of ABL fields, especially array fields, in a more ABL-like manner. For more information on how to access column and field values using the ProResultSet interface, see the “Accessing TABLE or TABLE-HANDLE parameters as OUTPUT” section on page C–14.

Data type mapping for temp-table fields

The Java default data type mapping for the individual fields of a temp-table is different than for non-temp-table parameters, to conform to the JDBC standard. Table C–1 shows the mappings.

Table C–1: SQL ResultSet data type mappings for temp-table fields

<table>
<thead>
<tr>
<th>Progress data type</th>
<th>Java data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>java.sql.Blob</td>
</tr>
<tr>
<td></td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>byte [ ]</td>
</tr>
<tr>
<td><strong>Note:</strong> For an INPUT temp-table, you can supply either of the two object types shown above. For output, your application can access a BLOB by calling any of three methods: getBlob() which returns a java.sql.Blob, getBytes() which returns a byte[ ], or getObject() which returns a java.sql.Blob.</td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>
### Table C–1: SQL ResultSet data type mappings for temp-table fields

<table>
<thead>
<tr>
<th>Progress data type</th>
<th>Java data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB</td>
<td>java.sql.Clob</td>
</tr>
<tr>
<td></td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>java.lang.String</td>
</tr>
<tr>
<td><strong>Note:</strong> For an INPUT temp-table, you can supply either of the two object types shown above. For output, your application can access a CLOB by calling any of three methods: getClob() which returns a java.sql.Clob, getString() which returns a java.lang.String, or getObject() which returns a java.sql.Clob.</td>
<td></td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>Long</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td></td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td><strong>Note:</strong> For an INPUT temp-table, you can supply either of the two object types shown above. For output, your application can access a date by calling any of three methods: getDate() which returns a java.sql.Date, getGregorianCalendar() which returns a java.util.GregorianCalendar, or getObject() which returns a java.sql.Date.</td>
<td></td>
</tr>
<tr>
<td>DATETIME</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td></td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td><strong>Note:</strong> For an INPUT temp-table, you can supply either of the two object types shown above. For output, your application can access a datetime by calling any of three methods: getTimestamp() which returns a java.sql.Timestamp, getGregorianCalendar() which returns a java.util.GregorianCalendar, or getObject() which returns a java.sql.Timestamp.</td>
<td></td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>byte[]</td>
</tr>
<tr>
<td>RECID</td>
<td>Long</td>
</tr>
<tr>
<td>ROWID</td>
<td>byte[]</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>Long</td>
</tr>
</tbody>
</table>
**Note:** The column order for ResultSet objects going between a Java Open Client and the AppServer always is the default field order (physical position of the fields in the table). Normally, temp-tables have the same default field order (_order) and user-defined field order (_field-rpos). However, if the temp-table is defined using the LIKE option for a database table, and the database table has set the user-defined field order to be different than the default field order, the Java client sees the fields in the default field order, not the user-defined field order. To avoid this, define the temp-table with the individual fields LIKE the database table fields, instead of defining the whole temp-table LIKE the database table.

## Accessing temp-table array fields

In ABL, fields in a temp-table can be defined with an extent value that creates a one-dimensional array of the specified data type, with the number of elements specified by the extent. JDBC ResultSet objects do not support arrays.

An array field in an ABL temp-table can be viewed as a ResultSet in two different ways. In the flat model, each array element becomes a separate column of the table. In the array model, each array is a single column of the table, with each array element being accessed by an index on the array column. The array model matches how ABL handles array fields.

For Java clients, there are access methods to support both models. These methods access field values and meta data of an output ResultSet.

For input, the proxy assumes the fields use the flat column model. For output, your application can access array fields using either the flat column model or array model. For more information, see the “Accessing TABLE or TABLE-HANDLE parameters as OUTPUT” section on page C–14.

The following examples use the Java flat model meta data methods `getColumnType()` and `getColumnCount()` and the array model methods `getFieldProtoType()` and `getFieldCount()`. These examples illustrate the use of a temp-table with three fields in a Java client application, where the second field is an array of 12 elements.

### Using the flat model

The flat model methods behave as follows:

- `getColumnCount()` returns the value 14
- `getColumnType(4)` gets the data type of the third element of the second TABLE field (which is the array field)
- `getColumnType(14)` gets the data type of the third TABLE field

Since all elements of an array field have the same meta data information, `getColumnType(2)` through `getColumnType(13)` return the same value.
Using the array model

The array model methods behave as follows:

- `getFieldCount()` returns the value 3
- `getFieldProType(2)` gets the data type of the second TABLE field
- `getFieldProType(3)` gets the data type of the third TABLE field

In this model, since all elements of an array field have the same meta data information, it is necessary to pass only the field index (without an array item index) to these meta data methods. For methods that get a field value, each method has an extra parameter to specify an array index. For example, the method to get the value of a field as an object is as follows:

```
getObject(int fieldNum, int arrayIndex)
```
SQL ResultSet streaming and call-back sequence

Java Open Client support for TABLE and TABLE-HANDLE parameters depends on a call-back mechanism used by both the client application and the proxy code. Therefore, either the client application or the proxy always must request data from the sender, rather than having the sender set or send the data. As a result, the client application calls back to the proxy for output parameters, and the proxy calls back to the client application for input parameters.

Table C–2 shows how the call-back sequence works when passing an INPUT TABLE or TABLE-HANDLE parameter.

**Table C–2: Passing an INPUT temp-table as an SQL ResultSet**

<table>
<thead>
<tr>
<th>Client application</th>
<th>Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Creates input ResultSet object</td>
<td></td>
</tr>
<tr>
<td>2) Passes ResultSet object as a parameter to proxy</td>
<td></td>
</tr>
<tr>
<td>3) Uses methods on the input ResultSet object to get data</td>
<td></td>
</tr>
<tr>
<td>4) Passes input data to the AppServer</td>
<td></td>
</tr>
</tbody>
</table>

Table C–3 shows how the call-back sequence works when passing an OUTPUT TABLE or TABLE-HANDLE parameter.

**Table C–3: Passing an OUTPUT temp-table as an SQL ResultSet**

<table>
<thead>
<tr>
<th>Client application</th>
<th>Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Creates holder object for the output ResultSet</td>
<td></td>
</tr>
<tr>
<td>2) Passes holder object to proxy</td>
<td></td>
</tr>
<tr>
<td>3) Gets output data from the AppServer</td>
<td></td>
</tr>
<tr>
<td>4) Creates output ResultSet object and stores it in holder object</td>
<td></td>
</tr>
<tr>
<td>5) Returns to client</td>
<td></td>
</tr>
<tr>
<td>6) Uses methods on the ResultSet contained in the holder object</td>
<td></td>
</tr>
</tbody>
</table>
Table C–4 shows how the call-back sequence works when passing an INPUT-OUTPUT TABLE or TABLE-HANDLE parameter.

**Table C–4: Passing an INPUT-OUTPUT temp-table as an SQL ResultSet**

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Creates holder object</td>
<td></td>
</tr>
<tr>
<td>2) Creates input ResultSet object and stores it in holder</td>
<td>4) Uses methods on the ResultSet in the holder object to get data</td>
</tr>
<tr>
<td>3) Passes holder object to proxy</td>
<td>5) Passes input data to the AppServer</td>
</tr>
<tr>
<td>6) Gets output data from the AppServer</td>
<td>7) Creates output ResultSet object and stores it in holder object</td>
</tr>
<tr>
<td>8) Returns to client</td>
<td></td>
</tr>
<tr>
<td>9) Uses methods on the ResultSet in the holder object to get data</td>
<td></td>
</tr>
</tbody>
</table>

The input ResultSet object is an object written by the Java client application. This also can be a standard ResultSet object available as the result of a SQL query (a java.sql.ResultSet). In contrast, the output ResultSet object always is provided by the generated proxy.

**INPUT ResultSet objects**

For INPUT (or the INPUT side of INPUT-OUTPUT parameters), all the data must be available before the proxy method call is made. The methods of the input ResultSet object (which feed the input data to the proxy) may not call any other AppServer methods on the proxy that share the same connection. If they do, an error results.

**OUTPUT ResultSet objects**

On OUTPUT (or the OUTPUT side of INPUT-OUTPUT parameters), the ResultSet is not necessarily finished being transmitted to the client by the time the method call returns. The proxy buffers data for the ResultSet and makes it available sequentially to the client.

This process of buffering an OUTPUT ResultSet from the AppServer and making it sequentially available to the client is called *streaming*. An AppServer connection that is streaming a ResultSet is said to be in a STREAMING state. This STREAMING state remains in effect until all data is received by the client or the ResultSet is closed.

When the method containing an output ResultSet parameter returns, the client gets an object it can use to call back for more data; however, the client cannot make any other AppServer method calls while the connection is in the STREAMING state. To determine whether a connection is in the streaming state, invoke the _isStreaming_() method for any proxy object using the connection.
Order of access for OUTPUT ResultSet objects

The stream of data in an OUTPUT ResultSet can be thought of as a one-directional tape. It must be read in sequence, and there is no way to get the same data more than once or go backwards. When getting the data from a single TABLE or TABLE-HANDLE, the data for each row and each field in that row must be requested in sequence, and no request can be repeated. Also, if there is more than one OUTPUT TABLE or TABLE-HANDLE parameter for a method, they must be perused in sequence. Data from the first TABLE or TABLE-HANDLE parameter must be completely fetched or the ResultSet closed before any data can be retrieved from the second, and so on. It is possible, however, to skip data. For example, you can fetch the value of field 1 and then the value of field 10; however, you cannot then reverse and obtain the values for fields 2 through 9.

Closing OUTPUT ResultSet objects

To close an OUTPUT ResultSet, you generally execute the close() method on the ResultSet as appropriate for your Java Open Client. You also can close the ResultSet by executing the _release() method on the proxy objects that returned the ResultSet. Releasing the last such AppObject, SubAppObject, or ProcObject disconnects the Open Client from the AppServer and closes any OUTPUT ResultSet objects it provides.

Result set schemas

Every ResultSet object (INPUT and OUTPUT) has a schema associated with it. The schema describes the number, data type, name and extent of the columns.

This schema information for an output ResultSet is available to the Java client as a ResultSetMetaData object through the getMetaData() method of the ResultSet object. This information can be useful to the client if it wants to write generic routines to handle temp-table data, without hard coding specifics about the number of fields, their names, and their types.
Passing a TABLE or TABLE-HANDLE as an INPUT parameter

When the client application passes a static temp-table as an INPUT parameter, the client provides an object that contains data for the table. In this instance, the proxy already knows the schema (meta data) for the table; therefore, the client application does not need to supply the schema.

When the client application passes a TABLE-HANDLE parameter, the proxy does not know the schema (meta data) for the table. The client must provide both the schema and data for the TABLE-HANDLE parameters.

For an INPUT TABLE or TABLE-HANDLE parameter, you provide an instance of a class that implements the java.sql.ResultSet interface. You then pass this class instance directly as the INPUT TABLE or TABLE-HANDLE parameter.

Instantiating the INPUT parameter

To provide an appropriate class instance as an INPUT parameter, your client application must complete one of the following tasks:

- Extend the com.progress.open4gl.InputResultSet class, and implement the next() and getObject() methods, as described in the following section
- Obtain a ResultSet instance by running a method on some java.sql object (for example, run the executeQuery() method on an instance of java.sql.Statement)

Using com.progress.open4gl.InputResultSet

The com.progress.open4gl.InputResultSet abstract class implements the java.sql.ResultSet interface except for two methods that you must implement in your extended class. The proxy calls back through these methods to get the data to pass to ABL.

There are two methods you must implement. The first is:

Syntax

```java
boolean next()
```

This advances the ResultSet cursor to the next row. The cursor always is initially positioned before the first row of the ResultSet; as a result, the first time next() is called the cursor moves to the first row. It returns false when there are no more rows.

**Caution:** Make sure the ResultSet cursor is positioned before the first row if you plan to pass the InputResultSet as an input parameter, and you want the receiving context to have access to all rows from the beginning of the InputResultSet. Only rows after the current cursor position are passed to the AppServer.
The second method is:

**Syntax**

```java
Object getObject(int columnNum)
```

*columnNum*

Obtains the value of an individual column as an object. The `columnNum` parameter is a 1-based index indicating which column value to get. This must correctly map to the order in which the temp-table fields are defined in the ABL procedure.

If there is an array field in the temp-table, it is treated as a flattened set of columns by this method. On input, since the proxy is calling back to the client, there can be only one way to get the data, and that one way assumes the flat column model native to the standard JDBC ResultSet.

Your implementation of these methods can differ depending on the data source for the input temp-table. For example, the data might come from an in-memory vector or be read from a file. The exact implementation, therefore, depends on the specific requirements of your application.

You can use the following methods with the `InputResultSet` class to omit all schema information when marshaling data from the Java Open Client to the AppServer. Using these methods suppresses index descriptions and all field information and aids in faster transmission of data, thereby increasing the performance of your application. You can use these methods when the receiving side knows the schema definition for the table and validation is not necessary.

The following method tells the proxy to marshal the specified `InputResultSet` from the Java client to the AppServer without schema information, when `flag` is set to `true`:

**Syntax**

```java
void setNoSchemaMarshal(boolean flag)
```

The corresponding ABL temp-table on the AppServer must have the same schema as the ResultSet on the client.

The Java Open Client generates a `RunTime4GLErrorException` when:

- The ABL temp-table’s schema does not match the incoming data from the Java client
- `flag` is set to `true` and the `InputResultSet` is passed to a `TABLE-HANDLE` parameter on the AppServer

The following method on the `InputResultSet` returns `true` if the proxy will not marshal schema information to the AppServer; it returns `false` if it will marshall information:

```java
boolean getNoSchemaMarshal()
```
Providing metadata for INPUT TABLE-HANDLE parameters

For TABLE parameters, the proxy calls the next() and getObject() methods to get the data that needs to be passed to the AppServer methods. For a TABLE-HANDLE parameter, the proxy must first obtain a ResultSetMetaData object by calling the getMetaData() method in order to determine the format of the data.

There are several alternatives for how getMetaData() can supply the appropriate ResultSetMetaData object:

- Use a standard JDBC ResultSet if:
  - The input ResultSet is a standard JDBC ResultSet obtained from a query, and the client application does not need to do anything
  - The ResultSet instance already implements the getMetaData() method and returns a standard ResultSetMetaData object.

The proxy automatically maps SQL data types to ABL schema data types, as shown in Table C–5.

Note: Because a standard ResultSet supports a standard ResultSetMetaData object and not the ABL extension (ProResultSetMetaData), there can be no array fields.

Table C–5: Mapping between SQL and ABL schema data types for standard JDBC ResultSet

<table>
<thead>
<tr>
<th>SQL type</th>
<th>ABL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>INT64</td>
</tr>
<tr>
<td>BIT</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATETIME</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>RAW</td>
</tr>
</tbody>
</table>

If the ResultSet contains any other type, an exception is thrown.
• Extend the OpenEdge InputResultSet if:

  – The ResultSet contains a column type other than the types shown in Table C–5 and the column can be converted to a compliant type, implement your own ResultSet class instead of passing the java.sql.ResultSet directly as the parameter. For example, you can extend com.progress.open4gl.InputResultSet with your own class.

    When getObject() is called on your class, it retrieves data from the corresponding standard object and converts the data before returning it to the proxy. When the getMetaData() method is called, the implementation of your extended class must supply the appropriate meta data. To do this, the application must create and populate a ProResultSetMetaDataImpl object, as described in the “ProResultSetMetaDataImpl class” section on page C–12”.

  – The application requires the resulting server-side temp-table to have an array field, this is another reason for you to extend the InputResultSet class. This time, the implementation of both next() and getObject() call the standard instance’s corresponding methods to do the work. When the getMetaData() method is called, however, the extending class must supply the required meta data, with the flattened schema view converted to the corresponding array field view. Again, to do this, the application must create and populate a ProResultSetMetaDataImpl object, as described in the “ProResultSetMetaDataImpl class” section on page C–12”.

• Use the ResultSetMetaData from a previously obtained output ResultSet under the following condition:

  – If the input ResultSet is a ResultSet that was first obtained from the AppServer through an OUTPUT parameter, meta data can be obtained from that output set by calling getMetaData() on it. You can think of this as an OUTPUT-INPUT TABLE-HANDLE, parameter even though it is output from one method call and input through another. The ResultSetMetaData object obtained from the output ResultSet can then be provided when getMetaData() is called for the input ResultSet.

ProResultSetMetaDataImpl class

If the meta data for the input ResultSet must be supplied by code implemented in the client application, the getMetaData() method must be implemented to return an instance of ResultSetMetaData. The recommended way to do this is to use the com.progress.open4gl.ProResultSetMetaDataImpl class provided in the Open Client Toolkit. This is an implementation of the ProResultSetMetaData interface, the same interface that is used by the client when getting output data.

The only two methods of ProResultSetMetaDataImpl that the application must call are the constructor:

Syntax

ProResultSetMetaDataImpl(int numFields)
And the `setFieldMetaData()` method:

**Syntax**

```java
setFieldMetaData(int fieldNumber, String Name, int extent, int type)
```

The `setFieldMetaData()` method requires the following:

- `fieldNumber` must be between 1 and the number of fields.
- `Name` cannot be null.
- The value of `Name` must be unique within the table.
- `extent` must be zero or greater (for BLOB and CLOB fields, it must be zero or one).
- `type` must be one of the class constant values defined in `com.progress.open4gl.Parameter`. For more information, see the information on specifying data type meta data for temp-tables in Chapter 4, “Passing Parameters.”

**Note:** The Open Client Toolkit class documentation in `OpenEdge-install-directory/java/doc/` also describes these methods used by the client application for the `ProResultSetMetaDataImpl` class.

To pass a self-implemented `ResultSet` for a `TABLE-HANDLE` parameter, the application must:

1. Create a new class that extends `com.progress.open4gl.InputResultSet`, to provide access to the data to be passed to the server. This object’s implementation includes code for `next()` and `getObject()` (as for a static temp-table) and for `getMetaData()`, which should return the `com.progress.open4gl.ProResultSetMetaDataImpl` instance when called.

2. Create an instance of the class you created in Step 1.


4. Call `ProResultSetMetaDataImpl.setFieldMetaData()` for each field of the table.

5. Make the `ProResultSetMetaDataImpl` instance available to the `InputResultSet` object. You can do this in any way you like, for example, by adding a `setMetaVdata()` method to your `InputResultSet` implementation.

6. Make the proxy method call passing the `com.progress.open4gl.InputResultSet` instance as the parameter.

**Note:** Step 3 and Step 4 can be implemented in `getMetaData()` which eliminates the need for Step 5.
Passing Temp-tables as SQL ResultSet Parameters

Passing a TABLE or TABLE-HANDLE as an OUTPUT parameter

An application uses the same model when working with an OUTPUT TABLE or TABLE-HANDLE parameter as with an INPUT-OUTPUT parameter, by having the client pass a ResultSetHolder object. The data is returned through a ResultSet object from which a meta data object can be obtained. Although the model is the same, TABLE-HANDLES are different in that they can be Unknown. As a result, you must ensure the client can handle these cases.

For an OUTPUT parameter, you must pass an instance of com.progress.open4gl.ResultSetHolder, which is the holder class for a ProResultSet object. The contents of this object are left empty by the client. On return, the holder’s content is set to a com.progress.open4gl.ProResultSet object. When the method returns, the Java client calls back through this object’s methods to get the data. For more information on com.progress.open4gl.ProResultSet, see the “Accessing TABLE or TABLE-HANDLE parameters as OUTPUT” section on page C–14.

If the parameter is a TABLE-HANDLE, the value returned can be unknown. As a result, calling getResultSetValue() on the holder returns null.

Accessing TABLE or TABLE-HANDLE parameters as OUTPUT

For both output TABLE and TABLE-HANDLE parameters and INPUT-OUTPUT TABLE and TABLE-HANDLE parameters (on output), you must obtain the output ProResultSet object from the ResultSetHolder class and call its methods to get the data. As explained previously, com.progress.open4gl.ProResultSet extends the java.sql.ResultSet interface.

In addition to the standard flat model for viewing individual columns presented by ResultSet objects, ProResultSet allows you to view the object using the ABL array model, which presents a temp-table as individual fields that can contain arrays. You can choose the model that is most natural or convenient. For more information on the ABL field model, see OpenEdge Development: Open Client Introduction and Programming.

Standard methods in the ProResultSet interface

The standard methods include the methods of java.sql.ResultSet, which use the flat column model. The primary methods in this set are described below.

The following method advances the ResultSet cursor to the next row:

Syntax

```java
boolean next()
```

The following method closes the ResultSet:

Syntax

```java
void close()
```

For output ResultSet objects, the close() method may be called to close the ResultSet before all the rows have been fetched. No other AppServer method can be called until all the rows are fetched or close() is called.
The following method returns the value of the column identified by the 1-based \texttt{columnNum}:

**Syntax**

\begin{verbatim}
Object getObject(int columnNum)
\end{verbatim}

The data type of the returned object is the default data type of the column. For example, the default data type for an ABL \texttt{CHARACTER} field is \texttt{java.lang.String}. For more information, see the “Data type mapping for temp-table fields” section on page C–2.

The following method returns the value of the column identified by the \texttt{fieldName}:

**Syntax**

\begin{verbatim}
Object getObject(String fieldName)
\end{verbatim}

The data type of the returned object is the default data type of the field. For example, the default data type for an ABL \texttt{CHARACTER} field is \texttt{java.lang.String}. For more information, see the “Data type mapping for temp-table fields” section on page C–2.

The following method returns the value of the column identified by the 1-based \texttt{columnNum}:

**Syntax**

\begin{verbatim}
DataType getDataType(int columnNum)
\end{verbatim}

\texttt{DataType} is the name of the data type, for example, \texttt{String getString()}, \texttt{Integer getInteger()}, and so on.

The following method returns the value of the column identified by the \texttt{fieldName}:

**Syntax**

\begin{verbatim}
DataType getDataType(String fieldName)
\end{verbatim}

\texttt{DataType} is the name of the data type; for example, \texttt{String getString()}, \texttt{Integer getInteger()}, and so on.

If the specified return \texttt{DataType} is different from the default data type of the column, the Open Client takes one of the following actions:

- The value is automatically converted from the default data type to \texttt{DataType}
- An exception is thrown with a message explaining that the conversion is not supported

For example, if the ABL data type is \texttt{INTEGER} and its value is 10, \texttt{getString(columnNum)} automatically converts the value from \texttt{Integer} to the \texttt{java.lang.String}, “10”. If the application calls \texttt{getDate(columnNum)}, an exception is thrown with the message that an \texttt{Integer} value cannot be converted to \texttt{Date}. For information on the supported conversions, see Table C–6.
ProResultSet interface extensions to java.sql.ResultSet

ProResultSet adds the following methods not found in java.sql.ResultSet, where DataType is the name of the data type. These extensions allow your application to access the data in a ResultSet based on the ABL array model (see the “Accessing temp-table array fields” section on page C–4).

The following method returns the value of the arrayIndex element of the array field identified by fieldNum:

**Syntax**

```
Object getObject(int fieldNum, int arrayIndex)
```

The data type of the object is the default data type of the arrayIndex. The arrayIndex and fieldNum are 1-based.

The following method returns the value of the arrayIndex element of the array field identified by fieldName:

**Syntax**

```
Object getObject(String fieldName, int arrayIndex)
```

The data type of the object is the default data type of the array field. The arrayIndex and fieldName are 1-based.

The following method returns the array field identified by fieldNum:

**Syntax**

```
DataType getDataType(int fieldNum, int arrayIndex)
```

**Syntax**

```
DataType getDataType(String fieldName, int arrayIndex)
```

DataType is the name of the data type; for example, String getString(), Integer getInteger(), and so on. The arrayIndex and fieldNum are 1-based.

The following method returns the array field:

**Syntax**

```
String getString(int fieldNum, int arrayIndex)
```

```
String getString(String fieldName, int arrayIndex)
```

```
Integer getInteger(int fieldNum, int arrayIndex)
```

```
Integer getInteger(String fieldName, int arrayIndex)
```

For example, the following is the syntax for methods that return a value for a String and an Integer array element of a temp-table field:
For a complete listing of the methods implemented by the ProResultSet interface, see the documentation in OpenEdge-install-directory/java/doc/.

**Temp-table field data type conversion**

When the `getDataType()` method you use to get the value of a temp-table field does not match the default Java data type of the field (Table C–1), a valid conversion occurs as long as the `getDataType()` method corresponds to the ABL data type, as shown in Table C–6. Otherwise, an exception is thrown with a message explaining that the conversion is not supported. For example, if the temp-table field is defined as LOGICAL, you can use any of `getString()`, `getInt()`, or `getBoolean()` to retrieve the field value.

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Valid Java data type accessor methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td><code>byte[]</code> <code>getBytes()</code> <code>java.sql.Blob</code> <code>getBlob()</code></td>
</tr>
<tr>
<td>CHARACTER</td>
<td><code>java.lang.String</code> <code>getString()</code></td>
</tr>
<tr>
<td>CLOB</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>java.sql.Clob</code> <code>getClob()</code></td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>int</code> <code>getInt()</code> <code>long</code> <code>getLong()</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>java.sql.Date</code> <code>getDate()</code> <code>java.util.GregorianCalendar</code> <code>getGregorianCalendar()</code></td>
</tr>
<tr>
<td>DATETIME</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>java.sql.Timestamp</code> <code>getTimeStamp()</code> <code>java.util.GregorianCalendar</code> <code>getGregorianCalendar()</code></td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>java.util.GregorianCalendar</code> <code>getGregorianCalendar()</code></td>
</tr>
<tr>
<td>DECIMAL</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>int</code> <code>getInt()</code> <code>long</code> <code>getLong()</code> <code>double</code> <code>getDouble()</code> <code>java.math.BigDecimal</code> <code>getBigDecimal()</code></td>
</tr>
<tr>
<td>INT64</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>int</code> <code>getInt()</code> <code>long</code> <code>getLong()</code> <code>double</code> <code>getDouble()</code> <code>java.math.BigDecimal</code> <code>getBigDecimal()</code></td>
</tr>
<tr>
<td>INTEGER</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>int</code> <code>getInt()</code> <code>long</code> <code>getLong()</code> <code>double</code> <code>getDouble()</code> <code>java.math.BigDecimal</code> <code>getBigDecimal()</code></td>
</tr>
<tr>
<td>LOGICAL</td>
<td><code>java.lang.String</code> <code>getString()</code> <code>int</code> <code>getInt()</code> <code>boolean</code> <code>getBoolean()</code></td>
</tr>
<tr>
<td>RAW</td>
<td><code>byte[]</code> <code>getBytes()</code></td>
</tr>
</tbody>
</table>

Table C–6: ProResultSet type conversions for temp-table fields (1 of 2)
### Getting meta data for OUTPUT TABLE and TABLE-HANDLE parameters

You can obtain schema information for a standard ResultSet object by accessing the java.sql.ResultSetMetaData object returned by the getMetaData() method on the ResultSet. This ResultSetMetaData object allows you to access the schema information for each column in the ResultSet.

The com.progress.open4gl.ProResultSetMetaData interface is an OpenEdge extension of the java.sql.ResultSetMetaData interface. The extensions in ProResultSetMetaData access the schema information for the ProResultSet object viewed as temp-table fields, using the array model. To use these extensions, you must cast the ResultSetMetaData object returned from getMetaData() as a ProResultSetMetaData object. For example:

```java
// In this code, resultOut is a ProResultSet obtained as an output parameter.
java.sql.ResultSetMetaData metaData = resultOut.getMetaData();

// To access the extended set of methods the application does:
com.progress.open4gl.ProResultSetMetaData proMetaData =
    (com.progress.open4gl.ProResultSetMetaData)metaData;

// Now we use proMetaData to access the ProResultSetMetaData extensions

The rest of this section describes the ProResultSetMetaData methods and how they allow you to view a ProResultSet object both as standard ResultSet columns, using the flat model, and as temp-table fields, using the array model.

---

**Table C–6: ProResultSet type conversions for temp-table fields**

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Valid Java data type accessor methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECID</td>
<td>java.lang.String getString() int getInt() long getLong()</td>
</tr>
<tr>
<td>ROWID</td>
<td>byte[] getBytes()</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>java.lang.String getString() int getInt() long getLong()</td>
</tr>
</tbody>
</table>
Standard methods in the ProResultSetMetaData class

The ProResultSetMetaData class has the following standard public methods, which allow you to view the elements of ABL array fields as individual columns, using the flat model:

- The following methods are part of the standard java.sql.ResultSetMetaData interface and are useful if the client wants to write standard code that could access any JDBC ResultSet:

  ```java
  int getColumnCount() throws ProSQLException
  String getColumnName(int columnNum) throws ProSQLException
  int getColumnType(int columnNum) throws ProSQLException
  String getColumnTypeName(int columnNum) throws ProSQLException
  ```

The `getColumnType()` method returns an int constant for the standard SQL types as shown in Table C–7. The SQL data type column in the table shows the type names as specified by the java.sql.Types class. For example:

  ```java
  String getColumnTypeName(int columnNum) throws ProSQLException
  ```

The `getColumnTypeName()` method returns the corresponding ABL data type names shown in Table C–7 under ABL data type.

### Table C–7: ABL to SQL data type mappings

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>LONGVARCHAR</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMPTZ</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>INT64</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>BIT</td>
</tr>
<tr>
<td>RAW</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>RECID</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>
Table C–7:  ABL to SQL data type mappings

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWID</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>

**ProResultSetMetaData class extension to java.sql.ResultSetMetaData**

All the methods based directly on the standard java.sql.ResultSetMetaData class view the temp-table using the standard SQL flat column model (columns without arrays)

These methods all view the columns as fields with arrays:

- These methods let you see the data type of each ResultSet column as ABL and Java types:

  ```java
  int getColumnProType(int columnNum) throws ProSQLException
  ```

  The following returns one of the constant values accessible from com.progress.open4gl.Parameter:

  ```java
  String getColumnJavaTypeName(int columnNum) throws ProSQLException
  ```

  getColumnJavaTypeName() returns the name of the Java class to which each ABL data type maps. For information on these mappings, see Table C–1.

  For more information, see the section on specifying data type meta data for temp-tables in Chapter 4, “Passing Parameters.”

- The following methods provide access to the schema of temp-table parameters that contain array fields, viewed from an ABL viewpoint:

  ```java
  int getFieldCount() throws ProSQLException
  ```

  ```java
  StringgetFieldName(int fieldNum) throws ProSQLException
  ```

  ```java
  int getFieldExtent(int fieldNum) throws ProSQLException
  ```

  ```java
  int getFieldProType(int fieldNum) throws ProSQLException
  ```

  ```java
  String getFieldJavaTypeName(int fieldNum) throws ProSQLException
  ```

  ```java
  StringgetFieldTypeName(int fieldNum) throws ProSQLException
  ```
The ABL-oriented methods, which view the temp-table using the ABL array field model (fields that can include arrays), have prototypes that refer to Field or field. For more information on the relationship between ResultSet columns and ABL array fields, see the “Accessing temp-table array fields” section on page C–4.

Although you can access the data using either the SQL (flat) or ABL (array) model, both provide equivalent functionality. Table C–8 shows the correspondence between the ProResultSetMetaData methods.

Table C–8: Comparing some array and flat model methods

<table>
<thead>
<tr>
<th>This array model method returns the same set of values . . .</th>
<th>As this flat model method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFieldName()</td>
<td>getName()</td>
</tr>
<tr>
<td>getFieldProType()</td>
<td>getNameProType()</td>
</tr>
<tr>
<td>getFieldJavaTypeName()</td>
<td>getNameJavaTypeName()</td>
</tr>
<tr>
<td>getType()</td>
<td>getName()</td>
</tr>
<tr>
<td>getType()</td>
<td>getNameProType()</td>
</tr>
<tr>
<td>getType()</td>
<td>getNameJavaTypeName()</td>
</tr>
<tr>
<td>getType()</td>
<td>getName()</td>
</tr>
</tbody>
</table>
Passing a TABLE or TABLE-HANDLE as an INPUT-OUTPUT parameter

INPUT-OUTPUT TABLE and TABLE-HANDLE parameters are a combination of INPUT and OUTPUT parameters.

For an INPUT-OUTPUT parameter, you must pass an instance of `com.progress.open4gl.ResultSetHolder`, which contains a reference to a client-supplied `java.sql.ResultSet` object instance. On output, the holder object contents are changed to reference a `com.progress.open4gl.ProResultSet` object instance.

If the parameter is a TABLE-HANDLE, the value returned can be unknown. As a result, calling `getResultSetValue()` on the holder returns null.

For more information on creating a `java.sql.ResultSet` for input, see the “Passing a TABLE or TABLE-HANDLE as an INPUT parameter” section on page C–9. For more information on `ProResultSet` objects, see the “Accessing TABLE or TABLE-HANDLE parameters as OUTPUT” section on page C–14.
Example Java client passing an SQL ResultSet parameter

The following procedure supports a small portion of an accounting system. The AccountInfo procedure is expected to run persistently and has an internal procedure called getPaymentsInfo:

Note: These samples are not available on the OpenEdge product DVD or the PSDN Web site.

AccountInfo.p

```pascal
/* AccountInfo.p Procedure */
DEFINE INPUT PARAM account-num AS INTEGER.
DEFINE TEMP-TABLE payee-list
  FIELD name AS CHAR
  FIELD id AS INTEGER.
DEFINE TEMP-TABLE payments-info
  FIELD payment-date AS DATE
  FIELD payee-id AS INTEGER
  FIELD payee-name AS CHAR
  FIELD amount AS DECIMAL
  FIELD cleared AS LOGICAL.

/* ... AccountInfo.p code ... */

/* The Internal Procedure getPaymentsInfo returns payment records. This is information about payments done after a specific date to payees who belong to the payeeList set */
PROCEDURE getPaymentsInfo:
  DEFINE INPUT PARAM from-date AS DATE.
  DEFINE INPUT PARAM TABLE FOR payee-list.
  DEFINE OUTPUT PARAM payments-num AS INTEGER.
  DEFINE OUTPUT PARAM TABLE FOR payments-info.

  /* ... getPaymentsInfo Code ... */
END PROCEDURE.
```
The code in Example C–1 demonstrates how to create and use AppObjects and ProcObjects from a Java client. In this example, the AppObject was defined with ProxyGen as Account.

**Example C–1: Java client using proxy objects**

```java
// Creates and connects to the AppObject.
Account account = new Account("AppServer://myhost:2290",
        "user", "password", null);

// Creates a Persistent Procedure instance for account 777
AccountInfo info = account.createPO_AccountInfo(777);

// Gets information about payments done after a specific date to
// payees who belong to the payeeList set
java.util.GregorianCalendar fromDate; // Starting date.
java.sql.ResultSet payeeList; // List of payees.
com.progress.open4gl.IntHolder paymentsNum; // # of returned payment
    // records.
com.progress.open4gl.ResultSetHolder payHolder; // Holder for record set.

fromDate = new java.util.GregorianCalendar(1982, 6, 31);
payeeList = new PayeeList(); // See Example 4-4 for PayeeList class.
paymentsNum = new com.progress.open4gl.IntHolder();
payHolder = new com.progress.open4gl.ResultSetHolder();

// Makes the call
info.getPaymentsInfo(fromDate, payeeList, paymentsNum, payHolder);
// Prints the # of payment records.
System.out.println("The number of records is: " +
        paymentsNum.getIntValue());

// Extracts payment records.
java.sql.ResultSet paymentsInfo = payHolder.getResultSetValue();
// Prints all the payment records.
while (paymentsInfo.next())
{
    // Gets the next record.
    int colNum = paymentsInfo.getMetaData().getColumnCount();
    // Prints the columns of the current record.
    for (int i = 1; i <= colNum; i++)
        System.out.println (paymentsInfo.getObject(i));
}

// Releases the ProcObject.
info._release();

// Releases the AppObject. Since this is the last object to
// share the connection this disconnects from the AppServer.
account._release();
```
Example C–2 shows the PayeeList class for Example C–1. This class implements the passing of a temp-table parameter as an SQL ResultSet.

Example C–2: Java input ResultSet example

```java
import java.sql.*;
import java.util.Vector;

/* An example of a simple implementation of payeeList INPUT ResultSet. */
public class PayeeList extends com.progress.open4gl.InputResultSet
{
    private Vector rows;
    private int rowNum;
    private Vector currentRow;

    // Create a payee ResultSet with two payees.
    // Each has an Id and a name.
    PayeeList ()
    {
        rows = new Vector();
        Vector row;
        // {431, "Bay Plumbing"} payee.
        row = new Vector();
        row.addElement(new Integer(431));
        row.addElement(new String("Bay Plumbing"));
        rows.addElement(row);
        // {711, "Laura's Gifts"} payee.
        row = new Vector();
        row.addElement(new Integer(711));
        row.addElement(new String("Laura's Gifts"));
        rows.addElement(row);
        currentRow = null;
        rowNum = 0;
    }

    // Position the cursor on the next row.
    public boolean next() throws SQLException
    {
        try
        {
            currentRow = (Vector)rows.elementAt(rowNum++);
        }
        catch (Exception e) {return false;}
        return true;
    }

    // Return the nth object of the row when n = pos. In this example, pos can be
    // 1 or 2.
    public Object getObject(int pos) throws SQLException
    {
        return currentRow.elementAt(pos-1);
    }
}
```
Index

A

ABL methods 2–3
ABL unknown value. See Unknown value (?)

absolute( ) method
SDOResultSet 9–17, 9–25

acceptChanges( ) method
ProDataGraph 5–31

Accessing
ABL arrays in SQL ResultSet parameters C–4
ProChangeSummary instances 5–25
ProDataObject instances
  using known temp-table name 5–13
  using related table rows 5–27
Progress ProDataSets 5–1
proxy properties
  command line 7–2
  Connection class 7–3
  methods 7–4
  RunTimeProperties class 7–3

Accessor methods
  general property
    Connection class 7–5
    RunTimeProperties class 7–4
  purposed property 7–6

addDataRelation( ) method
ProDataGraphMetaData 5–50

Additional Java permission
  for applets 10–3

addParameter( ) method 11–16

addProDataObject( ) method
ProDataGraph 5–32

addTable( ) method
ProDataGraphMetaData 5–51

Applets 10–1
  adding to an HTML page 10–6
  Internet Explorer 10–6
    multiple browser support 10–8
  Netscape Navigator 10–7
  additional Java permissions 10–3
  multiple browser support 10–4
  Open Client support 10–1
  requesting permission 10–2
  sample Open Client 10–9
  signing 10–5

Application services
ABL methods 2–3
proxy objects 2–2

AppObjects
  constructors 3–3
  example 2–11
  generation 2–2
  instantiating 3–3
  session-free method execution 2–9
  SmartDataObject access 9–5

AppServer
  establishing a connection 3–3
  operating modes 3–5

AppServer connection procedure 3–4

Array model for SQL ResultSets
  array fields in temp-tables C–4
  behavior C–5
  methods compared to flat model C–21
  ProResultSet extended methods C–16
Index

ProResultSetMetaData extended methods C–20
Arrays
parameter mappings 4–3
Unknown value (?) 4–4
attachToAppObj( ) method
SDOResultSet 9–17

B
Basic packages, Open Client Runtime 1–2
beforeFirst( ) method
SDOResultSet 9–18
beginLogging( ) method
ProChangeSummary 5–48
boolean wasNull( ) method
SDOResultSet 9–25
Building Java client applications 1–8

C
Call-back sequence SQL ResultSets C–6
_cancelAllRequests( ) method 2–5
cancelBatch( ) method
SDOResultSet 9–23
cancelRowUpdates( ) method
SDOResultSet 9–21
Certificate store files B–2
certificateStore property 7–15
Change tracking
ProDataGraph 5–6
ChangeSummary interface
Java SDO features 5–5
See also ProChangeSummary class
Class factory methods 2–4
clear( ) method 11–8
Client programming
ABL unknown values 4–4
applets 10–1
connecting to an AppServer 3–1
exception handling 8–2
ProDataSet parameters 5–1
temp-table parameters
ProDataGraphs 5–29
SQL ResultSets C–1
close( ) method
ProResultSet C–14
SDOResultSet 9–24
Closing OUTPUT SQL ResultSets C–8
Column properties 5–4
Common proxy methods 2–5
Connecting to an AppServer using proxy objects 3–1
using the OpenAPI 11–3
Connection
class 3–2
definition 7–3
general accessor methods 7–5
modifying object properties 7–14
relationship to RunTimeProperties class 7–12
releasing objects 3–5
session-free AppObjects 2–9
methods 2–3, 3–1
pool
session-free AppObject sharing 2–9
connection procedure 3–4
connectionLifetime property 7–15
Constructors
AppObject 3–3
element classes extending proxy objects 6–2
Holder classes 4–4
Java OpenAPI
ProDataGraph 11–26
ProDataGraphMetaData 11–25
ProDataObjectMetaData 11–28
ProDataRelationMetaData 11–25
ProResultSetMetaDataImpl 11–22
OpenAppObject 11–3
ParamArray 11–7
ProDataGraph 5–30
ProDataGraphMetaData 5–50
ProDataObjectMetaData 5–53
ProDataRelationMetaData 5–59
ProResultSetMetaDataImpl C–12
SDOAppObject 9–8
SDOResultSet objects 9–9
CreateAO_SubAppObject( ) method 2–4
CreatePO_ProcObject( ) method 2–4
createPO( ) method
OpenAppObject 11–5
createProDataObject( ) method
D

Data type mappings
ABL to Java proxy 4–2
ABL unknown values 4–4
INPUT parameters 4–2
OUTPUT and INPUT-OUTPUT parameters 4–4
Progress to SQL ResultSet (OUTPUT) C–19
SQL ResultSet to Progress (INPUT) C–11
temp-table field defaults
ProDataObject 5–11
SQL ResultSet C–2

DataGraph interface
Java SDO features 5–4
See also ProDataGraph class

DataObject interface
Java SDO features 5–4
property meta data 5–63
See also ProDataObject class

Data-relations
ProDataGraph 5–6
ProDataSet 5–3

Date conversions in parameters 4–3
deleteRow( ) method
SDOResultSet 9–21
Deployment Java client applications 1–9
Deployment, application requirements 1–9
detachFromAppObj ( ) method
SDOResultSet 9–16
Digital certificates
managing B–2
Distribution packages for Open Client 1–3

E

enableTracing property 7–16
endLogging( ) method
ProChangeSummary 5–48
Error handling, proxies 8–2
See also Exceptions

Examples
ABL procedures mapped to Java proxy methods 2–3
Java OpenAPI code 11–39
non-persistent procedure 11–39
persistent procedure 11–40

Exceptions
descriptions 8–3
general ABL
class hierarchy 8–3
definition 8–2
output ResultSet 8–2, 8–5

F

findColumn( ) method
SDOResultSet 9–24
first( ) method
SDOResultSet 9–18
Flat model for SQL ResultSets
array fields in temp-tables C–4
behavior C–4
methods compared to array model C–21
ProResultSet standard methods C–14
ProResultSetMetaData standard methods C–19

G

Generating proxies 1–6
get( ) method
ProDataObject 5–36
getBigDecimal( ) method
ProDataObject 5–37
getBImageFlag( ) method
ProDataObjectMetaData 5–55
getBoolean( ) method
ProDataObject 5–37
getBytes( ) method
ProDataObject 5–38
getChangedDataObjects( ) method
ProChangeSummary 5–48
getChanges( ) method
ProChangeSummary 5–49
getChangeSummary( ) method
ProDataGraph 5–31
getChildColumns( ) method
ProDataRelationMetaData 5–61
getChildRows( ) method
ProDataObject 5–44

getChildTable( ) method
ProDataRelationMetaData 5–61

getClass( ) method
Type 5–63

getColumnCount( ) method
SDOResultSetMetaData 9–27

getColumnDisplaySize( ) method
SDOResultSetMetaData 9–27

getColumnJavaTypeName( ) method
ProResultSetMetaData C–20

getColumnLabel( ) method
SDOResultSetMetaData 9–27

getColumnName( ) method
SDOResultSetMetaData 9–27

getColumnProType( ) method
ProResultSetMetaData C–20

getColumnType( ) method
SDOResultSetMetaData 9–27

getColumnTypeName( ) method
SDOResultSetMetaData 9–27

_getConnectionId( ) method 2–6

getDataType( ) method
ProResultSet C–15, C–16

getDefault( ) method
Property 5–64

getExtent( ) method
ProDataObjectMetaData 5–56

getFetchSize( ) method
SDOParameters 9–11
SDOResultSet 9–24

getFieldCount( ) method
ProDataObject 5–45
ProDataObjectMetaData 5–56
ProResultSetMetaData interface C–20

getFieldExtent( ) method
ProResultSetMetaData C–20

getFieldJavaTypeName( ) method
ProResultSetMetaData C–20

getFieldName( ) method
ProDataObjectMetaData 5–56

getFieldProType( ) method
ProResultSetMetaData C–20

getFields( ) method
SDOModificationException 9–32

getFieldType( ) method
ProResultSetMetaData C–20

getGregorianCalendar( ) method
ProDataObject 5–45

getInstanceClass( ) method
Type 5–63

getInt( ) method
ProDataObject 5–38

getList( ) method
ProDataObject 5–39

getLong( ) method
ProDataObject 5–39

getMetaData( ) method
ProDataGraph 5–32
SDOResultSetMetaData 9–25
SQL ResultSet C–8

getName( ) method
Property 5–65

getNoSchemaMarshal( ) method
InputResultSet C–10
ProDataObjectMetaData 5–56

getNumRelations( ) method
ProDataGraphMetaData 5–51

getNumTables( ) method
ProDataGraph 5–33
ProDataGraphMetaData 5–51

getObject( ) method
InputResultSet C–10
ProResultSet C–15, C–16

getOldValues( ) method
ProChangeSummary 5–49

getOutputParameter( ) method
ParamArray 11–37

getParentColumns( ) method
ProDataObjectMetaData 5–56

getParentRow( ) method
ProDataObject 5–45

getParentTable( ) method
ProDataRelationMetaData 5–61

getPrefetchMaxRows( ) method
SDOParameters 9–12

getProChangeSummary( ) method
ProDataGraph 5–33
_getProcReturnString( ) method
common proxy object 2–7
OpenAppObject 11–38
OpenProcObject 11–38

getProcReturnString( ) method
ParamArray 11–38

getProDataGraphName( ) method
ProDataGraph 5–33

getProDataObjects( ) method
ProDataGraph 5–33

getProperties( ) method
Type 5–63

getProperty( ) method
Type 5–64

getProType( ) method
ProDataObjectMetaData 5–57

getQuery( ) method
SDOResultSet 9–25

getRelationMetaData( ) method
ProDataGraphMetaData 5–52

getRelationName( ) method
ProDataRelationMetaData 5–61

_getRequestId( ) method 2–7

getRow( ) method
SDOResultSet 9–18

getRowErrorString( ) method
ProDataObject 5–46

getRowIdentity( ) method
SDOParameters 9–11
SDOResultSet 9–25

getRowErrorString( ) method
ProDataObject 5–46

getTableErrorString( ) method
ProDataObject 5–46

getTableMetaData( ) method
ProDataGraphMetaData 5–52

getTableNames( ) method
ProDataGraph 5–34
ProDataGraphMetaData 5–52

getText( ) method
SDOModificationException 9–32

getType( ) method
ProDataObject 5–40
Property 5–65

getUserOrder( ) method
ProDataObjectMetaData 5–57

H

hasError( ) method
ProDataGraph 5–34

hasRowError( ) method
ProDataObject 5–46

hasTableError( ) method
ProDataObject 5–46

Holder classes 4–4
constructors 4–4
super class definition 4–4
type-specific
array mappings 4–6
definitions 4–7
mappings 4–5
ProDataGraph 5–10

HTTP
Open Client Runtime packages 1–2
HTTPS
Lite Open Client Runtime packages 1–2
Standard
Open Client Runtime packages 1–2
See also Secure Sockets Layer (SSL)

I

IDLE state
connection 3–6
idleConnections property 7–16
idleConnectionTimeout property 7–16
Implementation classes 6–2
  inBatch( ) method
    SDOResultSet 9–23

INPUT parameters 4–2
  Java OpenAPI 11–33
  ProDataGraph 5–17
    adding data 5–22
    initializing meta data 5–19
    preparing 5–18

INPUT SQL ResultSets C–7

INPUT-OUTPUT parameters 4–4
  Java OpenAPI 11–34
  ProDataGraph 5–17
    adding data 5–22
    initializing meta data 5–19
    preparing 5–18

InputResultSet class
  definition C–2
  getObject( ) method C–10
  next( ) method C–9
  using C–9

insertRow( ) method
  SDOResultSet 9–21

Internet Explorer
  permissions for applets 10–2

isAfterLast( ) method
  SDOResultSet 9–18

isAttached( ) method
  SDOResultSet 9–17

isBeforeFirst( ) method
  SDOResultSet 9–18

isCreated( ) method
  ProChangeSummary 5–49

isDeleted( ) method
  ProChangeSummary 5–49

isFirst( ) method
  SDOResultSet 9–18

isInstance( ) method
  Type 5–64

isMany( ) method
  Property 5–65

isNull( ) method
  SDOResultSetMetaData 9–28

isReadOnly( ) method
  SDOResultSetMetaData 9–28

isSet( ) method
  ProDataObject 5–40

_isStreaming( ) method 2–7

isWritable( ) method
  SDOResultSetMetaData 9–28

J

Java SDO. See Java Service Data Objects (Java SDO)

Java Service Data Objects (Java SDO) 5–4
  Data Mediator Services 5–5
  DataGraph interface 5–4
    documentation 5–5
  ProDataGraph extensions 5–6
  property meta data 5–63
  reference implementation 5–5
  See also DataGraph interface

JSR 235
  documentation 5–5
  overview 5–4

L

last( ) method
  SDOResultSet 9–18

List of files in the certificate store B–2

logFileName property 7–16

loggingLevel property 7–17

M

Managing digital certificates B–2

maxConnections property 7–17

Meta data
  Java Service Data Objects (SDO) 5–5
    underlying objects 5–63
  ProDataGraphs 5–6
    accessing output parameters with
      known schema 5–13
    accessing output parameters with
      unknown schema 5–15
    initializing input parameters 5–19
  SmartDataObjects (SDOResultSets) 9–27

SQL ResultSets
  INPUT TABLE-HANDLE C–11
  OUTPUT TABLE and
    TABLE-HANDLE C–18
  temp-table ABL field data types 4–10
Methods
absolute( )  9–17, 9–25
acceptChanges( )  5–31
addDataRelation( )  5–50
addProDataObject( )  5–32
addTable( )  5–51
attachToAppObj( )  9–17
beforeFirst( )  9–18
beginLogging( )  5–48
boolean wasNull( )  9–25
cancelAllRequests( )  2–5
cancelBatch( )  9–23
cancelRowUpdates( )  9–21
class factory  2–4
clear( )  11–8
close( )
   ProResultSet  C–14
   SDOResultSet  9–24
connection  2–3
CreateAO_SubAppObject( )  2–4
CreatePO_ProcObject( )  2–4
createPO( )
   OpenAppObject  11–5
createProDataObject( )  5–32
   _CreateSDOResultSet( )  2–5
deleteRow( )  9–21
detachFromAppObj( )  9–16
endLogging( )  5–48
findColumn( )  9–24
first( )  9–18
get( )  5–36
getBigDecimal( )  5–37
getBImageFlag( )  5–55
getBoolean( )  5–37
getBytes( )  5–38
getChangedDataObjects( )  5–48
getChanges( )  5–49
getChangeSummary( )  5–31
getChildColumns( )  5–61
getChildRows( )  5–44
getProDataObjects( )  5–33
getClassName( )  5–63
getColumnCount( )  9–27
getColumnDisplaySize( )  9–27
getColumnJavaTypeName( )  C–20
getColumnLabel( )  9–27
getColumnName( )  9–27
getColumnProType( )  C–20
getProDataObjectName( )  9–27
   _getConnectionId( )  2–6
dataType( )  C–15, C–16
getDefaultValue( )  5–64
getExtent( )  5–65
getFetchSize( )
   SDOParameters  9–11
SDOResultSet  9–24
getFieldCount( )
   ProDataObject  5–45
   ProDataObjectMetaData  5–56
   ProResultSetMetaData  C–20
getFieldExtent( )  C–20
getFieldJavaTypeName( )  C–20
getFieldName( )
   ProDataObjectMetaData  5–56
   ProResultSetMetaData  C–20
getFieldProType( )  C–20
getFieldIds( )  9–32
getFieldTypeName( )  C–20
getGregorianCalendar( )  5–45
getInstanceClass( )  5–38
getLong( )  5–39
getMetaData( )
   ProDataGraph  5–32
   SDOResultSetMetaData  9–25
   SQL ResultSet  C–8
getName( )  5–65
getNoSchemaMarshal( )  5–56
InputResultSet  C–10
getNumRelations( )  5–51
getNumTables( )
   ProDataGraph  5–33
   ProDataGraphMetaData  5–51
ggetObject( )
   InputResultSet  C–10
   ProResultSet  C–15, C–16
getOldValues( )  5–49
getOutputParameter( )  11–37
getParentColumns( )  5–61
getParentRow( )  5–45
getParentTable( )  5–61
getPrefetchMaxRows( )  9–12
getProChangeSummary( )  5–33
   _getProcReturnString( )
      common proxy object  2–7
      OpenAppObject  11–38
      OpenProcObject  11–38
getProcReturnString( )
   ParamArray  11–38
getProDataGraphName( )  5–33
getProDataObjects( )  5–33
getProperties( )  5–63
getProperty( )  5–64
getProType( )  5–57
getQuery( )  9–25
getRelationMetaData( )  5–52
getRelationName( )  5–61
   _getRequestId( )  2–7
getReturnValue( )  11–38
getRow( )  9–18
getRowErrorString( )  5–46
getRowIdentity( )
   SDOParameters  9–11
   SDOResultSet  9–25
getScrollingMode( )  9–12
getSDOInterface( )  9–26
   _getSSLSubjectName( )  2–8
getStateless( )  9–11
getString( )  5–40
getTable( )  9–32
getTableErrorString( )  5–46
Index

getableMetaData( ) 5–52
getAddress( )
  ProDataObject 5–46
  ProDataObjectMetaData 5–57
  SDOResultSetMetaData 9–28
getTableName( )
  ProDataGraph 5–34
  ProDataGraphMetaData 5–52
getText( ) 9–32
getType( )
  ProDataObject 5–40
  Property 5–65
getUserOrder( ) 5–57
hasError( ) 5–34
hasRowError( ) 5–46
hasTableError( ) 5–46
inBatch( ) 9–23
insertRow( ) 9–21
isAfterLast( ) 9–18
isAttached( ) 9–17
isBeforeFirst( ) 9–18
isCreated( ) 5–49
isDeleted( ) 5–49
isFirst( ) 9–18
isInstance( ) 5–64
isMany( ) 5–65
isNullable( ) 9–28
isReadOnly( ) 9–28
iset( ) 5–40
  _isStreaming( ) 2–7
isWritable( ) 9–28
Java OpenAPI
  data type-specific add parameter 11–8
generic add parameter 11–16
moveToCurrentRow( ) 9–21
moveToInsertRow( ) 9–21
next( )
  InputResultSet C–9
  ProResultSet C–14
  SDOResultSet 9–17
nextMessage( ) 9–32
proxy object 7–4
  common 2–5
  Secure Sockets Layer property
    accessor 7–10
  server property accessor 7–9
  session property accessor 7–7
  thread control property accessor 7–8
  tracing property accessor 7–10
relative( ) 9–18
_release( ) 2–7
releaseConnection( ) 3–5
releaseSDOInterface( ) 9–26
remote ABL 2–3
reOpenQuery( )
  Appserver connections 9–16
  query management 9–26
  row fetch 9–14
  rowDeleted( ) 9–21
  rowInserted( ) 9–21
  rowUpdated( ) 9–21
running on session-free AppObjects 2–9
runProc( )
  OpenAppObject 11–5
  OpenProcObject 11–6
sendBatch( ) 9–16
sendBatchAndReOpen( )
  AppServer connections 9–16
    batch mode 9–23
set( ) 5–41
setBigDecimal( ) 5–41
setBoolean( ) 5–42
setBytes( ) 5–42
setChildTableReferences( ) 5–34
setColumns( ) 5–62
setFetchSize( ) 9–11
setFieldMetaData( )
  ProDataObjectMetaData 5–57
  ProResultSetMetaDataImpl C–13
setGregorianCalendar( ) 5–47
setInt( ) 5–43
setList( ) 5–43
setLong( ) 5–44
setNoSchemaMarshal( ) 5–58
InputResultSet C–10
setPrefetchMaxRows( )
  SDOParameters 9–12
  SDOResultSet stateless mode 9–14
  setRowIdentity( ) 9–10
  setScrollingMode( ) 9–12
  setStateless( ) 9–11
  setString( ) 5–44
  startBatch( ) 9–23
  updateRow( ) 9–21
minConnections property 7–17
moveToCurrentRow( ) method
  SDOResultSet 9–21
moveToInsertRow( ) method
  SDOResultSet 9–21

N

Nescape
  permissions for applets 10–3

Network connections
  permissions for applets 10–2

next( ) method
  InputResultSet C–9
  ProResultSet C–14
  SDOResultSet 9–17

nextMessage( ) method
  SDOModificationException 9–32
noHostVerify property 7–18
noSslSessionReuse property 7–18
nsClientMaxPort property 7–18
<table>
<thead>
<tr>
<th>Property</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsClientMinPort property</td>
<td>7–18</td>
</tr>
<tr>
<td>nsClientPickListExpiration property</td>
<td>7–19</td>
</tr>
<tr>
<td>nsClientPicklistSize property</td>
<td>7–19</td>
</tr>
<tr>
<td>nsClientPortRetry property</td>
<td>7–19</td>
</tr>
<tr>
<td>nsClientPortRetryInterval property</td>
<td>7–19</td>
</tr>
</tbody>
</table>

**O**

Objects and methods. See Methods, Proxy objects and methods

Open Client
- deployment requirements: 1–9
- Java applets: 10–1
- support for applets: 10–1
  See also: Applets
  supported applications: 1–3
  supported protocols: 1–3

Open Client Runtime package
- supported security: 1–4

Open Client Runtime packages
- basic packages: 1–2
  HTTP: 1–2
- HTTPS Lite: 1–2
- HTTPS Standard: 1–2
- list of packages: 1–3
- SSL: 1–2
- supported security: 1–4
- unique class ID: 10–6

Open4GLException class: 8–3

OpenAPI
- AppServer connections: 11–3
- classes summary: 11–2
- parameters: 11–7
  - adding with data type-specific methods: 11–8
  - adding with generic methods: 11–16
- CHARACTER methods: 11–10
- COM-HANDLE methods: 11–11
- creating variables: 11–7
- DATASET and DATASET-HANDLE methods: 11–11
- DATE methods: 11–11
- DATETIME methods: 11–12
- DATETIME-TZ methods: 11–12
- DECIMAL methods: 11–12
- defining ProDataSet data-relations: 11–25
- defining ProDataSet temp-tables: 11–25
- HANDLE methods: 11–13
- INPUT, passing: 11–33
- INPUT-OUTPUT, passing: 11–34
- INTEGER methods: 11–13
- LOGICAL methods: 11–14
- LONGCHAR methods: 11–14
- MEMPTR methods: 11–14
- OUTPUT, passing: 11–35
- OUTPUT, value: 11–37
- parameter array: 11–7
- ProDataSet schema: 11–24
- ProDataSet temp-table schema: 11–28
- RAW methods: 11–15
- RECID methods: 11–15
- ROWID methods: 11–15
- TABLE and TABLE-HANDLE methods: 11–16
- temp-table instance (SQL ResultSet): 11–24
- temp-table schema (SQL ResultSet): 11–22
- temp-table schema and instance (ProDataGraph): 11–31
- UDF return types: 11–19
- returned values: 11–37
- OUTPUT parameters: 11–37
- RETURN-VALUE function from procedures: 11–38
- user-defined function: 11–38
- running
  - application services: 11–4
  - internal procedures and user-defined functions: 11–6
  - non-persistent procedures: 11–5
  - persistent procedures: 11–5
  - user-defined function return types: 11–19

OpenAppObject class
- accessing ABL procedure return values: 11–38
- constructors: 11–3
- parameters. See OpenAPI
  - parameters
  - running
    - application services: 11–4
    - internal procedures and user-defined functions: 11–6
    - non-persistent procedures: 11–5
    - persistent procedures: 11–5

OpenEdge Reference Architecture (OERA)
- 5–1
  - preparing a ProDataGraph parameter: 5–18
  - ProDataSet parameter passing mode: 5–24, 5–25

OpenProcObject class
- accessing ABL procedure return values: 11–38
- creating instances: 11–5
  - method parameters. See OpenAPI
  - parameters
  - running internal procedures and UDFs: 11–6
OUTPUT parameters 4–4
  Java OpenAPI 11–35
  ProDataGraph 5–12
  accessing with a known schema 5–13
  accessing with an unknown schema 5–15

OUTPUT SQL ResultSets C–7
  closing C–8
  order of access C–8

**P**

ParamArray class
  accessing ABL procedure return values 11–38
  constructor 11–7

ParamArrayMode class 11–2
  constants passed to data type-specific methods 11–9
  constants passed to generic methods 11–17

Parameter class 4–10

Parameters
  date and time conversions 4–3
  holder classes 4–4
  See also Holder classes

INPUT mappings 4–2

Java OpenAPI 11–7
  adding with data type-specific methods 11–8
  adding with generic methods 11–16
  INPUT, passing 11–33
  INPUT-OUTPUT, passing 11–34
  OUTPUT, passing 11–35
  See also OpenAPI parameters

OUTPUT and INPUT-OUTPUT array mappings 4–6

OUTPUT and INPUT-OUTPUT mappings 4–5
  passing 4–1

Progress array mappings 4–3
  temp-tables and ProDataSets 4–8
  ABL mechanisms for passing 4–8
  choosing a Java mechanism 4–9
  Java mechanisms for passing 4–9
  See also Temp-table parameters, ProDataSet parameters

Unknown value (?) mappings 4–4

Passing parameters 4–1
  See also Parameters

Permissions for applets
  Internet Explorer 10–2
  Netscape 10–3

PRO_* data type constants C–20

Procertm utility B–2

ProChangeSummary class 5–48
  accessing instances 5–25
  methods
    extended 5–49
    inherited from ChangeSummary 5–48
  tracking changes to a ProDataGraph 5–24

ProcObjects
  class factory method 2–4
  example 2–13
  generation 2–2

ProDataGraph class 5–30
  accessing the meta data (unknown schema) 5–15
  accessing with a known schema 5–13
  adding data 5–22
  change tracking 5–6
  constructors
    using an initial DataGraph 5–30
    using meta data 5–30
  Data Mediator Services requirements 5–5
  DataGraph extensions for ProDataSets 5–6
  data-relations 5–6
  function and foundations 5–3
  holder class for output parameters 5–10
  implementation of DataGraph 5–5
  initializing meta data 5–19
  meta data 5–6
  methods
    extended 5–31
    inherited from DataGraph 5–31
  object model 5–7
  overview 4–9
  passing as a parameter 5–9
  for INPUT or INPUT-OUTPUT 5–17
  for OpenAPI 11–24, 11–28
  for OUTPUT 5–12
  modes 5–10
  preparing 5–9
  supporting OpenEdge classes 5–6
  updating an instance 5–24
  adding a row 5–28
  deleting a row 5–28
  error checking 5–26
  finding a row 5–27
  management cycle 5–24
  See also ProDataSet parameters

ProDataGraphHolder class 5–10

ProDataGraphMetaData class 5–50
  constructor 5–50
  Java OpenAPI 11–25
  defining data-relations, OpenAPI 11–25
  defining temp-tables, OpenAPI 11–25
  methods 5–50
ProDataObject class 5–36
accessing
instances using known temp-table name 5–13
instances using related table rows 5–27
property meta meta 5–63
temp-table meta data 5–15
creating instances 5–28
initializing meta data 5–19
methods
extended 5–44
inherited from DataObject 5–36
temp-table field data type mappings 5–11
array fields 5–12
initial values 5–12
single-valued fields 5–11
Unknown value (?) 5–12
ProDataObjectMetaData class 5–53
constructor 5–53
methods 5–55
ProDataRelationMetaData class 5–59
constructors
Java OpenAPI 11–25
specifying parent and child
ProDataObject key fields 5–60
specifying ProDataObjectMeta Data relations 5–59
methods 5–61
ProDataSet parameters
ABL mechanisms for passing 4–8
accessing 5–1
accessing unknown schema (meta data) 5–15
accessing with a known schema 5–13
data-relation support 5–6
defining data-relations, OpenAPI 11–25
defining temp-tables, OpenAPI 11–25
Java mechanisms for passing 4–9
mapping to Java ProDataGraph 5–9
meta data support 5–6
overview 5–3
passing 4–8
as INPUT or INPUT-OUTPUT 5–17
as OUTPUT 5–12
modes 5–10
preparing to pass 5–9
Java OpenAPI 11–24, 11–28
static and dynamic 4–8
supporting Java classes 5–6
temp-table field data type mappings 5–11
array fields 5–12
initial values 5–12
single-valued fields 5–11
Unknown value (?) 5–12
updating 5–24
See also ProDataGraph class
Programming Open Clients
accessing a SmartDataObject API A–1
Properties
certificateStore 7–15
column 5–4
connectionLifetime 7–15
enableTracing 7–16
idleConnections 7–16
idleConnectionTimeout 7–16
logFileName 7–16
loggingLevel 7–17
maxConnections 7–17
minConnections 7–17
noHostVerify 7–18
noSslSessionReuse 7–18
nsClientMaxPort 7–18
nsClientMinPort 7–18
nsClientPickListExpiration 7–19
nsClientPicklistSize 7–19
nsClientPortRetry 7–19
nsClientPortRetryInterval 7–19
proxy 7–1
command-line access 7–2
Connection class 7–3
list of available 7–6
method access 7–4
modifying Connection object 7–14
RunTimeProperties class 7–3
proxyHost 7–19
proxyPassword 7–20
proxyPort 7–20
proxyUserId 7–21
reference 5–4
requestWaitTimeout 7–21
sessionModel 7–22
waitForBusy 7–22
Property interface 5–64
Java SDO features 5–4
methods 5–64
property types 5–4
ProResultset interface
close( ) method C–14
definition C–2
extended methods C–16
getType( ) method C–15, C–16
getObject( ) method C–15, C–16
next( ) method C–14
standard methods C–14
using C–14
ProResultsetMetaData interface C–18
extension methods C–20
ggetColumnJavaTypeName( ) method C–20
ggetColumnProType( ) method C–20
ggetFieldCount( ) method C–20
ggetFieldExtent( ) method C–20
ggetFieldJavaTypeName( ) method C–20
getFieldName( ) method C–20
ggetFieldProType( ) method C–20
getFieldTypeName() method C–20
standard ResultSetMetaData methods
  C–19

ProResultSetMetaDataImpl class C–12
  adding field descriptions for OpenAPI
  11–23
  constructor C–12
    Java OpenAPI 11–22
  defining an instance for OpenAPI 11–22

ProSQLException class 8–5

Proxy
  exceptions 8–2
  handling errors 8–2
  methods 2–3
    ABL return values 2–4
    class factory 2–4
    common 2–5
    passing parameters. See Parameters
    property accessor 7–4
    running on session-free AppObjects
    2–9
  objects
    AppObject instantiation 3–3
    extending in Java 6–2
    generated by ProxyGen 2–2
  objects and methods 2–1
  properties 7–1
    command-line access 7–2
    Connection class 7–3
    list of available 7–6
    method access 7–4
    modifying Connection object 7–14
  RunTimeProperties class 7–3
  property accessor methods
    Secure Sockets Layer 7–10
    server 7–9
    session 7–7
    thread control 7–8
    tracing 7–10
  requirements
    generation on Windows 1–6, 1–7
    using in applications 1–8
  sample
    AppObject 2–11
    ProcObject 2–13
    SubAppObject 2–12
  proxyHost property 7–19
  proxyPassword property 7–20
  proxyPort property 7–20
  proxyUserId property 7–21

Q

IQUEUED state
  request 2–6

R

Reference properties 5–4

refreshRow() method
  SDOResultSet 9–24

relative() method
  SDOResultSet 9–18

_release() method 2–7

releaseConnection() method 3–5

releaseSDOInterface() method,
  SDOResultSet 9–26

Releasing proxy objects
  closing OUTPUT SQL ResultSets C–8

Remote ABL methods 2–3

reOpenQuery() method
  SDOResultSet
    AppServer connections 9–16
    query management 9–26
    row fetch 9–14

Requesting permissions
  for applets 10–2

requestWaitTimeout property 7–21

ResultSetHolder class
  definition C–2

ResultSetMetaData objects C–8
  INPUT TABLE-HANDLE parameters
    C–11
  OUTPUT TABLE and
    TABLE-HANDLE parameters C–18

Return values, ABL 2–4
  Java OpenAPI 11–37

Returning connection string to a client 3–4

rowDeleted() method
  SDOResultSet 9–21

rowInserted() method
  SDOResultSet 9–21

rowUpdated() method
  SDOResultSet 9–21

RUNNING state
  connection 3–6
    request 2–6

runProc() method
  OpenAppObject 11–5
  OpenProcObject 11–6
RunTimeProperties class
  definition 7–3
  general accessor methods 7–4
  purposed accessor methods 7–6
  relationship to Connection class 7–12

S

Sample Java client applications
  accessing and updating ProDataSets 5–66
  Documentation and Samples directory 1–11
  OpenAPI example code 11–39
    non-persistent procedure 11–39
    persistent procedure 11–40
  passing a temp-table as an SQL ResultSet
    C–23

Schema. See Meta data

Scrolling modes, SDOResultSet 9–13

SDOAppObject class
  connecting to AppServer 9–8
  definition 9–5

SDOModificationException class 9–31
  getFields( ) method 9–32
  getTable( ) method 9–32
  getText( ) method 9–32
  nextMessage( ) method 9–32

SDOParameters class
  definition 9–5
  getFetchSize( ) method 9–11
  getPrefetchMaxRows( ) method 9–12
  getRowIdentity( ) method 9–11
  getScrollingMode( ) method 9–12
  getStateless( ) method 9–11
  passing 9–10
  setFetchSize( ) method 9–11
  setPrefetchMaxRows( ) method 9–12
    SDOResultSet stateless mode 9–14
  setRowIdentity( ) method 9–10
  setScrollingMode( ) method 9–12
  setStateless( ) method 9–11

SDOResultSet class
  absolute( ) method 9–17, 9–25
  attachToAppObj( ) method 9–17
  batch mode 9–22
  beforeFirst( ) method 9–18
  boolean wasNull( ) method 9–25
  cancelBatch( ) method 9–23
  cancelRowUpdates( ) method 9–21
  class factory method 2–5
  close( ) method 9–24
  comparing with JDBC 9–34
  creating instance 9–9
  definition 9–5
  deleteRow( ) method 9–21
  detachFromAppObj( ) method 9–16
  detaching 9–16
    typical scenario 9–17
  findColumn( ) method 9–24
  first( ) method 9–18
  getFetchSize( ) method 9–24
  getQuery( ) method 9–25
  getRow( ) method 9–18
  getRowIdentity( ) method 9–25
  getSDOInterface( ) method 9–26
  getting column values 9–18
  inBatch( ) method 9–23
  insertRow( ) method 9–21
  isAfterLast( ) method 9–18
  isAttached( ) method 9–17
  isBeforeFirst( ) method 9–18
  isFirst( ) method 9–18
  meta data 9–27
  nonstandard methods 9–28
    standard JDBC 2 methods 9–27
    methods and operations 9–16
  miscellaneous management methods 9–24
  moveToCurrentRow( ) method 9–21
  moveToInsertRow( ) method 9–21
  multi-threading 9–30
  navigating rows 9–17
  next( ) method 9–17
  refreshRow( ) method 9–24
  relative( ) method 9–18
  releaseSDOInterface( ) method 9–26
  reOpenQuery( ) method
    AppServer connections 9–16
    query management 9–26
    row fetch 9–14
    rowDeleted( ) method 9–21
    rowInserted( ) method 9–21
    rowUpdated( ) method 9–21
    scrolling modes 9–13
    sendBatch( ) method 9–16
    sendBatchAndReOpen( ) method
      AppServer connections 9–16
      batch mode 9–23
  SmartDataObject management methods 9–25
  startBatch( ) method 9–23
  stateless mode 9–14
  transaction concurrency control 9–19
  transactions 9–19, 9–22
  updateRow( ) method 9–21
  updating 9–19
    columns 9–20
    rows 9–21
    visibility 9–23

SDOResultSetMetaData interface 9–27
  getColumnType( ) method 9–27
  getColumnDisplaySize( ) method 9–27
  getColumnLabel( ) method 9–27
  getColumnName( ) method 9–27
  getColumnType( ) method 9–27
  getColumnTypeName( ) method 9–27
  getColumnTypeName( ) method 9–27
  getMetaData( ) method 9–25
getTableName( ) method 9–28
isNullable( ) method 9–28
isReadOnly( ) method 9–28
isWritable( ) method 9–28

SDOScrollingMode class 9–13
FORWARD_ONLY 9–13
KEEP_ROWS 9–13
PREFETCH 9–13

Secure Sockets Layer (SSL)
certificate store B–2
file format conversion B–5
management B–3
client certificate management B–1
Open Client Runtime packages 1–2
property accessor methods 7–10

Security
Open Client packages 1–4

sendBatch( ) method
SDOResultSet 9–16

sendBatchAndReOpen( ) method
SDOResultSet
AppServer connections 9–16
batch mode 9–23

Session property
accessor methods 7–7

sessionModel property 7–22

set( ) method
ProDataObject 5–41

setBigDecimal( ) method
ProDataObject 5–41

setBoolean( ) method
ProDataObject 5–42

setBytes( ) method
ProDataObject 5–42

setChildTableReferences( ) method
ProDataGraph 5–34

setColumns( ) method
ProDataRelationMetaData 5–62

setFetchSize( ) method
SDOParameters 9–11

setFieldMetaData( ) method
ProDataObjectMetaData 5–57
ProResultSetMetaDataImpl C–13

setGregorianCalendar( ) method
ProDataObject 5–47

setInt( ) method
ProDataObject 5–43

setList( ) method
ProDataObject 5–43

setLong( ) method
ProDataObject 5–44

setNoSchemaMarshal( ) method
InputResultSet C–10
ProDataObjectMetaData 5–58

setPrefetchMaxRows( ) method
SDOParameters 9–12
SDOResultSet stateless mode 9–14

setRowIdentity( ) method
SDOParameters 9–10

setScrollingMode( ) method
SDOParameters 9–12

setStateless( ) method
SDOParameters 9–11

setString( ) method
ProDataObject 5–44

SmartDataObjects (SDOResultSets)
definition 9–2
direct API access A–1
Java access 9–3
methods supported 9–25
objects required 9–5
requirements 9–3
sample application 9–35
tools and documentation 9–5
mapping from ProcObjects A–2
techniques for accessing A–1
See also SDOResultSet class

startBatch( ) method
SDOResultSet 9–23

Stateless mode, SDOResultSet 9–14

Streaming
connection state 3–6
definition C–7
request state 2–6

STREAMING state
connection 3–6
request 2–6

SubAppObjects
class factory method 2–4
definition 2–12
generation 2–2

Supported applications, Open Client 1–3

Supported protocols, Open Client 1–3
Temp-table parameters
ABL field data types 4–10
ABL mechanisms for passing 4–8
choosing a Java mechanism 4–9
field data types and meta data 4–10
Java mechanisms for passing 4–9
passing 4–8
ProDataGraph mapping 4–9
See also Temp-table parameters as ProDataGraphs
SQL ResultSet mapping 4–9
See also Temp-table parameters as SQL ResultSets
static and dynamic 4–8

Temp-table parameters as ProDataGraphs
5–29
default field data type mappings 5–11
array fields 5–12
initial values 5–12
single-valued fields 5–11
Unknown value (?) 5–12
INPUT and INPUT-OUTPUT 5–17
Java OpenAPI
schema definition 11–31
OUTPUT 5–12

Temp-table parameters as SQL ResultSets
C–1
accessing ABL array fields C–4
accessing schemas C–8
application example C–23
call-back sequence C–6
data type conversions supported C–17
default field data type mappings C–2
INPUT
call-back sequence C–6
passing C–9
ResultSet objects C–7
TABLE-HANDLE meta data C–11
INPUT-OUTPUT
call-back sequence C–7
passing C–22
Java OpenAPI
adding 11–24
adding field descriptions 11–23
defining a ProResultSetMetaDataImpl object 11–22
schema definition 11–22
OUTPUT
accessing C–14
call-back sequence C–6
closing C–8
passing C–14
ResultSet objects C–7
TABLE and TABLE-HANDLE meta data C–18
preparing C–2
Progress data type constants C–20
Thread control property
accessor methods 7–8
Threading model
session-free 2–9
Time conversions in parameters 4–3
Tracing property
accessor methods 7–10
Transactions, SDOResultSet 9–19
concurrency control 9–19
extended 9–22
Type interface 5–63
Java SDO features 5–4
methods 5–63
See also ProDataObjectMetaData class

Unique class ID
for Open Client Runtime packages 10–6

Unknown value (?)
arrays 4–4
Java client programming 4–4
mapping to ProDataObject column properties 5–12
updateRow( ) method
SDOResultSet 9–21

Updating
ProDataGraph objects 5–24
adding a row 5–28
deleting a row 5–28
error checking 5–26
finding a row 5–27
management cycle 5–24
SDOResultSet objects 9–19
batch mode 9–22
column values 9–20
rows 9–21
visibility 9–23

Viewing files in the certificate store B–2

waitIfBusy property 7–22