Progress®



OpenEdge® Development:
Java™ Open Clients

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....11
Purpose....11
Audience....11
Organization....12
Using ABL documentation....13
References to ABL compiler and run-time features....13
References to ABL data types....13
Typographical conventions....13

Chapter 1: Configuring and Deploying Java Open Client Applications..15
Java Open Client Runtime package....15
Preparing to generate proxies for a Java client in Windows using ProxyGen or Batch ProxyGen....17
Preparing to generate proxies for a Java client on UNIX using Batch ProxyGen....18
Building an Open Client application that uses a Java proxy....18
Deploying an Open Client application that uses a Java proxy....19

Chapter 2: Proxy Objects and Methods....21
Introduction....21
Proxy objects....22
Package....22
Proxy methods....22
Connection methods....23
Remote ABL methods....23
Class factory methods....24
Common methods....25
Running methods on session-free AppObjects....28
Session-free threading model....28
Sharing of the connection pool between session-free AppObjects....29
Sample proxy....29
Sample AppObject....29
Sample SubAppObject....31
Sample ProcObject....32

Chapter 3: Connecting to an AppServer....33
Connection class....33
Client principal class....34
Attributes of the ClientPrincipal class....35

Methods of the ClientPrincipal class....36
Establishing the connection....38
Returning a user defined string to the client from the AppServer connection procedure....39
Supported AppServer modes....40
Releasing a session-free Connection object....40
Connection states....41

Chapter 4: Passing Parameters....43

ABL data type mappings....43
INPUT parameters....43
Date and time conversions....45
Arrays as parameters....45
INPUT-OUTPUT and OUTPUT parameters....46
Passing temp-tables and ProDataSets....49
Static and dynamic temp-tables and ProDataSets....50
ABL mechanisms for passing temp-tables and ProDataSets....50
Java mechanisms for passing temp-tables and ProDataSets....50
Mapping temp-table field data types and meta data....51

Chapter 5: Accessing ABL ProDataSets....53

Introducing ABL ProDataSets....54
ProDataSets and using ProDataGraphs to access them....54
ProDataSet structure and usage....54
ProDataGraph function and foundations....55
ProDataGraph object model....58
Preparing and passing ProDataSets as ProDataGraph parameters....59
Preparing and managing a ProDataGraph parameter....60
Managing ProDataObject mappings to temp-tables....61
Passing a ProDataGraph as OUTPUT....63
Passing a ProDataGraph as INPUT or INPUT-OUTPUT....68
Updating a ProDataSet....73
Typical cycle for ProDataSet updates....73
Managing a ProDataGraph through the update cycle....74
Checking for errors....75
Finding a row in a ProDataGraph....76
Adding a row to a ProDataGraph....77
Deleting a row from a ProDataGraph....77
Passing temp-tables as ProDataGraph parameters....77
ProDataGraph class....78
Constructors....78
Inherited methods....79
Extended methods....80
ProDataObject class....83

Inherited methods....84
Extended methods....92
ProChangeSummary class....96
Inherited methods....96
Extended method....98
ProDataGraphMetaData class....98
Constructor....98
Methods....99
ProDataObjectMetaData class....101
Constructor....101
Methods....103
ProDataRelationMetaData class....107
Constructors....107
Methods....109
Using Java SDO classes to access Property meta data....111
Type interface....111
Property interface....112

Chapter 6: Extending Proxy Objects....115
Example of extending proxy objects....115

Chapter 7: Accessing Proxy Properties....119
Introducing proxy properties....119
Setting properties on the command line....120
RunTimeProperties and Connection classes....120
RunTimeProperties class....120
Connection class....120
Accessing properties with methods....121
General property accessor methods for the RunTimeProperties class....121
General property accessor methods for Connection objects....122
Available properties....123
Accessing properties using purposed accessor methods....124
Relationship between the RunTimeProperties class and the Connection class....130
Modifying an instantiated Connection object's properties....131
Alphabetical listing of properties....133
PROGRESS.Session.AppServerKeepalive....133
PROGRESS.Session.certificateStore 133
PROGRESS.Session.clientPrincipal 134
PROGRESS.Session.connectionLifetime....134
PROGRESS.Session.connectionTimeout....134
PROGRESS.Session.enableTracing....134
PROGRESS.Session.idleConnectionTimeout....135
PROGRESS.Session.initialConnections....135

PROGRESS.Session.logFileName....135
PROGRESS.Session.loggingLevel....135
PROGRESS.Session.maxConnections....135
PROGRESS.Session.minConnections....136
PROGRESS.Session.noHostVerify....136
PROGRESS.Session.noSslSessionReuse....136
PROGRESS.Session.nsClientMaxPort....136
PROGRESS.Session.nsClientMinPort....137
PROGRESS.Session.nsClientPicklistExpiration....137
PROGRESS.Session.nsClientPicklistSize....137
PROGRESS.Session.nsClientPortRetry....137
PROGRESS.Session.nsClientPortRetryInterval....137
PROGRESS.Session.proxyHost....138
PROGRESS.Session.proxyPassword....138
PROGRESS.Session.proxyPort....138
PROGRESS.Session.proxyUserId....139
PROGRESS.Session.requestWaitTimeout....139
PROGRESS.Session.sessionModel....140
PROGRESS.Session.CompressionEnabled....140
PROGRESS.Session.CompressionLevel....140
PROGRESS.Session.CompressionThreshold....140
PROGRESS.Session.waitIfBusy....141

Chapter 8: Handling Errors....143

Exceptions....143
General ABL exceptions....144
Exception class hierarchy....144
Exception descriptions....144
Output ResultSet exceptions....147

Chapter 9: Using SmartDataObjects from Java Clients....149

What is a SmartDataObject?....150
Accessing remote SmartDataObjects from Java....150
Requirements for accessing SmartDataObjects....150
Example Java access to a SmartDataObject....151
SmartDataObject access tools and documentation....151
Objects required to access SmartDataObjects....152
Developing and deploying a SmartDataObject-aware application....152
Writing a SmartDataObject-aware Java client....153
Connecting to an AppServer using an SDOAppObject....153
Creating an SDOResultSet object....154
The _createSDOResultSet() method....154
Setting and getting values on the SDOParameters object....156

Understanding SDOResultSet scrolling modes....158
Understanding SDOResultSet stateless mode....159
Working with SDOResultSet objects....160
Detaching SDOResultSet objects [extension]....161
Navigating SDOResultSet rows....162
Getting SDOResultSet column values....163
Updating SDOResultSet objects....164
Miscellaneous management methods....169
Meta data methods....171
Multi-threading....173
Errors and exceptions....173
SDOResultSet exceptions....174
Typical errors by method category....175
Choosing JDBC ResultSets or SDOResultSets to access OpenEdge data....176

Chapter 10: Using the Open Client Java OpenAPI to Directly Access the AppServer....177

Overview of Java OpenAPI classes....177
Connecting to the AppServer....178
Running procedures and user-defined functions....179
Running a non-persistent procedure on an OpenAppObject....180
Running a persistent procedure (OpenProcObject) on an OpenAppObject....180
Running a single-run or singleton procedure (OpenProcObject) on an OpenAppObject....181
Running an internal procedure or user-defined function on an OpenProcObject....182
Setting up parameters....182
Creating variables for parameters....183
Creating a parameter array....183
Adding parameters using data type-specific methods....184
Adding parameters using a generic method....191
Defining the return type for a user-defined function....194
Arrays as return values....195
Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet....197
Defining the schema for a ProDataSet parameter....199
Defining the schema for temp-tables in a ProDataSet....202
Defining the schema for a temp-table parameter mapped to a ProDataGraph....205
Passing parameters....206
Passing INPUT parameters....206
Passing INPUT-OUTPUT parameters....207
Passing OUTPUT parameters....208
Handling returned values....209
Getting OUTPUT parameter values....209
Accessing RETURN-VALUE function output from procedures....209
Accessing user-defined function return values....210
Sample Java OpenAPI code....210

Non-persistent procedure example....211
Persistent procedure example....212
Single-run/singleton procedure example....213

Appendix A: Accessing a SmartDataObject API Directly....215

Introduction....215
Directly accessing the SmartDataObject API in a proxy....216
Defining a ProcObject that maps a SmartDataObject....216

Appendix B: Java Open Client Certificate Management Utility....217

Overview....217
Managing certificate store files....218
Converting digital certificates....219

Appendix C: Passing Temp-tables as SQL ResultSet Parameters....221

Introduction....221
Preparing and managing temp-table parameters....222
Data type mapping for temp-table fields....222
Accessing temp-table array fields....224
SQL ResultSet streaming and call-back sequence....225
INPUT ResultSet objects....227
OUTPUT ResultSet objects....227
Result set schemas....227
Passing a TABLE or TABLE-HANDLE as an INPUT parameter....228
Instantiating the INPUT parameter....228
Using com.progress.open4gl.InputResultSet....228
Providing metadata for INPUT TABLE-HANDLE parameters....230
Passing a TABLE or TABLE-HANDLE as an OUTPUT parameter....232
Accessing TABLE or TABLE-HANDLE parameters as OUTPUT....233
Temp-table field data type conversion....235
Getting meta data for OUTPUT TABLE and TABLE-HANDLE parameters....237
Passing a TABLE or TABLE-HANDLE as an INPUT-OUTPUT parameter....241
Example Java client passing an SQL ResultSet parameter....241

Index....245

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using ABL documentation
- Typographical conventions

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

Configuring and Deploying Java Open Client Applications on page 15

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

Proxy Objects and Methods on page 21

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

Connecting to an AppServer on page 33

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

Passing Parameters on page 43

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

Accessing ABL ProDataSets on page 53

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

Extending Proxy Objects on page 115

Describes how to extend the AppObjects, SubAppObjects, and ProcObjects created by ProxyGen for Java applications.

Accessing Proxy Properties on page 119

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

Handling Errors on page 143

Describes error handling in Java Open Client applications.

Using SmartDataObjects from Java Clients on page 149

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.

Using the Open Client Java OpenAPI to Directly Access the AppServer on page 177

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

Accessing a SmartDataObject API Directly on page 215

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

Java Open Client Certificate Management Utility on page 217

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.

Passing Temp-tables as SQL ResultSet Parameters on page 221

Describes how to pass and manage temp-tables as SQL `ResultSet` objects.

Using ABL documentation

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 updates go to https://docs.progress.com.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the *ABL Virtual Machine (AVM)*. When the 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 the 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, built-in 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 *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` 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 built-in 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 documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	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.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

1

Configuring and Deploying Java Open Client Applications

This chapter contains the configuration tasks necessary for Java™ Open Client development and deployment.

For details, see the following topics:

- Java 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

Java Open Client Runtime package

If your application is a Java servlet or a Java application, use `o4glrt.jar` as the Open Client runtime package. This package supports the following protocols:

- AppServer or AppServerDC
- AppServerS or AppServerDCS
- HTTP
- HTTPS

These protocols are supported over the intranet and internet. The `o4glrt.jar` Open Client runtime package supports user-implemented authentication to a web server and proxy server.

The Java Open Client runtime package `o4glrt.jar` supports most SSL-enabled web servers. The SSL configuration for the individual web server and the digital certificate used to authenticate the identity of the web server determines whether you can use this package.

This package supports user-implemented authentication to a web server, data encryption, and Proxy server.

Note: 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 servers.

If you are using Progress Developer Studio for OpenEdge, you will also need to copy the following third-party Apache Tuscany 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
      oeauth-<version>.jar
OpenEdge-install-directory/java/ext/
      common-<version>.jar
      commons-codec-<version>.jar
      commons-logging-<version>.jar
      ecore-<version>.jar
      ecore-change-<version>.jar
      ecore-xmi-<version>.jar
      httpclient-<version>.jar
      httpasyncclient-<version>.jar
      httpcore-<version>.jar
      httpcore-nio-<version>.jar
      tuscany-sdo-api-r2.1-<version>.jar
      tuscany-sdo-impl-<version>.jar
      tuscany-sdo-lib-<version>.jar
      tuscany-sdo-tools-<version>.jar
      xsd-<version>.jar
```

The following table lists the security supported by various Open Client Runtime packages.

Table 1: Security supported by package

Package	Supported security
Basic	User-implemented authentication using a userid and password

SSL	• Authentication that includes:
	  • User-implemented authentication using a userid and password
	  • SSL V2, SSL V3, and TLS V1, Digital Signatures: 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

Package	Supported security

HTTP	• Authentication that includes:
	  • User-implemented authentication using a userid and password
	  • Basic authentication to the AIA web server
	  • Basic Proxy server authentication
	• Proxy servers

HTTPS	• Authentication that includes:
	  • User-implemented authentication using a userid and password
	  • Basic Proxy server authentication
	  • HTTP basic authentication to the AIA web server
	  • HTTPS using SSL V2, SSL V3, and TLS V1, Digital Signatures: 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

These are the system configuration steps you must perform before you can generate proxies with ProxyGen on Windows platforms:

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` in the Java **Client Details** group of the ProxyGen **Generate Proxies** dialog box. For more information, see the section about the Java client details 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.

   An alternative is to 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

These are the system configuration steps you must perform before you can generate proxies with Batch ProxyGen on UNIX platforms:

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 back slashes (`\`) 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

Before you can build an Open Client application that uses a Java proxy, your system must have the OpenEdge Toolkit installed and it must have a properly configured Java SDK.

To build a client application that uses a Java proxy:

Note: ProxyGen provides the proxy `.java` files only for documentation. Do not attempt to modify and rebuild your proxy from these files.

---

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 for generation of the proxy in the Java **Client Details** group of the ProxyGen **Generate Proxies** dialog box. 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 Java Open Client Runtime package on page 15.

3. Update the Java `CLASSPATH` setting for the client application's developmentenvironment to include the following:

   a) The directory containing the proxy (`.class` files) or the `zip` or `jar` file from Step 2 on page 19. For class files, this is the directory above the package hierarchy.

   b) The Open Client Runtime package you copied in Step 2 on page 19.

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:

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 for generation of the proxy in the Java **Client Details** group of the ProxyGen **Generate Proxies** dialog box. 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 Building an Open Client application that uses a Java proxy on page 18.

d) Copy the following third-party Apache Tuscany jar files to the same directory where you copied the Java Open Client Runtime package in Step 2.c on page 20, above. These files reside in the specified OpenEdge installation directory, as shown:

```
OpenEdge-install-directory/java/ext/
      common-2.2.3.jar
      ecore-2.2.3.jar
      ecore-change-2.2.3.jar
      ecore-xmi-2.2.3.jar
      tuscany-sdo-api-r2.1-1.1.1.jar
      tuscany-sdo-impl-1.1.1.jar
      tuscany-sdo-lib-1.1.1.jar
      tuscany-sdo-tools-1.1.1.jar
      xsd-2.2.3.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 . For class files, this is the directory above the package hierarchy.

   c) The Open Client Runtime package and third-party jar files you copied in Step 2 on page 19.

   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 Java Open Client Certificate Management Utility on page 217.

5. Run your Java Open Client application as designed.

2

Proxy Objects and Methods

This chapter describes the Java proxy classes and methods generated by ProxyGen.

For details, see the following topics:

- Introduction
- Proxy objects
- Proxy methods
- Running methods on session-free AppObjects
- Sample proxy

Introduction

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*.

Proxy objects

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

For each . . .	ProxyGen generates . . .	Example
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 in the Java **Client Details** group of the ProxyGen **Generate Proxies** dialog box. You also can access remote OpenEdge SmartDataObjects from a Java client with or without a ProxyGen-generated proxy. For more information, see Using SmartDataObjects from Java Clients on page 149.

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 in the general settings of the ProxyGen **Generate Proxies** dialog box.

Package

All proxy classes may be created within a package specified in the Java **Client Details** group of the ProxyGen **Generate Proxies** dialog box. The class files are placed in a directory structure defined by the package.

Proxy methods

This section describes the following proxy methods:

- Connection methods on page 23
- Remote ABL methods on page 23
- Class factory methods on page 24
- Common methods on page 25

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 Connecting to an AppServer on page 33.

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:

```
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:

```
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 Passing Parameters on page 43.

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:

```
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 Common methods on page 25.

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

```
public SubAppObject createAO_SubAppObject()
```

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

```
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:

Syntaxs

```
public ProcObject createPO_ProcObject()
```

For example, a procedure `AccountInfo.p` added in ProxyGen as persistent, single-run, or singleton generates this Java method:

```
public AccountInfo createPO_AccountInfo()
```

Note that `AccountInfo.p` cannot be run as single-run or singleton if it has parameters in its main block, and ProxyGen will not allow you to designate a procedure as single-run or singleton if that procedure has parameters in its main block.

For more information on persistent, single-run, and singleton procedures, see *OpenEdge Application Server: Developing AppServer Applications*.

SDOResultSet

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

Syntax

```
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

```
public void _cancelAllRequests()
```

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

This method throws a `com.progress.open4gl.Open4GLException`.

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 Passing Temp-tables as SQL ResultSet Parameters on page 221.

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

```
public String _getConnectionId()
```

The `_getConnectionId()` method throws a `com.progress.open4gl.Open4GLException` if the Application Service is not connected.`_getConnectionId()`

The method is most often used to identify entries in AppServer log files.

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

```
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 Passing Temp-tables as SQL ResultSet Parameters on page 221.

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

```
public String _getProcReturnString()
```

Note: The method returns the current value of the AppServer `RETURN-VALUE` function. For more information on handling return values, see Handling return values on page 23.

Release

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

Syntax

```
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. The connection to the AppServer is automatically released for single-run and singleton procedures after their internal procedures or user-defined functions are completed, and in the case of a single-run procedure, the procedure itself is also automatically deleted form the AppServer. However, the `Dispose( )` method does clean up client-side resources for single-run and singleton procedures.

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

```
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()
```

Procedure type

The `_getProcedureType()` method identifies the type of procedure that a ProcObject encapsulates as persistent, single-run, or singleton. The method returns a Java enum type that can hold a value of `PERSISTENT`, `SINGLE_RUN`, or `SINGLETON`:

Syntax

```
public ProcedureType _getProcedureType()
```

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.

Unlike persistent procedures, single-run and singleton procedures do not reserve connections. In these cases, the connection is released automatically when the internal procedure or user-defined function is complete. However, the `_release()` method does clean up client-side resources.

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 Common methods on page 25.
- 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 ESD 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:

```
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:

```
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:

```
public AccountInfo createPO_AccountInfo(int accountNum)
   throws Open4GLException, RunTime4GLException, SystemErrorException
```

If `AccountInfo.p` were to be run as a single-run or singleton procedure, the parameter in its main block would have to be removed and passed to its internal procedures or user-defined functions directly. The class factory method would then look like this:

```
public AccountInfo createPO_AccountInfo()
   throws Open4GLException, RunTime4GLException, SystemErrorException
```

See theSample ProcObject on page 32 for an example of how the methods for the internal procedures and user-defined functions of might look if `AccountInfo.p` were called as a single-run or singleton procedure.

- Built-in class factory methods that run and instantiate a SmartDataObject on the AppServer. For example:

```
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:

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

public void Remove(int accountNum)
```

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

```
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:

```
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 Sample AppObject on page 29.
- A remote ABL method to run the non-persistent procedure `SetStatus` (defined by `SetStatus.p`) on the AppServer. For example:

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

- Common methods in all Java proxy objects, excluding the _getProcedureType() method.

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:

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

public void setDirectDeposit(ResultSet ddData)

public void getDirectDeposit(ResultSetHolder ddData)
```

  If `AccountInfo.p` were to be run as single-run or singleton, then its input parameter *`accountNum`* would have to removed from the main block and instead be passed directly to its internal procedures. In that case, the internal procedure methods might look like the following:

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

public void setDirectDeposit(int accountNum, ResultSet ddData)

public void getDirectDeposit(int accountNum, ResultSetHolder ddData)
```

- Common methods in all Java proxy objects, including the _getProcedureType() method.

3

Connecting to an AppServer

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

For details, see the following topics:

- Connection class
- Client principal class
- Establishing the connection
- Connection states

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

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

Note: The combined length of the `url`, `userid`, `password`, and `AppServer-info` fields must not exceed 30,000 non-Unicode characters when connecting to an AppServer.

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

```
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 Name** setting specified in the general settings of the ProxyGen **Generate Proxies** dialog box.

Before you establish a connection, you can set properties for the `Connection` object. See Accessing Proxy Properties on page 119 for information on setting and updating `Connection` object properties.

In particular, the `Connection` class has a `clientPrincipal` property, which associates a `ClientPrincipal` object with a `Connection` object. A `ClientPrincipal` object acts as a token for passing security information, like user credentials, between various parts of an application. See Client principal class on page 34 for more information about `ClientPrincipal` objects.

Client principal class

OpenEdge provides a *client principal class*, `com.progress.open4gl.javaproxy.ClientPrincipal`. A `ClientPrincipal` object provides a means to store and transmit user credentials. The `ClientPrincipal` object supports the following use cases:

- Creating an unsealed `ClientPrincipal` object containing a user's credentials and sending this object to an AppServer for a direct login.
- Creating a sealed `ClientPrincipal` object (verified by an external authentication system) and sending this object to an AppServer for a single sign-on.

The Java client principal class mirrors the implementation of the client principal class in ABL. For more detailed information on the ABL implementation, see *OpenEdge Getting Started: Identity Management*.

The following constructor specifies the qualified user ID (composed of the user ID and domain), session ID, expiration date, and pass phrase:

Syntax

```
public ClientPrincipal(String qualifiedUserID, String sessionID,
                       GregorianCalendar expirationDateTime,
                       String primaryPassphrase)
```

An alternate version of the constructor can initialize the object by passing a byte array that contains a serialized `ClientPrincipal` object:

Syntax

```
public ClientPrincipal(Byte[] serializedCP)
```

Calling the constructor with no parameters creates a `ClientPrincipal` object with all String-valued attributes set to null, all long integer-valued attributes set to 0, and the `loginState` attribute set to "`INITIAL`".

Attributes of the ClientPrincipal class

All attributes of the `ClientPrincipal` class are private. The following table lists the accessor methods of the `ClientPrincipal` class attributes.

Note: In general, attributes may only be updated on a `ClientPrincipal` object with a `loginState` of "`INITIAL`". Attempts to update attributes once the `ClientPrincipal` object is sealed, expired, failed, or logged out will throw an exception.

Table 2: Accessor methods of ClientPrincipal attributes

Accessor methods	Description of attribute
`getAuditEventContext()`	The application-defined audit context string
`getClientTTY()` `setClientTTY(String clientTTY)`	The value of this attribute is defined by the application.
`getClientWorkstation()` `setClientWorkstation(String clientWS)`	The value of this attribute is defined by the application.
`getDomainDescription()` `setDomainDescription(String domainDescr)`	A description of the domain that authenticated the user represented by the `ClientPrincipal` object
`getDomainName()` `setDomainName(String domainName)`	The name of the security domain to which the user represented by the `ClientPrincipal` object belongs
`getDomainType()` `setDomainType(String domainType)`	The name of the authentication system configured for the security domain to which the user represented by the `ClientPrincipal` object belongs
`getLoginExpirationTimestamp()` `setLoginExpirationTimestamp(long timeStamp)`	The timestamp of when the `ClientPrincipal` object will expire, expressed as the number of seconds since midnight, January 1, 1970
`getLoginHost()` `setLoginHost(String loginHost)`	The value of this attribute is defined by the application.

Accessor methods	Description of attribute
`getLoginState()`	A string representing the current state of the `ClientPrincipal` object. This field may contain a value of "`INITIAL`", "`LOGIN`", "`LOGOUT`", "`EXPIRED`", or "`FAILED`". This value is set as a result of calling the `seal()`, `validateSeal()`, or `logout()` methods.
`setPrimaryPassphrase(String` *`passphrase`*`)`	The password required to authenticate the user identity represented by the `ClientPrincipal` object
`getQualifiedUserID()` `setQualifiedUserID(String` *`qualifiedUserID`*`)`	A string containing the `userID` and `domainName` attributes separated by the "`@`" delimiter.
`getRoles()` `setRoles(String` *`domainRoles`*`)`	A comma-separated list of domain roles for the user represented by the `ClientPrincipal` object
`getSealTimestamp(long` *`timeStamp`*`)`	The timestamp of when the `ClientPrincipal` object was sealed, expressed as the number of seconds since midnight, January 1, 1970. This value is set as a result of calling the `seal()` method.
`getSessionID()` `setSessionID(String` *`sessionID`*`)`	The user login session ID for the user represented by the `ClientPrincipal` object
`getStateDetail()`	A string that provides detailed information regarding the state of the `ClientPrincipal` object
`getUserID()` `setUserID(String` *`userID`*`)`	The unqualified name of the user represented by the `ClientPrincipal` object. The value of this attribute will also be set automatically if the `setQualifiedUserID()` method is called.

Methods of the ClientPrincipal class

The following table lists the methods of the of the `ClientPrincipal` class.

Table 3: Methods for the ClientPrincipal class

Method	Description
`exportPrincipal()`	Serializes the `ClientPrincipal` object into a byte array. A serialized `ClientPrincipal` object can be deserialized using the `importPrincipal()` method. If errors occur during the serialization process, an exception will be thrown.

Method	Description
`getProperty(String` *`propertyName`*`)`	Returns a string containing the value of the specified user-defined property stored in the `ClientPrincipal` object. If the specified property is not found, null will be returned.
`importPrincipal(Byte[]` *`serializedCP`*`)`	A static method that deserializes a byte array into a `ClientPrincipal` object. If errors occur during the deserialization process (e.g., an improperly formatted byte array), an exception will be thrown.
`seal(String` *`domainAccessCode`*`)`	When called by a user-defined authentication system, this method asserts the authenticity of the user represented by the `ClientPrincipal` object. If successful, the method returns `true`. Calling this method is the beginning of the user's login session, and as a result, the `loginState` attribute is set to, and the `sealTimestamp` attribute is also set. The `ClientPrincipal` object is cryptographically sealed using the specified domain access code. Once sealed, any attempt to set attributes will throw an exception.
`seal(OEDomainRegistry` *`domainRegistry`*`)`	When called by a user-defined authentication system, this method asserts the authenticity of the user represented by the `ClientPrincipal` object. If successful, the method returns `true`. Calling this method is the beginning of the user's login session, and as a result, the `loginState` attribute is set to, and the `sealTimestamp` attribute is also set. The `ClientPrincipal` object is cryptographically sealed using the domain access code defined in the `OEDomainRegistry` object. Once sealed, any attempt to set attributes will throw an exception.
`setProperty(String` *`propertyName`*`,` `String` *`propertyValue`*`)`	Sets the value of the specified property in the unsealed `ClientPrincipal` object. If successful, the method returns `true`.

Method	Description
`validateSeal(String domainAccessCode)`	Validates that the the `ClientPrincipal` object was sealed with the specified domain access code. If successful, the method returns `true`. The `loginExpirationTimeout` attribute is also checked, and if the `ClientPrincipal` object has expired, the `loginState` attribute is set to "`EXPIRED`".
`validateSeal(OEDomainRegistry domainRegistry)`	Validates that the the `ClientPrincipal` object was sealed using the domain access code defined in the `OEDomainRegistry` object. If successful, the method returns `true`. The `loginExpirationTimeout` attribute is also checked, and if the `ClientPrincipal` object has expired, the `loginState` attribute is set to "`EXPIRED`".

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`*):

```
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 Connection class on page 33.

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:

```
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:

```
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 Name** setting specified in the general settings of the ProxyGen **Generate Proxies** dialog box.

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

```
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 Name** setting specified in the general settings of the ProxyGen **Generate Proxies** dialog box.

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

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:

```
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.

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 Accessing Proxy Properties on page 119.

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.

4

Passing Parameters

This chapter provides details about data type mappings in various parameter passing scenarios.

For details, see the following topics:

- 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

The following table shows the ABL-to-Java data type mapping for `INPUT` parameters.

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

ABL data type	Java proxy data type
`CHARACTER`	`java.lang.String`

ABL data type	Java proxy data type
COM-HANDLE	com.progress.open4gl.COMHandle
DATASET DATASET-HANDLE[1]	com.progress.open4gl.ProDataGraph
DATE DATETIME DATETIME-TZ	java.util.GregorianCalendar
DECIMAL	java.math.BigDecimal
HANDLE	com.progress.open4gl.Handle
INT64	long
INTEGER	int
LOGICAL	boolean
LONGCHAR	java.lang.String
MEMPTR	com.progress.open4gl.Memptr
RAW	byte[ ]
RECID	long
ROWID	com.progress.open4gl.Rowid
TABLE TABLE-HANDLE[1]	com.progress.open4gl.ProDataGraph (when mapped as a Java SDO DataGraph)
TABLE TABLE-HANDLE[1]	java.sql.ResultSet (when mapped as an SQL ResultSet)[1]
WIDGET-HANDLE	com.progress.open4gl.Handle

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.

[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 Passing temp-tables and ProDataSets on page 49.

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 the following table.

Table 5: GeorgianCalendar object processing

Progress data type	Process
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 one of the JAVA proxy data types listed in INPUT parameters on page 43. 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 the following example.

Example: OUTPUT and INPUT-OUTPUT holder class

```
// 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

```
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).

The following table shows the ABL-to-Java mapping of INPUT-OUTPUT and OUTPUT parameters to specific holder classes.

Table 6: ABL-to-Java data type mapping for INPUT-OUTPUT and OUTPUT parameters

ABL data type	Java proxy data type
CHARACTER	com.progress.open4gl.StringHolder
COM-HANDLE	com.progress.open4gl.COMHandleHolder
DATASET DATASET-HANDLE[2]	com.progress.open4gl.ProDataGraphHolder
DATE DATETIME DATETIME-TZ	com.progress.open4gl.DateHolder
DECIMAL	com.progress.open4gl.BigDecimalHolder
INT64	com.progress.open4gl.LongHolder
INTEGER	com.progress.open4gl.IntHolder
LOGICAL	com.progress.open4gl.BooleanHolder
LONGCHAR	com.progress.open4gl.StringHolder
MEMPTR	com.progress.open4gl.MemptrHolder
RAW	com.progress.open4gl.ByteArrayHolder

[2] 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 Passing temp-tables and ProDataSets on page 49.

ABL data type	Java proxy data type
RECID	com.progress.open4gl.LongHolder
ROWID	com.progress.open4gl.RowidHolder
TABLE TABLE-HANDLE	com.progress.open4gl.ProDataGraphHolder (when mapped as a Java SDO DataGraph)[2]
TABLE TABLE-HANDLE	com.progress.open4gl.ResultSetHolder (when mapped as an SQL ResultSet[2], input holds java.sql.ResultSet, output holds com.progress.open4gl.ProResultSet)
WIDGET-HANDLE	com.progress.open4gl.HandleHolder

The following table shows the ABL array-to-Java mapping of INPUT-OUTPUT and OUTPUT array parameters to specific holder classes.

Table 7: ABL with array-to-Java data-type mapping for INPUT-OUTPUT and OUTPUT array parameters

ABL array option	Java proxy data type
CHARACTER	com.progress.open4gl.StringArrayHolder
COM-HANDLE	com.progress.open4gl.COMHandleArrayHolder
DATE DATETIME DATETIME-TZ	com.progress.open4gl.DateArrayHolder
DECIMAL	com.progress.open4gl.BigDecimalArrayHolder
INT64	com.progress.open4gl.LongArrayHolder
INTEGER	com.progress.open4gl.IntArrayHolder
LOGICAL	com.progress.open4gl.BooleanArrayHolder
LONGCHAR	com.progress.open4gl.StringArrayHolder
MEMPTR	com.progress.open4gl.MemptrArrayHolder
RAW	com.progress.open4gl.ByteArrayArrayHolder
RECID	com.progress.open4gl.LongArrayHolder
ROWID	com.progress.open4gl.RowidArrayHolder
WIDGET-HANDLE	com.progress.open4gl.HandleArrayHolder

Holder class definitions

The holder class definitions for each data type follow the form shown in the following example.

Example: Holder class definitions

```
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 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 Accessing ABL ProDataSets on page 53.

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 Passing Temp-tables as SQL ResultSet Parameters on page 221.

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 Accessing ABL ProDataSets on page 53. For the list of Java data types available to map temp-table fields using an SQL `ResultSet`, see Passing Temp-tables as SQL ResultSet Parameters on page 221.

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 the following table. Thus, the sections of this manual that describe how to create temp-table field metadata refer to the class constants listed here.

Table 8: Parameter class constants for temp-table field metadata

Progress temp-table field data type	com.progress.open4gl.Parameterclass constant
BLOB	PRO_BLOB
CHARACTER	PRO_CHARACTER
CLOB	PRO_CLOB
COM-HANDLE	PRO_COMHANDLE
DATE	PRO_DATE
DATETIME	PRO_DATETIME
DATETIME-TZ	PRO_DATETIMETZ
DECIMAL	PRO_DECIMAL
INT64	PRO_LONG
INTEGER	PRO_INTEGER
LOGICAL	PRO_LOGICAL
RAW	PRO_RAW
RECID	PRO_RECID
ROWID	PRO_ROWID
WIDGET-HANDLE	PRO_WIDGETHANDLE

5

Accessing ABL ProDataSets

This chapter explains how to pass ProDataSets as application service parameters.

For details, see the following topics:

- Introducing ABL ProDataSets
- 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

Introducing 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

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 Passing a ProDataGraph as INPUT or INPUT-OUTPUT on page 68.

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 Passing temp-tables as ProDataGraph parameters on page 77.

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 Using Java SDO classes to access Property meta data on page 111.)

A `DataGraph` encapsulates a tree of `DataObject` instances starting with a single root `DataObject`. 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 on page 58.)

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, ProDataGraphMetaData 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.

The following table 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 ProDataGraph object model on page 58.

Table 9: OpenEdge classes that extend Java SDO

OpenEdge class	Basis	For more information, see . . .
ProDataGraph	Extends DataGraph to map ProDataSets	ProDataGraph class on page 78

OpenEdge class	Basis	For more information, see . . .
`ProDataObject`	Extends `DataObject` to support `java.util.GregorianCalendar` and provide relational navigation among tables	ProDataObject class on page 83
`ProChangeSummary`	Extends `ChangeSummary` to return all `ProDataObject` changes in a changes-only `ProDataGraph`	ProChangeSummary class on page 96
`ProDataGraphMetaData`	OpenEdge-based class to more easily manage `ProDataGraph` schema information	ProDataGraphMetaData class on page 98
`ProDataObjectMetaData`	OpenEdge-based class to more easily manage `ProDataObject` schema information	ProDataObjectMetaData class on page 101
`ProDataRelationMetaData`	OpenEdge-based class to describe `ProDataGraph` data-relations	ProDataRelationMetaData class on page 107

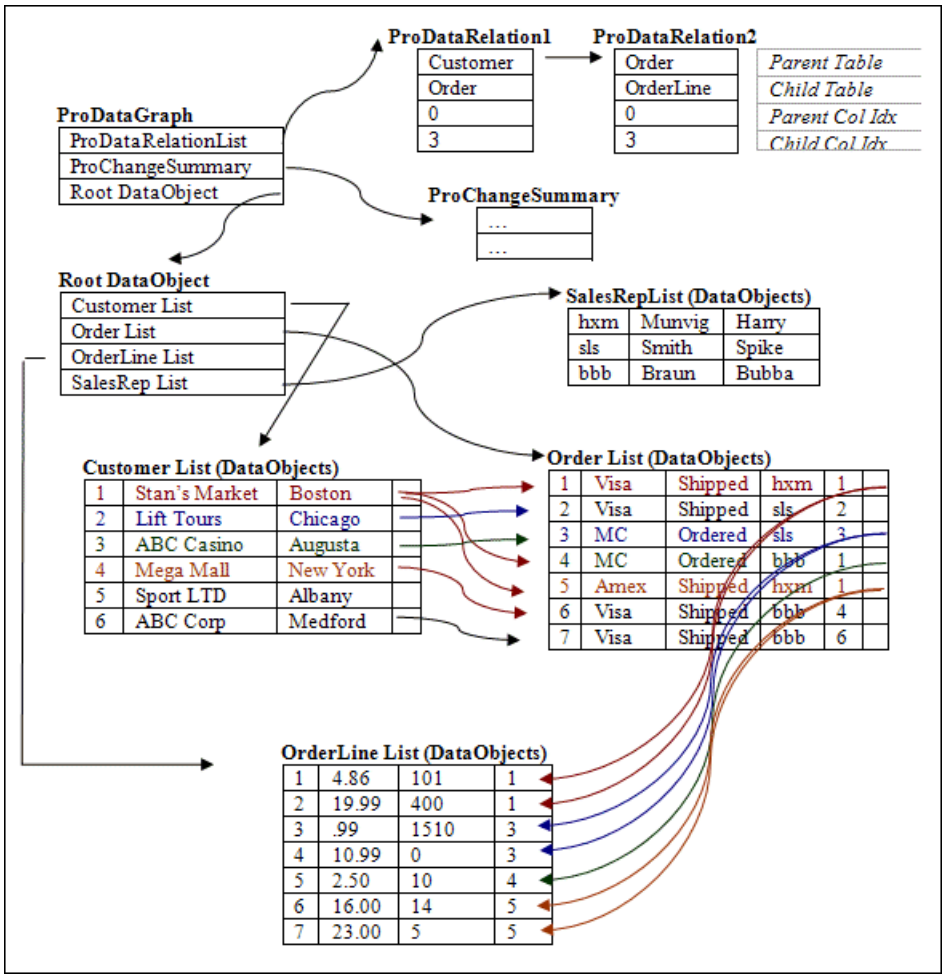
ProDataGraph object model

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`

The following figure 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`.

Figure 1: ProDataGraph object model



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

The following table shows how the components of an ABL ProDataSet map to a ProDataGraph and its supporting objects.

Table 10: ABL ProDataSet mappings to Java ProDataGraph

ABL component	Java object
ProDataSet (static or dynamic)	`ProDataGraph` containing a schema-matching `ProDataGraphMetaData` object
Temp-table (static or dynamic)	`ProDataObject` list supported by a schema-matching `ProDataObjectMetaData` object
Buffer (temp-table row)	Any corresponding `ProDataObject`
Before-table	The original property values derived from a `ProDataObject` list in the `ProDataGraph` returned by the `ProChangeSummary.getChanges()`[3]
Field	`Property` object in a `ProDataObject`
Data-relation	`ProDataRelationMetaData`
Any primary or unique index	`ProDataRelationMetaData`
Any non-unique index	Not mapped

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 ProDataGraphHolder class on page 61). 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 Passing a ProDataGraph as OUTPUT on page 63.
- **`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

[3] 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`*).

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 Passing a ProDataGraph as INPUT or INPUT-OUTPUT on page 68.

- **`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 on page 61). 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 Passing a ProDataGraph as INPUT or INPUT-OUTPUT on page 68.

ProDataGraphHolder class

The following ProDataGraphHolder class definition shows the constructor and method signatures for the `ProDataGraphHolder` class.

Example: ProDataGraphHolder class definition

```
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 Passing Parameters on page 43.

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 Passing Temp-tables as SQL ResultSet Parameters on page 221.

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 Passing temp-tables as ProDataGraph parameters on page 77.)

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 Java SDO Foundations on page 55), 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. The following table shows the `ProDataObject` mappings to temp-table fields supported by ABL.

Table 11: Java ProDataObject mappings to temp-table fields

ABL temp-table field data type	Java ProDataObject Property data type
BLOB	byte[]
CHARACTER	java.lang.String
CLOB	java.loan.String
COM-HANDLE	long
DATE	java.util.GregorianCalendar
DATETIME	java.util.GregorianCalendar
DATETIME-TZ	java.util.GregorianCalendar
DECIMAL	java.math.BigDecimal
INT64	long
INTEGER	int
LOGICAL	boolean
RAW	byte[]
RECID	long
ROWID	byte[]
WIDGET-HANDLE	long

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 ProDataObject class on page 83.

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 of the Java data types listed in Mapping single-valued fields on page 62 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 ProDataObject class on page 83.

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 ProDataObject class on page 83. 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.

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 Accessing the ProDataGraph meta data for an unknown schema on page 66.

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:

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

2. Iterate through the `ProDataObject` list returned in Step 1 on page 64 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 on page 64, 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 on page 111.

- 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:

```
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 Mapping single-valued fields on page 62. 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):

```
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:

```
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:

```
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 on page 64, you can use the following methods:

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

```
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.

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

```
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 ProDataObject class on page 83.

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

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

      Syntax

      ```
      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

```
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

```
int getFieldCount()
```

This value is also identical to the index in the corresponding `ProDataObject` property list that references the corresponding column property (see Using Java SDO classes to access Property meta data on page 111).

For more information on `ProDataObjectMetaData` methods, see ProDataObjectMetaData class on page 101.

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

```
int getNumRelations()
```

b) Get each `ProDataRelationMetaData` object associated with the `ProDataGraph` using the following `ProDataGraphMetaData` method:

Syntax

```
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

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

For more information on `ProDataGraphMetaData` methods, see ProDataGraphMetaData class on page 98.

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`**:
  a) Prepare the `ProDataGaph` as required. For more information, see Preparing an input ProDataGraph on page 68.
  b) 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`**:
  a) Prepare the `ProDataGaph` as required. For more information, see Preparing an input ProDataGraph on page 68.
  b) 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.
  c) Pass the `ProDataGraphHolder` as a parameter to the proxy method.
  d) 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 Passing a ProDataGraph as OUTPUT on page 63.

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 Updating a ProDataSet on page 73.

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 ProDataObjectMetaData class on page 101.

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 ProDataGraph class on page 78.

To create and initialize a **ProDataGraph** with meta data:

1. Create a ProDataGraphMetaData object using the following constructor:

   Syntax

   ```
   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

```
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 ProDataObjectMetaData class on page 101.

3. For each `ProDataObjectMetaData` object created in Step 2 on page 69, 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

   ```
   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 Passing Parameters on page 43
   - 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 ProDataObjectMetaData class on page 101.

4. Add each `ProDataObjectMetaData` object completed in Step 3 on page 70 to the `ProDataGraphMetaData` object created in Step 1 on page 69 using the following `ProDataGraphMetaData` method:

Syntax

```
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

   ```
   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 ProDataRelationMetaData class on page 107.

6. Add each ProDataRelationMetaData object completed in Step 5 on page 71 to the ProDataGraphMetaData object created in Step 1 on page 69 using the following ProDataGraphMetaData method:

   Syntax

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

7. Create the ProDataGraph from the ProDataGraphMetaData object completed in Step 4 on page 70 and Step 6 on page 71 using the following constructor:

   Syntax

   ```
   ProDataGraph(ProDataGraphMetaData dgmd)
   ```

For more information on creating meta data for a ProDataGraph, see ProDataGraphMetaData class on page 98.

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 set DataTypeName(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 Mapping single-valued fields on page 62. 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.

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

     Syntax

     ```
     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 Using Java SDO classes to access Property meta data on page 111.

3. Add each `ProDataObject`, *`dataObj`*, filled with data in Step 2 on page 72 to the `ProDataGraph` using the following `ProDataGraph` method:

   Syntax

   ```
   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 on page 73, generate the parent-child `ProDataObject` references specified by all the `ProDataRelationMetaData` objects contained in the `ProDataGraph` using the following `ProDataGraph` method:

   Syntax

   ```
   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 ProDataGraph class on page 78.

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 on page 73 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 `ProChangesSummary` object, see ProChangeSummary class on page 96. 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 Passing a ProDataGraph as OUTPUT on page 63.

   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 Finding a row in a ProDataGraph on page 76.
   - For information on modifying rows, see Adding data to a ProDataGraph on page 71
   - For information on adding rows, see Adding a row to a ProDataGraph on page 77.
   - For information on deleting rows, see Deleting a row from a ProDataGraph on page 77.

   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 on page 75. This `ProDataGraph` now contains any additional changes, including errors, returned from the AppServer, as follows:

   1. 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 Checking for errors on page 75.
   2. Check the final changes in the output changes-only `ProDataGraph` using the methods of its `ProChangeSummary` object. For more information, see ProChangeSummary class on page 96. 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 on page 74.

7. Once you have merged all final changes into your working `ProDataGraph` in Step 6 on page 75, use the following method to accept all changes in the `ProDataGraph`:

   Syntax

   ```
   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 on page 75 and Step 6 on page 75.
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 on page 74, 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

```
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

```
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

```
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 on page 63.

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 Adding data to a ProDataGraph on page 71.

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 Passing a ProDataGraph as OUTPUT on page 63, and for information on passing a temp-table ProDataGraph as INPUT or INPUT-OUTPUT, see Passing a ProDataGraph as INPUT or INPUT-OUTPUT on page 68.

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 Passing Parameters on page 43.

ProDataGraph class

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 ProDataObject class on page 83.)

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

```
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 ProDataGraphMetaData class on page 98.)

ProDataGraph created from an initial DataGraph

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

Syntax

```
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`.

dgName

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

drList

Specifies an array of `ProDataRelationMetaData` 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 `ProDataRelationMetaData` object that describes this relationship to the `ProDataRelationMetaData` array. (See ProDataRelationMetaData class on page 107.)

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`.

Syntax

```
ChangeSummary getChangeSummary()
```

getRootObject( )

Returns the root `DataObject` of this `DataGraph`.

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.

Syntax

```
void acceptChanges()
```

This method has the same behavior as the `ChangeSummary.beginLogging()` method. For more information, see ProChangeSummary class on page 96.

addProDataObject( )

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

Syntax

```
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.

Syntax

```
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 and `Exception`. For more information on the `ProDataObjectMetaData` class, see ProDataObjectMetaData class on page 101.

getMetaData( )

Returns the `ProDataGraphMetaData` object contained by this `ProDataGraph`.

Syntax

```
ProDataGraphMetaData getMetaData()
```

For more information on the `ProDataGraphMetaData` class, see ProDataGraphMetaData class on page 98.

getNumTables( )

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

Syntax

```
int getNumTables()
```

getProChangeSummary( )

Returns the `ProChangeSummary` object associated with this `ProDataGraph`.

Syntax

```
ProChangeSummary getProChangeSummary()
```

For more information on the `ProChangeSummary` class, see ProChangeSummary class on page 96.

getProDataObjects( )

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

Syntax

```
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 getTableNames( ) on page 82.)

For more information on the `ProDataObject` class, see ProDataObject class on page 83.

getProDataGraphName( )

Returns the name of this `ProDataGraph` object.

Syntax

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

Note

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.

Syntax

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

Note

Typically, each table name is 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. It is set on return from the AppServer when an update to a data source of the corresponding ProDataSet parameter is not successful.

Syntax

```
boolean hasError()
```

setChildTableReferences( )

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

Syntax

```
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 getTableNames( ) on page 82.) 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 ProDataRelationMetaData class on page 107.

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 Using Java SDO classes to access Property meta data on page 111.

The `com.progress.open4gl.ProDataObject` class provides no constructor, because it is created by the `ProDataGraph.createProDataObject()` factory method (see ProDataGraph class on page 78). 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 listed in Mapping single-valued fields on page 62. 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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 `ProDataObjectMetaData` methods. For more information, see ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 `ProDataObjectMetaData` methods. For more information, see ProDataObjectMetaData class on page 101.

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.

Syntax

```
byte[] getBytes(int propertyIndex)
byte[] getBytes(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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
int getInt(int propertyIndex)
int getInt(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 ProDataObjectMetaData class on page 101.

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.

getList( )

Returns the value of a `List` column property, as specified by the overloaded methods.

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 ProDataObjectMetaData class on page 101.

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.

Note

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 Mapping single-valued fields on page 62.

getLong( )

Returns the value of a `long` column property, as specified by the overloaded methods.

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 ProDataObjectMetaData class on page 101.

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.

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 ProDataObjectMetaData class on page 101.

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.

getType( )

Returns the type of the `DataObject`.

Syntax

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

For more information on `DataObject` types, see Using Java SDO classes to access Property meta data on page 111.

isSet( )

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

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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

Syntax

```
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 ProDataObjectMetaData class on page 101.

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

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 Mapping single-valued fields on page 62.

setLong( )

Sets the value of a `long` column property, as specified by the overloaded methods.

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 ProDataObjectMetaData class on page 101.

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.

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`.

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 ProDataRelationMetaData class on page 107.

getFieldCount( )

Returns the number of column properties that this `ProDataObject` represents.

Syntax

```
int getFieldCount()
```

Note

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 (`ProDataRelatonMetaData`). 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.util.GregorianCalendar getGregorianCalendar(int propertyIndex)
java.util.GregorianCalendar getGregorianCalendar(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 ProDataObjectMetaData class on page 101.

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.

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.

getParentRow( )

Returns the parent `ProDataObject` of the current child `ProDataObject`.

Syntax

```
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 ProDataRelationMetaData class on page 107.

getRowErrorString( )

Returns the error string for this `ProDataObject` instance (row).

Syntax

```
String getRowErrorString()
```

Note

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

Syntax

```
String getTableErrorString()
```

getTableName( )

Returns the ABL name of the temp-table that this `ProDataObject` maps to.

Syntax

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

hasRowError( )

Returns `true` if the AppServer returned an error message for the temp-table record (row) mapped to this `ProDataObject`. It is set on return from the AppServer when an update to a data source using this row is not successful.

Syntax

```
boolean hasRowError()
```

hasTableError( )

Returns `true` if the AppServer returned an error message for the temp-table that contains the data from this `ProDataObject` row. 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.

Syntax

```
boolean hasTableError()
```

setGregorianCalendar( )

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

Syntax

```
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 ProDataObjectMetaData class on page 101.

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.

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:

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

beginLogging( )

Clears the list of changes and turns change logging on. This method has the same behavior as the `ProDataGraph.acceptChanges()` method. For more information, see ProDataGraph class on page 78.

Syntax

```
void beginLogging()
```

endLogging( )

Turns off change logging.

Syntax

```
void endLogging()
```

getChangedDataObjects( )

Returns a list consisting of all the `DataObject` instances that have changed while logging.

Syntax

```
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.

Syntax

```
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.

Syntax

```
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.

Syntax

```
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.

Syntax

```
ProDataGraph getChanges()
```

ProDataGraphMetaData class

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 ProDataGraph class on page 78.

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 ProDataObjectMetaData class on page 101).

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 ProDataRelationMetaData class on page 107).

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

Constructor

The `ProDataGraphMetaData` class has the following one constructor:

Syntax

```
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.

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.

Syntax

```
void addDataRelation (ProDataRelatonMetaData drmd)
```

drmd

Specifies a ProDataRelationMetaData object that defines a data-relation for a ProDataGraph.

addTable( )

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

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.

Syntax

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

tablemd

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

getNumRelations( )

Returns the number of ProDataRelationMetaData objects in this ProDataGraphMetaData object.

Syntax

```
int getNumRelations()
```

getNumTables( )

Returns the number of `ProDataObjectMetaData` objects defined in this `ProDataGraphMetaData` object.

Syntax

```
int getNumTables()
```

getRelationMetaData( )

Returns the specified `ProDataRelationMetaData` object in this `ProDataGraphMetaData` object.

Syntax

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

idx

Specifies a 0-based index of data-relations up to the limit specified by getNumRelations( ) on page 99 method.

getTableMetaData( )

Returns the specified `ProDataObjectMetaData` object in this `ProDataGraphMetaData` object.

Syntax

```
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 getTableNames( ) on page 82 method up to the limit specified by getNumTables( ) on page 81 method.

getTableNames( )

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

Syntax

```
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 correponding 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 ProDataObject class on page 83.

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 Using Java SDO classes to access Property meta data on page 111.

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

Constructor

The `ProDataObjectMetaData` class has the following 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]...:primeIdxName.]
[uniqueIdxfld1[,uniqueIdxfldn]...:uniqueIdxName.]..."
```

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*]**...**

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

primeIdxName

Specifies the primary index name.

uniqueIdxfld1 **[**,*uniqueIdxfldn*]**...**

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

uniqueIdxName

Specifies a unique secondary index name.

XMLNamespace

Specifies the namespace for XML serialization or `null`.

XMLPrefix

Specifies the prefix for XML serialization or `null`.

Examples

Note that 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`

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).

Syntax

```
boolean getBImageFlag()
```

getExtent( )

Returns the extent value of the specified column property. If the column property does not represent an array field, the method always returns 0.

Syntax

```
int getExtent(int propertyIndex)
```

propertyIndex

Specifies a 0-based index into the `ProDataObject` property list, limited by the value of getFieldCount( ) on page 93 method. It returns a value that corresponds to the relative position specified by *fieldNumber* in setFieldMetaData( ) on page 106 method.

getFieldCount( )

Returns the number of column properties defined for the specified `ProDataObject` type. The value is identical to the setting of *numFields* in the constructor that instantiates this `ProDataObjectMetaData`.

Syntax

```
int getFieldCount()
```

getFieldName( )

Returns the name of the specified column property.

Syntax

```
int getFieldName(int propertyIndex)
```

propertyIndex

Specifies a 0-based index into the `ProDataObject` property list, limited by the value of getFieldCount( ) on page 93 method. It returns a value that corresponds to the relative position specified by *fieldNumber* in setFieldMetaData( ) on page 106 method.

getNoSchemaMarshal( )

Returns `true` if the most recent invocation of setNoSchemaMarshal( ) on page 106 method indicated that the specified `ProDataObject` collection is to be marshalled to the AppServer without schema information.

Syntax

```
boolean getNoSchemaMarshal()
```

getProType( )

Returns the ABL data type of the specified column property as a class constant value found in `com.progress.open4gl.Parameter`.

Syntax

```
int getProType(int propertyIndex)
```

propertyIndex

Specifies a 0-based index into the `ProDataObject` property list, limited by the value of getFieldCount( ) on page 93 method. It returns a value that corresponds to the relative position specified by *`fieldNumber`* in setFieldMetaData( ) on page 106 method.

For more information on the class constant values provided by the `com.progress.open4gl.Parameter` class, see the information on specifying data type meta data for temp-tables in Passing Parameters on page 43.

getTableName( )

Returns the table name associated with this `ProDataObjectMetaData` object.

Syntax

```
String getTableName()
```

getUserOrder( )

Returns the user order position of the specified column property.

Syntax

```
int getUserOrder(int propertyIndex)
```

propertyIndex

Specifies a 0-based index into the `ProDataObject` property list, limited by the value of getFieldCount( ) on page 93 method. It returns a value that corresponds to the relative position specified by *`fieldNumber`* in setFieldMetaData( ) on page 106 method.

setFieldMetaData( )

Adds the specified field (column property) to the property list of the specified `ProDataObject` type.

Syntax

```
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 Passing Parameters on page 43. To identify the Java data type that the column property assumes for the specified ABL data type, see Mapping single-valued fields on page 62.

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 Using Java SDO classes to access Property meta data on page 111.

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.

Syntax

```
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 a 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( ) on page 83 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 Preparing and passing ProDataSets as ProDataGraph parameters on page 59

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 on page 108
- Creating a data-relation from selected parent and child key fields on page 108

Creating a data-relation from parent and child ProDataObjectMetaData

The following constructor creates a `ProDataRelationMetaData` object with the specified `ProDataObjectMetaData` objects. 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.

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.

For more information, see setColumns( ) on page 110.

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

```
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( ) on page 82 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( ) on page 82 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.

Syntax

```
String getChildTable()
```

getParentColumns( )

Returns the 0-based indexes of the parent table column properties for this `ProDataRelationMetaData` object.

Syntax

```
int[] getParentColumns()
```

getParentTable( )

Returns the name of the parent table for this `ProDataRelationMetaData` object.

Syntax

```
String getParentTable()
```

getRelationName( )

Returns the name of this `ProDataRelationMetaData` object.

Syntax

```
String getRelationName()
```

setColumns( )

Specifies the column properties that are to participate in primary-foreign key relationships between the parent and child tables of this `ProDataRelationMetaData` object, as specified by the overloaded methods.

Syntax

```
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.

Note

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:

  Syntax

  ```
  Property getGetProperty(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:

  Syntax

  ```
  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:

  Syntax

```
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

```
String getName()
```

- **getType( )** — This method returns the Type of the Property. For example:

  Syntax

```
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

```
boolean isMany()
```

6

Extending Proxy Objects

This chapter provides an example to demonstrate the technique for extending proxy objects.

For details, see the following topics:

- 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`.

The following table shows a skeleton of the `Account` delegating class.

Example: Delegating class

```
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.

The following example shows the constructor in the `Tax` class.

Example: SubAppObject/ProcObject constructor

```
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.

The following example shows how a client might extend the `Account` and `Tax` classes.

Example: Class extensions

```
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.

7

Accessing Proxy Properties

This chapter describe the different ways you can access proxy, run-time, and connection properties.

For details, see the following topics:

- Introducing proxy 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

Introducing 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*.

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 Alphabetical listing of properties on page 133.

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 programmatically. 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 Modifying an instantiated Connection object's properties on page 131.

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 on page 121
- General property accessor methods for Connection objects on page 122
- Available properties on page 123
- Accessing properties using purposed accessor methods on page 124

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

```
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

```
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 Available properties on page 123.

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:

```
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

```
public String getStringProperty(String propName)
public int getIntProperty(String propName)
public long getLongProperty(String propName)
public boolean getBooleanProperty(String propName)
```

Set property methods

```
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 Available properties on page 123.

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:

```
ConnectObj.setIntProperty("PROGRESS.Session.initialConnections",5);
```

Where *ConnectObj* is an instance of the `com.progress.open4gl.proxy.Connection` class.

Available properties

The following table lists all of the available properties according to their functional category. For more information on these property categories, see Accessing properties using purposed accessor methods on page 124. For a detailed description of each property, see Alphabetical listing of properties on page 133.

Table 12: Available properties

Category	Properties
Session	• PROGRESS.Session.sessionModel on page 140 • PROGRESS.Session.initialConnections on page 135 • PROGRESS.Session.maxConnections on page 135 • PROGRESS.Session.minConnections on page 136 • PROGRESS.Session.idleConnectionTimeout on page 135 • PROGRESS.Session.connectionLifetime on page 134 • PROGRESS.Session.connectionTimeout on page 134 • PROGRESS.Session.nsClientMinPort on page 137 • PROGRESS.Session.nsClientMaxPort on page 136 • PROGRESS.Session.nsClientPortRetry on page 137 • PROGRESS.Session.nsClientPortRetryInterval on page 137 • PROGRESS.Session.nsClientPicklistSize on page 137 • PROGRESS.Session.nsClientPicklistExpiration on page 137 • PROGRESS.Session.requestWaitTimeout on page 139
Thread control	• PROGRESS.Session.waitIfBusy on page 141
Proxy server	• PROGRESS.Session.proxyHost on page 138 • PROGRESS.Session.proxyPort on page 138 • PROGRESS.Session.proxyUserId on page 139 • PROGRESS.Session.proxyPassword on page 138
Secure Sockets Layer management	• PROGRESS.Session.certificateStore on page 133 • PROGRESS.Session.noHostVerify on page 136 • PROGRESS.Session.noSslSessionReuse on page 136

Category	Properties
Tracing	• PROGRESS.Session.enableTracing on page 134 • PROGRESS.Session.logFileName on page 135 • PROGRESS.Session.loggingLevel on page 135 **Note:** These tracing properties apply to the RunTimeProperties class only. (They do **not** apply to Connection objects.)
Compression	• PROGRESS.Session.CompressionEnabled on page 140 • PROGRESS.Session.CompressionLevel on page 140 • PROGRESS.Session.CompressionThreshold on page 140
Security	• PROGRESS.Session.clientPrincipal on page 134

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:

```
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.javaproxy.Connection class.

Session methods

The following table lists the session property methods. For a detailed description of each property, see Alphabetical listing of properties on page 133.

Table 13: Session methods

Method	Description
setSessionMode(int *sessionModel*)	Sets PROGRESS.Session.sessionModel on page 140
getSessionModel()	Gets PROGRESS.Session.sessionModel on page 140
setInitialConnections (int *nInitialConnections*)	Sets PROGRESS.Session.initialConnections on page 135
getInitialConnections()	Gets PROGRESS.Session.initialConnections on page 135
setMaxConnections (int *nMaxConnections*)	Sets PROGRESS.Session.maxConnections on page 135
getMaxConnections()	Gets PROGRESS.Session.maxConnections on page 135
setMinConnections (int *nMinConnections*)	Sets PROGRESS.Session.minConnections on page 136
getMinConnections()	Gets PROGRESS.Session.minConnections on page 136
setIdleConnectionTimeout (int *nTimeoutSec*)	Sets PROGRESS.Session.idleConnectionTimeout on page 135
getIdleConnectionTimeout()	Gets PROGRESS.Session.idleConnectionTimeout on page 135
setConnectionLifetime (int *nConnectionLifetimeSec*)	Sets PROGRESS.Session.connectionLifetime on page 134
getConnectionLifetime()	Gets PROGRESS.Session.connectionLifetime on page 134
setNsClientMinPort(int *nMinPortNum*)	Sets PROGRESS.Session.nsClientMinPort on page 137
getNsClientMinPort()	Gets PROGRESS.Session.nsClientMinPort on page 137
setNsClientMaxPort (int *nMaxPortNum*)	Sets PROGRESS.Session.nsClientMaxPort on page 136
getNsClientMaxPort()	Gets PROGRESS.Session.nsClientMaxPort on page 136
setNsClientPortRetry (int *nRetries*)	Sets PROGRESS.Session.nsClientPortRetry on page 137
getNsClientPortRetry()	Gets PROGRESS.Session.nsClientPortRetry on page 137
setNsClientPortRetryInterval (int *nDelayMs*)	Sets PROGRESS.Session.nsClientPortRetryInterval on page 137
getNsClientPortRetryInterval()	Gets PROGRESS.Session.nsClientPortRetryInterval on page 137

Method	Description
`setNsClientPicklistSize (int nPickListSize)`	Sets `PROGRESS.Session.nsClientPicklistSize` on page 137
`getNsClientPicklistSize()`	Gets `PROGRESS.Session.nsClientPicklistSize` on page 137
`setNsClientPicklistSize (int nPickListExpiration)`	Sets `PROGRESS.Session.nsClientPicklistExpiration` on page 137
`getNsClientPicklistExpiration()`	Gets `PROGRESS.Session.nsClientPicklistExpiration` on page 137
`setRequestWaitTimeout (int nRqWaitSec)`	Sets `PROGRESS.Session.requestWaitTimeout` on page 139
`getRequestWaitTimeout()`	Sets `PROGRESS.Session.requestWaitTimeout` on page 139

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`).

The methods in the following table provide thread control to the Open Client Runtime. For a detailed description of each property, see Alphabetical listing of properties on page 133.

Table 14: Thread control methods

Method	Description/Property
`setWaitIfBusy()`	Sets `PROGRESS.Session.waitIfBusy` on page 141 to `true`
`setNoWaitIfBusy()`	Sets `PROGRESS.Session.waitIfBusy` on page 141 to `false`
`getWaitIfBusy()`	Gets `PROGRESS.Session.waitIfBusy` on page 141Returns `TRUE` if `setWaitIfBusy()` was called; `FALSE`, if `setNoWaitIfBusy()` was called

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 the following table provide Proxy Web server support to a Java Open Client application. For a detailed description of each property, see Alphabetical listing of properties on page 133.

Table 15: Proxy server specification methods

Method	Description
`setProxyHost(String hostname)`	Sets `PROGRESS.Session.proxyHost` on page 138
`getProxyHost()`	Gets `PROGRESS.Session.proxyHost` on page 138
`setProxyPort(int port)`	Sets `PROGRESS.Session.proxyPort` on page 138
`getProxyPort()`	Gets `PROGRESS.Session.proxyPort` on page 138
`setProxyUserid(String userid)`	Sets `PROGRESS.Session.proxyUserId` on page 139
`getProxyUserId()`	Gets `PROGRESS.Session.proxyUserId` on page 139
`setProxyPassword(String password)`	Sets `PROGRESS.Session.proxyPassword` on page 138
`getProxyPassword()`	Gets `PROGRESS.Session.proxyPassword` on page 138

Secure Sockets Layer management methods

When using HTTPS to access the AppServer, you must supply digital certificates with your client application.

The methods in the following table provide digital certificate management to the Open Client application. For a detailed description of each property, see Alphabetical listing of properties on page 133.

Table 16: Secure Sockets Layer management methods

Method	Description
`setCertificateStore (String certStoreList)`	Sets `PROGRESS.Session.certificateStore` on page 133
`getCertificateStore()`	Gets `PROGRESS.Session.certificateStore` on page 133
`setNoHostVerify (boolean verify)`	Sets `PROGRESS.Session.noHostVerify` on page 136
`getNoHostVerify()`	Gets `PROGRESS.Session.noHostVerify` on page 136
`setnoSslSessionReuse (boolean verify)`	Sets `PROGRESS.Session.noSslSessionReuse` on page 136
`getnoSslSessionReuse()`	Gets `PROGRESS.Session.noSslSessionReuse` on page 136

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 the following table 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 Relationship between the RunTimeProperties class and the Connection class on page 130.

Table 17: Tracing methods

Method	Description
`traceOn(String filename, int level)`	Sets these properties: • `PROGRESS.Session.enableTracing` on page 134 • `PROGRESS.Session.logFileName` on page 135 • `PROGRESS.Session.loggingLevel` on page 135

Method	Description
`traceOn(int level)`	Sets these properties: • PROGRESS.Session.enableTracing on page 134 • PROGRESS.Session.loggingLevel on page 135
`traceOn(String filename)`	Sets these properties: • PROGRESS.Session.enableTracing on page 134 • PROGRESS.Session.logFileName on page 135
`traceOn()`	Sets PROGRESS.Session.enableTracing on page 134
`traceOff()`	Sets PROGRESS.Session.enableTracing on page 134
`isTracing()`	Gets PROGRESS.Session.enableTracing on page 134

Compression methods

You can compress the messages exchanged between the Java Open client and an AppServer. The methods listed in the following table provide compression functionality to the Java Open Client. These methods apply to the `RuntimeProperties` class and `Connection` class.

Table 18: Compression methods

Method	Description
`getCompressionEnabled()`	Gets PROGRESS.Session.CompressionEnabled on page 140Returns `TRUE` if `getCompressionEnabled()` is called. The default value is `FALSE`.
`getCompressionLevel()`	Gets PROGRESS.Session.CompressionLevel on page 140
`getCompressionThreshold()`	Gets PROGRESS.Session.CompressionThreshold on page 140
`setCompressionEnabled()`	Sets PROGRESS.Session.CompressionEnabled on page 140

Method	Description
setCompressionLevel()	Sets PROGRESS.Session.CompressionLevel on page 140
setCompressionThreshold()	Sets PROGRESS.Session.CompressionThreshold on page 140

The following example shows the creation of a new Connection object, which is compression enabled with a default compression level of 1 and a minimum message size of 300 bytes.

```
com.progress.open4gl.javaproxy.Connection conn = new
Connection("AppServer://localhost:5162/asbroker1", null, null, "TestApp");

conn.setCompressionEnabled(true);

conn.setCompressionLevel(1);

conn.setCompressionThreshold(300);
```

The following example shows the creation of a new RuntimeProperties object, which is compression enabled. The compression level is set to 5, and the message size is set to 550 bytes.

```
com.progress.open4gl.RuntimeProperties.setCompressionEnabled(true);

com.progress.open4gl.RuntimeProperties.setCompressionLevel (5);

com.progress.open4gl.RuntimeProperties.setCompressionThreshold (550);
```

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 the following table.

Table 19: Relationship between RunTimeProperties and Connection object properties

If the RunTimeProperties property has . . .	And the property in the Connection object has . . .	Then the value of the property is returned from the . . .
Not changed	Not changed	RunTimeProperties class
Changed	Not changed	RunTimeProperties class
Not changed	Changed	Connection object
Changed	Changed	Connection object

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:

```
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 the following table.

Table 20: Relationship between Connection object properties and session model

If this Connection object property is updated . . .	The update is . . .	
	Session-managed	Session-free
PROGRESS.Session.AppServerKeepalive on page 133	Recognized once pick list is refreshed	Recognized once pick list is refreshed
PROGRESS.Session.certificateStore on page 133	Ignored	Ignored
PROGRESS.Session.connectionLifetime on page 134	*Not applicable*	Ignored
PROGRESS.Session.connectionTimeout on page 134	Recognized once pick list is refreshed	Recognized once pick list is refreshed
PROGRESS.Session.idleConnectionTimeout on page 135	*Not applicable*	Ignored
PROGRESS.Session.initialConnections on page 135	*Not applicable*	Ignored
PROGRESS.Session.maxConnections on page 135	*Not applicable*	Ignored
PROGRESS.Session.minConnections on page 136	*Not applicable*	Ignored
PROGRESS.Session.noHostVerify on page 136	Ignored	Ignored
PROGRESS.Session.nsClientMaxPort on page 136	*Not applicable*	Ignored
PROGRESS.Session.nsClientMinPort on page 137	*Not applicable*	Ignored
PROGRESS.Session.nsClientPicklistExpiration on page 137	*Not applicable*	Recognized once pick list is refreshed
PROGRESS.Session.nsClientPicklistSize on page 137	*Not applicable*	Recognized when pick list is refreshed
PROGRESS.Session.nsClientPortRetry on page 137	*Not applicable*	Ignored
PROGRESS.Session.nsClientPortRetryInterval on page 137	*Not applicable*	Ignored
PROGRESS.Session.proxyHost on page 138	Ignored	Ignored
PROGRESS.Session.proxyPassword on page 138	Ignored	Ignored
PROGRESS.Session.proxyPort on page 138	Ignored	Ignored
PROGRESS.Session.proxyUserId on page 139	Ignored	Ignored
PROGRESS.Session.requestWaitTimeout on page 139	*Not applicable*	Recognized for new requests

If this Connection object property is updated . . .	The update is . . .	
	Session-managed	Session-free
PROGRESS.Session.sessionModel on page 140	Ignored	Ignored
PROGRESS.Session.waitIfBusy on page 141	Recognized	Recognized for new requests

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. If configured for the client, the Keepalive feature allows the client to ping the AppServer at set intervals to determine if the AppServer is still viable.Valid values are:

- AllowServerASK
- DenyServerASK
- AllowClientASK
- DenyClientASK

The absence of this property indicates that the default value for the AppServer Keepalive protocol will be used on this connection.

Default: AllowServerASK, DenyClientASK

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` `on page 133` 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.clientPrincipal

Data Type: `ClientPrincipal` object

Session model: All

A reference to the `ClientPrincipal` object associated with the `Connection` object. Setting this attribute to null will delete any `ClientPrincipal` object that has been previously associated with the `Connection` object.

Default: null

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 on page 136` 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.connectionTimeout

Data Type: Integer

Session model: Session-free and Session-managed

The connectionTimeout property specifies the duration, in integer, for which a client must stay connected to the session-managed or session-free AppServer. A positive integer value indicates the number of milliseconds after which the client will time out with a connection-timeout or a socket-timeout exception, and then disconnect from the AppServer.

You use this property when you want a client program to disconnect from a long executing connection request.

Default: 0 (indefinite wait)

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 on page 136 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` on page 135.

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 on page 137 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 on` property, you must also specify the `PROGRESS.Session.proxyPort on page 138` 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 on page 138` and `PROGRESS.Session.proxyPort on page 138` are specified and the Proxy server requires authentication.

If `PROGRESS.Session.proxyHost on page 138` and `PROGRESS.Session.proxyPort on page 138` 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 on page 139` 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 on page 138` 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 on page 138` and `PROGRESS.Session.proxyPort on page 138` 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 the following table.

Table 21: requestWaitTimeout property values

If the value is . . .	The Open Client Runtime . . .
-1	Queues the request indefinitely until an AppServer session becomes available
0	Rejects the request and returns an error message to the client indicating that there are too many concurrent requests
N	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

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 the following table.

Table 22: sessionModel property values

Value	Description
0	Session-managed
1	Session-free

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.CompressionEnabled

Data Type: Logical

Session model: Session-free and Session-managed

A logical property that allows you to enable or disable compression. This property must be set before establishing the connection between AppServer and the Java Open Client. If this property is modified after the connection is established, it does not affect the existing session between other clients and the AppServer.

Default: False

PROGRESS.Session.CompressionLevel

Data Type: Integer

Session model: Session-free and Session-managed

An integer value that sets the compression level for compressing the body of a message sent to and from the AppServer. The highest supported value for compression is 9. The valid range for compression is 1-9. If a value is set beyond this range, then, the value that is closest to the range is used. If a value greater than 9 or less than 1 is used, then, the compression level is set to 9 or 1 respectively. The higher the level of compression, the more time it takes to compress the message. Higher compression level also results in smaller messages.

Default: 1

PROGRESS.Session.CompressionThreshold

Data Type: Integer

Session model: Session-free and Session-managed

An integer value that sets the minimum size in bytes that a message body must have before it is compressed. The minimum value for compression threshold is 256. The highest level for compression threshold is 32764. If a value is set beyond the specified range, then, the value that is closest to the range is used. If a value greater than 32764 or less than 256 is used, then, the size of the message is set to 32764 or 256 bytes respectively.

Default: 256

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

8

Handling Errors

This chapter provides details on handling the different kinds of exceptions you may encounter.

For details, see the following topics:

- 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.

For information on the methods/properties of these classes and subclasses, see the Open Client Toolkit documentation in *OpenEdge-install-directory*`/java/doc/com/progress/open4gl/package-summary.html`.

The `Exception` class hierarchy is shown in the next section followed by descriptions of the `Exception` classes.

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.AppServerIOException
            com.progress.open4gl.BadURLException
            com.progress.open4gl.BrokerIOException
            com.progress.open4gl.ConnectAIAException
            com.progress.open4gl.ConnectFailedException
            com.progress.open4gl.ConnectHttpAuthException
            com.progress.open4gl.ConnectHttpsAuthException
            com.progress.open4gl.ConnectProtocolException
            com.progress.open4gl.ConnectProxyAuthException
            com.progress.open4gl.HostUnknownException
            com.progress.open4gl.InvalidNameServerPortException
            com.progress.open4gl.MsgVersionInvalidException
            com.progress.open4gl.NameServerClientPortException
            com.progress.open4gl.NameServerClientPortRangeException
            com.progress.open4gl.NameServerClientPortRetryException
            com.progress.open4gl.NameServerCommunicationsException
            com.progress.open4gl.NameServerInterruptException
            com.progress.open4gl.NameServerMessageFormatException
            com.progress.open4gl.NoSuchAppServiceException
            com.progress.open4gl.UnknownHostnameException
         com.progress.open4gl.OutputSetException
         com.progress.open4gl.ProDataException
         com.progress.open4gl.RunTime4GLException
            com.progress.open4gl.RunTime4GLErrorException
               com.progress.open4gl.RunTime4GLAppErrorException
               com.progress.open4gl.RunTime4GLStopErrorException
            com.progress.open4gl.RunTime4GLQuitException
            com.progress.open4gl.RunTime4GLStopException
               com.progress.open4gl.RunTime4GLStopAfterException
               com.progress.open4gl.RunTime4GLLockConflictException
         com.progress.open4gl.SchemaValidationException
         com.progress.open4gl.SystemErrorException
            com.progress.open4gl.NetworkException
               com.progress.open4gl.NetworkException.RecvDataException
               com.progress.open4gl.NetworkException.SendDataException
   java.sql.SQLException
      com.progress.open4gl.ProSQLException
         com.progress.open4gl.DeletedRowAccessException
         com.progress.open4gl.SDOModificationException
```

Exception descriptions

The following table shows the general ABL exception classes, in alphabetical order.

Table 23: General ABL exceptions for Java proxies

Exception class	Description
`AppServerIOException`	A general I/O exception was raised while initializing a connection to an OpenEdge application server, and the connection was not completed.
`BadURLException`	Invalid URL format in connection parameter.
`BrokerIOException`	A general I/O exception was raised while attempting to initiate a connection to an OpenEdge application server, and the connection was not completed.
`BusySessionException`	Busy servicing another request.
`ConnectAIAException`	A connection attempt to an AIA/APSV adapter returned and the connection was not completed.
`ConnectException`	Error connecting to AppServer.
`ConnectFailedException`	Failed AppServer connection attempt. The message contains the specified reason for the failure.
`ConnectHttpAuthException`	The HTTP BASIC userid/password authentication failed when connecting to an AIA/APSV adapter, and the connection was not completed.
`ConnectHttpsAuthException`	The HTTPS (TLS) connection's authentication failed when connecting to an AIA/APSV adapter, and the connection was not completed.
`ConnectProtocolException`	An uncategorized error occurred while attempting to initiate and initialize a connection to an OpenEdge application server, and the connection was not completed.
`ConnectProxyAuthException`	An error was detected while attempting to initiate and initialize a connection to an AIA/APSV adapter via an HTTP proxy server, and the connection was not completed.
`DeletedRowAccessException`	Application tried to access a deleted row in a `SDOResultSet` object.
`HostUnknownException`	Invalid NameServer host specified in the URL.
`InvalidNameServerPortException`	Invalid NameServer port specified in the URL.
`MsgVersionInvalidException`	Incorrect version for message received from NameServer.
`NameServerClientPortException`	A free UDP network port could not be found when attempting to connect to a NameServer and resolve the address of an OpenEdge authentication server, and the connection was not completed.

Exception class	Description
NameServerClientPortRangeException	The range of UDP port numbers configured for connecting to a NameServer included an invalid value, and no NameServer connection was attempted.
NameServerClientPortRetryException	The number of allowed retries to connect to a NameServer and resolve an OpenEdge application server network address was exceeded, and the connection was not completed.
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.
NetworkException	A socket I/O error occurred on the connection to the AppServer.
NoSuchAppServiceException	Application Service specified in URL unknown to NameServer.
Open4GLException	The root exception for all Open Client conditions.
OutputSetException	Output ResultSet error, for example, trying to get a column value out of column order or trying to make an illegal type conversion.
ProDataException	Standard exception for errors that occur while using the ProDataGraph object.
ProSQLException	Error occurred when using a `java.sql.ResultSet` parameter.
RecvDataException	An error occurred while receiving a message.
RunTime4GLAppErrorException	ABL application error. The message contains user-defined data specific to the application.
RunTime4GLErrorException	ABL `ERROR` condition.
RunTime4GLException	ABL run-time conditions.
RunTime4GLLockConflictException	ABL time-out condition while waiting for a record lock. The message contains data for identifying the device, database table, and user that hold the record lock.
RunTime4GLQuitException	ABL `QUIT` condition.

Exception class	Description
RunTime4GLStopAfterException	ABL time limit exceeded for a STOP-AFTER directive on a DO, FOR, or REPEAT statement.
RunTime4GLStopErrorException	ABL STOP condition raised due to certain unexpected error conditions. The message contains information on the originating error.
RunTime4GLStopException	ABL STOP condition.
SchemaValidationException	A schema validation inconsistency occurred when using the ProDataGraph object.
SDOModificationException	An update, delete, or insert failed on an SDOResultSet object.
SendDataException	An error occurred while sending a message.
SystemErrorException	An unexpected error that indicates an ABL bug or an unusual environmental problem (such as running out of disk space).
UnknownHostnameException	The host address/DNS name specified when connecting to an OpenEdge application server was invalid and the connection was not completed.

Output ResultSet exceptions

OpenEdge defines ProSQLException, which is returned from all methods on an OUTPUT TABLE or TABLE-HANDLE ResultSet, including the SDOResultSet object. For example:

```
ProSQLException class 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 Using SmartDataObjects from Java Clients on page 149.

9

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.

For details, see the following topics:

- What is a SmartDataObject?
- Accessing remote SmartDataObjects from Java
- SmartDataObject access tools and documentation
- 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

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*.

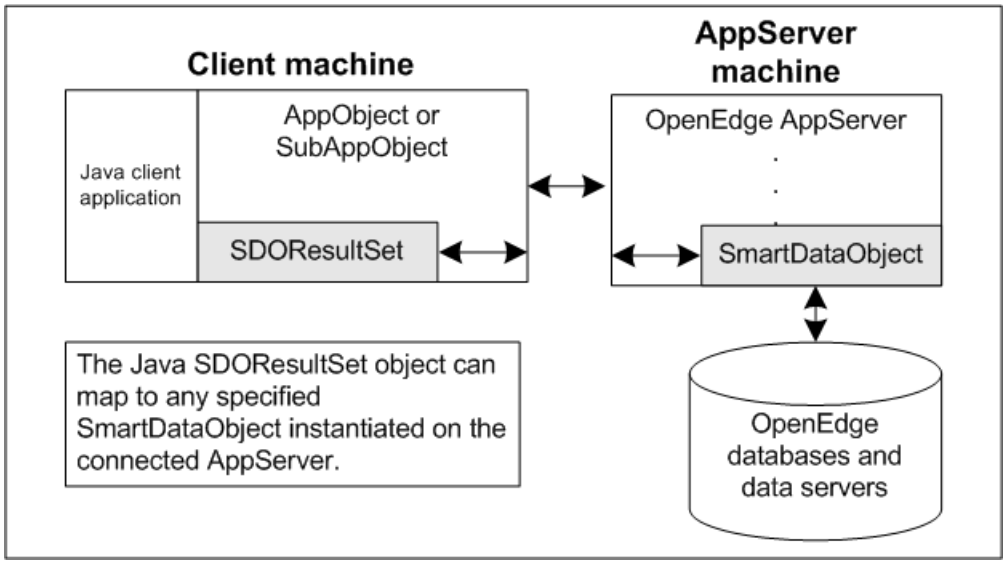
Using the OpenEdge AppBuilder, an ABL programmer can modify and extend the standard SmartDataObject methods to encapsulate any business logic needed to filter data into or out of an OpenEdge-managed database. When the SmartDataObject is deployed remotely (running on an AppServer), this same business logic becomes available to manage data from your Java Open Client application.

Accessing remote SmartDataObjects from Java

The following figure 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.

Figure 2: Java Open Client access to a remote SmartDataObject



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:

```
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 Configuring and Deploying Java Open Client Applications on page 15.

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.
6. 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

```
public SDOAppObject(com.progress.open4gl.javaproxy.Connection connectObj)
```

Syntax

```
public SDOAppObject(String urlString, String userId, String password,
   String AppServer-info)
```

Syntax

```
public SDOAppObject(String userId, String password, String AppServer-info)
```

Syntax

```
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 Connection class on page 33.

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 Connecting to an AppServer on page 33.

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:

```
public SDOResultSet _createSDOResultSet(String sdoName)
```

The following version uses the KEEP_ROWS scrolling mode:

```
public SDOResultSet _createSDOResultSet(String sdoName, String where,
   String sortBy)
```

The following version uses whatever scrolling mode is specified by *parameters*:

```
public SDOResultSet _createSDOResultSet(String sdoName, String where,
   String sortBy, SDOParameters parameters)
```

For more information, see Understanding SDOResultSet scrolling modes on page 158

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.open4gl` 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:

```
void setRowIdentity(String rowId)
```

A null `rowId` is equivalent to not calling `setRowIdentity()` (also see the `reOpenQuery(String rowId)` extended method in Miscellaneous management methods on page 169. 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 Understanding SDOResultSet scrolling modes on page 158.

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 Miscellaneous management methods on page 169.

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 Understanding SDOResultSet stateless mode on page 159.

The following method gets the `Stateless` mode for the SDOResultSet:

Syntax

```
boolean getStateless()
```

For more information, see Understanding SDOResultSet stateless mode on page 159.

The following method sets the scrolling mode for the SDOResultSet specified as a class constant in `com.progress.open4gl.SDOScrollingMode`:

Syntax

```
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 Understanding SDOResultSet scrolling modes on page 158.

The following method gets the scrolling mode specified for the SDOResultSet, returned as a class constant in `com.progress.open4gl.SDOScrollingMode`:

Syntax

```
SDOScrollingMode getScrollingMode()
```

For more information, see Understanding SDOResultSet scrolling modes on page 158.

The following method sets the maximum number of rows to be fetched when the scrolling mode is `SDOScrollingMode.PREFETCH`:

Syntax

```
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 Understanding SDOResultSet scrolling modes on page 158 and Understanding SDOResultSet stateless mode on page 159.

The following method gets the value for the maximum number of rows to be fetched when the scrolling mode is `SDOScrollingMode.PREFETCH`:

Syntax

```
int getPrefetchMaxRows()
```

For more information, see Understanding SDOResultSet scrolling modes on page 158 and Understanding SDOResultSet stateless mode on page 159.

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 Updating SDOResultSet objects on page 164.) 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 Understanding SDOResultSet stateless mode on page 159.

- **`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 Understanding SDOResultSet stateless mode on page 159.

- **`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 Using batch mode [extension] on page 167.
- Stateless mode is only allowed with the `PREFETCH` scrolling mode. For more information, see Understanding SDOResultSet scrolling modes on page 158.

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 Miscellaneous management methods on page 169.) 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:

```
 // The code shows the fetching of 19 rows in two batches under
// the Stateless mode.

SDOParameters params = new SDOParameters();
params.setStateless(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] on page 161
- Navigating SDOResultSet rows on page 162
- Getting SDOResultSet column values on page 163

- Updating SDOResultSet objects on page 164
- Miscellaneous management methods on page 169
- Meta data methods on page 171

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 Miscellaneous management methods on page 169.
- **`sendBatch()`** — See the information on `Batch` mode in Updating SDOResultSet objects on page 164
- **`sendBatchAndReOpen()`** — See the information on `Batch` mode in Updating SDOResultSet objects on page 164.

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

```
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

```
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 the following table change or report the position of the cursor in the SDOResultSet.

Table 24: Navigating SDOResultSet rows

Syntax	Description
`boolean absolute(int n)`	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.
`boolean previous()`	Movesthe cursor backwards.
`boolean next()`	Moves the cursor forward.
`boolean relative(int n)`	Moves the cursor *n* rows forward (backward if *n* is negative).
`boolean first()`	Positionsthe cursor on the first row.
`void beforeFirst()`	Positionsthe cursor before the first row.
`boolean last()`	Positions the cursor on the last row.
`void afterLast()`	Positions the cursor after the last row.
`boolean isAfterLast()`	Is the cursorafter the last row?
`boolean isFirst()`	Is the cursor before the first row?

Syntax	Description
`boolean isLast()`	Is the cursor on the last row?
`int getRow()`	Gets the position (1-based) of the cursor.

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 Passing parameters on page 206.

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 Using batch mode [extension] on page 167. For more information on automatic transactions, see Using extended transactions on page 166.

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 Errors and exceptions on page 173.

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 Passing Temp-tables as SQL ResultSet Parameters on page 221.

Inserting, deleting, and updating rows

The methods in the following table manage operations on rows in the SDOResultSet.

Table 25: Managing operations on SDOResultSet rows

Method	Description
`boolean rowDeleted()`	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 Using batch mode [extension] on page 167.
`boolean rowInserted()`	Returns `true` if the cursor is positioned on an inserted row position and the row is not yet sent to the AppServer.
`boolean rowUpdated()`	Returns `true` if the cursor is positioned on a row that is modified but not yet sent to the AppServer.
`void CancelRowUpdates()`	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()`.
`void deleteRow()`	Deletes the row in the current position and sends the delete request to the AppServer.
`void insertRow()`	Sends the newly created row to the AppServer.
`void moveToCurrentRow()`	Moves the cursor from the insert row back to the current position.
`void moveToInsertRow()`	Moves the cursor to a staging position for creating a new row.
`void updateRow()`	Sends the updates for the current row to the AppServer.

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 Using batch mode [extension] on page 167). 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 Visibility of updates on page 168.

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.

The following table lists the SDOResultSet methods that support `Batch` mode.

Table 26: SDOResultSet methods that support Batch mode

Method	Description
`void cancelBatch() [Extension]`	Undoes all the modifications of the current batch.
`void sendBatch() [Extension]`	Sends the current batch to the AppServer.
`void sendBatchAndReOpen() [Extension]`	Sends the current batch to the AppServer, then reopens the ResultSet. In `Stateless` mode, this method is more efficient than separately calling `sendBatch()` and `reOpenQuery()`. For more information, see Visibility of updates on page 168.

Method	Description
`void startBatch() [Extension]`	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.
`boolean inBatch() [Extension]`	Returns `true` if the SDOResultSet object is in `Batch` mode (that is, you have called `startBatch()` and have not yet called `sendBatch()` or `sendBatchandReopen()`). **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 the following table.

Table 27: Standard JDBC 2 management methods

Method	Description
`void close()`	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.
`int findColumn (String columnName)`	Returns the column index.
`int getFetchSize()`	Returns the size (in number of rows) of the buffer the underlying implementation uses to fetch rows from the AppServer.
`void refreshRow()`	Gets the latest version of the current row from the database.
`SDOResultSetMetaData getMetaData()`	Returns the SDOResultSetMetaData object, which is an extension of the JDBC ResultSetMetaData object.
`boolean wasNull()`	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: `int cValue; cValue = SDOResults.getInt("QUANTITY"); if (SDOResults.wasNull()) return(-1);` 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.

SmartDataObject management methods [extension]

The nonstandard SDOResultSet methods listed in the following table 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 28: Nonstandard SDOResultSet management methods

Method	Description
`boolean absolute (String rowId) [Extension]`	Repositions the cursor at the row specified by *rowId*.
`String getQuery() [Extension]`	Returns the ABL query that populated this SDOResultSet object.
`String getRowIdentity() [Extension]`	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)`.
`SDOInterface getSDOInterface() [Extension]`	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 `getSDOInterface()` creates a remote SmartDataObject procedure you can delete only by calling `releaseSDOInterface()`. Calling `releaseSDOInterface()` on a non-stateless SDOResultSet has no effect. See Note on SDOInterface method on page 171.
`void releaseSDOInterface() [Extension]`	On a stateless SDOResultSet, releases the remote SmartDataObject procedure. Calling `releaseSDOInterface()` on an SDOResultSet that is not `Stateless` has no effect. This method is used in conjunction with `getSDOInterface()`.
`void reOpenQuery() [Extension]`	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.
`void reOpenQuery (String rowId) [Extension]`	Works like `reOpenQuery()`, but the first row returned is the one specified by *rowId*. A null *rowId* is equivalent to `reOpenQuery()`. Usually it is more efficient to call `reOpenQuery(String rowId)` than to first call `reOpenQuery()` and then call `absolute(String rowId)` to reposition on the specified row. **Note:** If the SDOResultSet is in `SDOScrollingMode.PREFETCH` mode, you cannot get the rows before this *rowId*. For more information on `SDOScrollingMode.PREFETCH`, see Understanding SDOResultSet scrolling modes on page 158.

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 the following table:

Table 29: Standard JDBC meta data methods

Method	Description
`int getColumnCount()`	The number of columns in the SDOResultSet
`int getColumnDisplaySize (int columnIndex)`	The maximum display size width of this column in characters
`String getColumnLabel (int columnIndex)`	The ABL dictionary label for this column
`String getColumnName (int columnIndex)`	Thenameof the column
`int getColumnType (int columnIndex)`	The SQL type of the column
`String getColumnTypeName (int columnIndex)`	The ABL data type name of the column
`String getTableName (int columnIndex)`	The table name for this column. This is especially useful where you want to determine the original table for a column participating in a table join
`int isNullable (int columnIndex)`	0 if this column is mandatory (does not allow the `Unknown value (?)`); 1 if it is not mandatory (allows the `Unknown value (?)`)
`boolean isReadOnly (int columnIndex)`	True if the column is an SDOResultSet read-only column
`boolean isWritable (int columnIndex)`	False if the column is an SDOResultSet read-only column

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.

The following table lists the supported nonstandard SDOResultSetMetaData methods.

Table 30: Nonstandard SDOResultSetMetaData methods

Method	Description
`String getColumnFormat (int column) [Extension]`	The ABL dictionary display format
`String getColumnInitalValue (int column) [Extension]`	The ABL dictionary initial value as a string
`String getColumnJavaTypeName (int column) [Extension]`	The Java type name
`int getColumnProType (int column) [Extension]`	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 Passing parameters on page 206.
`String getColumnValExp (int column) [Extension]`	The ABL dictionary validation expression
`String getColumnValMsg (int column) [Extension]`	The ABL dictionary validation message

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. The following table shows the full exception hierarchy. Indentation in the table indicates the level of the hierarchy.

Table 31: Output ResultSet exceptions for SDOResultSet

Exception class hierarchy	Description
`java.sql.SQLException`	Root exception for all JDBC 2 exceptions
`com.progress.open4gl.ProSQLException`	Root exception for all SDOResultSet exceptions
`com.progress.open4gl.DeletedRowAccessException`	Attempted to access a deleted row
`com.progress.open4gl.SDOModificationException`	Failed attempt to modify an existing row

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

```
String getFields()
```

This method returns the name of the table (if any) related to this error message:

Syntax

```
String getTable()
```

This method returns the text of the current error message:

Syntax

```
String getText()
```

This method positions the cursor on the next error message:

Syntax

```
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 `getDate()` 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

10

Using the Open Client Java OpenAPI to Directly Access the AppServer

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 an access 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.

For details, see the following topics:

- Overview of Java OpenAPI classes
- Connecting to the AppServer
- Running procedures and user-defined functions
- Setting up parameters
- Passing parameters
- Handling returned values
- Sample Java OpenAPI code

Overview of Java OpenAPI classes

The following table shows a summary of the classes that support the Java OpenAPI.

Table 32: Java OpenAPI Classes

Class	Description
`com.progress.open4gl.javaproxy.OpenAppObject`	For accessing external ABL procedures in an application service using the OpenAPI
`com.progress.open4gl.javaproxy.OpenProcObject`	For accessing a persistent, single-run, or singleton procedure using the OpenAPI
`com.progress.open4gl.javaproxy.ParamArray`	An object containing an array for passing parameters to an ABL procedure or user-defined function
`com.progress.open4gl.javaproxy.ParamArrayMode`	Constants for specifying the mode of an ABL procedure or user-defined function parameter (`INPUT`, `INPUT-OUTPUT`, or `OUTPUT`)
`com.progress.open4gl.Parameter`	Constants for specifying the ABL data type of a procedure or user-defined function parameter, return type, or temp-table field
`com.progress.open4gl.ProcedureType`	A Java enum for specifying a procedure type as `PERSISTENT`, `SINGLE_RUN`, or `SINGLETON`
`com.progress.open4gl.ProDataGraphMetaData`	For defining the schema of an ABL ProDataSet or temp-table mapped to a `ProDataGraph`[4]
`com.progress.open4gl.ProDataObjectMetaData`	For defining the schema of an ABL temp-table contained within a `ProDataGraph` (mapped to a `ProDataObject`)[5]
`com.progress.open4gl.ProDataRelationMetaData`	For defining a ProDataSet data-relation contained within a `ProDataGraph`[5]
`com.progress.open4gl.dynamicapi.ProResultSetMetaDataImpl`	For defining the schema of an ABL temp-table mapped to an SQL `ResultSet`[5]

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 `com.progress.open4gl.javaproxy.OpenAppObject`. You can create an `OpenAppObject` using one of the following constructors:

[4] For complete description of the ProDataGraph class and its associated classes, see Accessing ABL ProDataSets on page 53.
[5] For complete information on temp-table parameters mapped to an SQL ResultSet, see Passing Temp-tables as SQL ResultSet Parameters on page 221.

Syntax

```
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)
```

connectObj

Specifies a `com.progress.open4gl.javaproxy.Connection` object, which defines connection parameters to access the AppServer. For more information on this object, see Connecting to an AppServer on page 33.

appservice

Specifies the name of the application service provided by the AppServer.

url

Specifies the URL to connect to an AppServer URL. Defaults to `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 *OpenEdge Application Server: Developing AppServer Applications*.

userid

Specifies a user ID required to access the AppServer.

password

Specifies a password required to access the AppServer.

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 `PROGRESS.Session.sessionModel` to 1. For more information on setting this and other session properties see Accessing Proxy Properties on page 119.

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, persistent, single-run, or singleton procedure that is available on the `OpenAppObject`. (Running a persistent, single-run, or singleton 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 Setting up parameters on page 182. 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 Passing parameters on page 206.

After running any procedure or user-defined function, you must handle any output (returned values) or error conditions. For more information, see Handling returned values on page 209.

The following sections describe:

- Running a non-persistent procedure on an OpenAppObject on page 180
- Running a persistent procedure (OpenProcObject) on an OpenAppObject on page 180
- Running a single-run or singleton procedure (OpenProcObject) on an OpenAppObject on page 181
- Running an internal procedure or user-defined function on an OpenProcObject on page 182

For examples that show how to run application service code, see Sample Java OpenAPI code on page 210.

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

```
public void runProc(String procName, ParamArray paramArray)
                  throws Open4GLException, RunTime4GLException,
                  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 Creating a parameter array on page 183.

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

```
public OpenProcObject createPO(String procName, ParamArray paramArray)
                  throws Open4GLException, RunTime4GLException,
                  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 Creating a parameter array on page 183. A procedure with no parameters can also be run as persistent using an alternative syntax (see Running a single-run or singleton procedure (OpenProcObject) on an OpenAppObject on page 181), or the null value can be passed through *paramArray*.

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 a single-run or singleton procedure (OpenProcObject) on an OpenAppObject

You can run a procedure with no parameters in the main block as single-run, singleton, or persistent on the connected AppServer using the following method on the `OpenAppObject`:

Syntax

```
public OpenProcObject createPO(String procName, ProcedureType procType)
                  throws Open4GLException, RunTime4GLException,
                  SystemErrorException
```

procName

Specifies the name of the procedure to run, including any path relative to the `PROPATH` setting for the AppServer.

procType

A Java enum type that specifies the procedure as persistent (`ProcedureType.PERSISTENT`), single-run (`ProcedureType.SINGLE_RUN`), or singleton. (`ProcedureType.SINGLETON`).

This method returns a `com.progress.open4gl.javaproxy.OpenProcObject`, which you can use to run internal procedures and user-defined functions provided by the procedure.

Note: The main block of a singleton or single-run procedure cannot have any parameters, so there is no *paramArray* available with this syntax. Although persistent procedures can have parameters in general, only persistent procedures without parameters can be run using this syntax.

The `OpenProcObject` also contains the method `getProcedureType()`, which returns `PERSISTENT`, `SINGLE_RUN`, or `SINGLETON`, depending on the procedure type of the object:

```
public ProcedureType getProcedureType()
```

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 Defining the return type for a user-defined function on page 194. For information on accessing the return value after calling the function, see Accessing user-defined function return values on page 210.

Run the internal procedure or user-defined function using the following method on the `OpenProcObject`:

Syntax

```
public void runProc(String procName, ParamArray paramArray)
                  throws Open4GLException, RunTime4GLException,
                  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 Creating a parameter array on page 183.

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 on page 183
- Creating a parameter array on page 183
- Adding parameters using data type-specific methods on page 184
- Adding parameters using a generic method on page 191

- Defining the return type for a user-defined function on page 194
- Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet on page 197
- Defining the schema for a ProDataSet parameter on page 199
- Defining the schema for temp-tables in a ProDataSet on page 202
- Defining the schema for a temp-table parameter mapped to a ProDataGraph on page 205

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 Passing Parameters on page 43.

Creating 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.

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:

```
// 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 add parameter methods

The general syntax of these methods is as follows:

Syntax

```
public void addProgressType(int position, DataTypevalue, int mode,
                    [ int extentValue | MetaTypemetaData ])
            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 Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet on page 197.
- **`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 Defining the schema for a ProDataSet parameter on page 199. For more information on defining the schema for temp-table parameters mapped to a `ProDataGraph`, see Defining the schema for a temp-table parameter mapped to a ProDataGraph on page 205.

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 on page 186
- COM-HANDLE on page 186
- DATASET and DATASET-HANDLE on page 186
- DATE on page 187
- DATETIME on page 187
- DATETIME-TZ on page 187
- DECIMAL on page 188
- HANDLE on page 188
- INT64 on page 188
- INTEGER on page 189
- LOGICAL on page 189
- LONGCHAR on page 189
- MEMPTR on page 190
- RAW on page 190
- RECID on page 190
- ROWID on page 191
- TABLE and TABLE-HANDLE on page 191

CHARACTER

These are the data type-specific methods for adding an ABL `CHARACTER` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

COM-HANDLE

These are the data type-specific methods for adding an ABL `COM-HANDLE` parameter:

Syntax

```
public void addCOMHandle(int position, com.progress.open4gl.COMHandle value,
                         int mode)
            throws Open4GLException
public void addCOMHandleArray(int position,
                              com.progress.open4gl.COMHandle[] value,
                              int mode, int extentValue)
            throws Open4GLException
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 184.

DATASET and DATASET-HANDLE

These are the data type-specific methods for adding an ABL ProDataSet (`DATASET` or `DATASET-HANDLE`) parameter:

Syntax

```
public void addDataset
               (int position,
                com.progress.open4gl.ProDataGraph value, int mode,
                com.progress.open4gl.ProDataGraphMetaData metaData)
            throws Open4GLException
public void addDatasetHandle
               (int position,
                com.progress.open4gl.ProDataGraph value, int mode,
                com.progress.open4gl.ProDataGraphMetaData metaData)
            throws Open4GLException
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 184.

DATE

These are the data type-specific methods for adding an ABL `DATE` parameter:

Syntax

```
public void addDate(int position, java.util.GregorianCalendar value,
                    int mode)
            throws Open4GLException
public void addDateArray(int position, java.util.GregorianCalendar[] value,
                         int mode, int extentValue)
            throws Open4GLException
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 184.

DATETIME

These are the data type-specific methods for adding an ABL `DATETIME` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

DATETIME-TZ

These are the data type-specific methods for adding an ABL `DATETIME-TZ` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

DECIMAL

These are the data type-specific methods for adding an ABL `DECIMAL` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

HANDLE

These are the data type-specific methods for adding an ABL `HANDLE` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

INT64

These are the data type-specific methods for adding an ABL `INT64` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

INTEGER

These are the data type-specific methods for adding an ABL `INTEGER` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

LOGICAL

These are the data type-specific methods for adding an ABL `LOGICAL` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

LONGCHAR

These are the data type-specific methods for adding an ABL `LONGCHAR` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

MEMPTR

These are the data type-specific methods for adding an ABL `MEMPTR` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

RAW

These are the data type-specific methods for adding an ABL `RAW` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

RECID

These are the data type-specific methods for adding an ABL `RECID` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

ROWID

These are the data type-specific methods for adding an ABL `ROWID` parameter:

Syntax

```
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 General syntax for add parameter methods on page 184.

TABLE and TABLE-HANDLE

These are the data type-specific methods for adding an ABL temp-table (`TABLE` or `TABLE-HANDLE`) parameter:

Syntax

```
public void addTable
               (int position, java.sql.ResultSet value, int mode,
                com.progress.open4gl.ProResultSetMetaDataImpl metaData)
            throws Open4GLException
public void addTableHandle
               (int position, java.sql.ResultSet value, int mode,
                com.progress.open4gl.ProResultSetMetaDataImpl metaData)
            throws Open4GLException
public void addTable
               (int position,
                com.progress.open4gl.ProDataGraph value, int mode,
                com.progress.open4gl.ProDataGraphMetaData metaData)
            throws Open4GLException
public void addTableHandle
               (int position,
                com.progress.open4gl.ProDataGraph value, int mode,
                com.progress.open4gl.ProDataGraphMetaData metaData)
            throws Open4GLException
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 184.

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

```
public void addParameter
               (int position, Object value, int mode,
                int proType, int extentValue,
                com.progress.open4gl.ProResultSetMetaDataImpl metaData)
            throws Open4GLException
public void addParameter
               (int position, Object value, int mode,
                int proType, int extentValue,
                com.progress.open4gl.ProDataGraphMetaData metaData)
            throws Open4GLException
```

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.

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 the following table.

Table 33: ABL parameter data types for the OpenAPI

Progress data type	com.progress.open4gl.Parameterclass constant
CHARACTER	PRO_CHARACTER
COM-HANDLE	PRO_COMHANDLE
DATASET	PRO_DATASET
DATASET-HANDLE	PRO_ DATASETHANDLE
DATE	PRO_DATE
DATETIME	PRO_DATETIME
DATETIME-TZ	PRO_DATETIMETZ

Progress data type	com.progress.open4gl.Parameterclass constant
DECIMAL	PRO_DECIMAL
INT64	PRO_INT64
INTEGER	PRO_INTEGER
LOGICAL	PRO_LOGICAL
LONGCHAR	PRO_LONGCHAR
MEMPTR	PRO_MEMPTR
RAW	PRO_RAW
RECID	PRO_RECID
ROWID	PRO_ROWID
TABLE	PRO_TEMPTABLE
TABLE-HANDLE	PRO_TABLEHANDLE
WIDGET-HANDLE	PRO_WIDGETHANDLE

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 Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet on page 197.
- **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 Defining the schema for a ProDataSet parameter on page 199. For more information on defining the schema for

temp-table parameters mapped to a `ProDataGraph`, see Defining the schema for a temp-table parameter mapped to a ProDataGraph on page 205.

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

```
public void setReturnType(int proType) throws Open4GLException
```

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 the following table.

Note: User-defined functions cannot return `LONGCHAR` and `MEMPTR` data across an AppServer.

Table 34: User-defined function return types for the OpenAPI

ABL data type	com.progress.open4gl.Parameterclass constant
CHARACTER	PRO_CHARACTER
COM-HANDLE	PRO_COMHANDLE
DATE	PRO_DATE
DATETIME	PRO_DATETIME
DATETIME-TZ	PRO_DATETIMETZ
DECIMAL	PRO_DECIMAL
INT64	PRO_INT64
INTEGER	PRO_INTEGER
LOGICAL	PRO_LOGICAL
RAW	PRO_RAW
RECID	PRO_RECID

ABL data type	com.progress.open4gl.Parameterclass constant
ROWID	PRO_ROWID
WIDGET-HANDLE	PRO_WIDGETHANDLE

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:

```
parms.setReturnType(Parameter.PRO_INTEGER);
```

Arrays as return values

Java Open Clients can return array values (called extents in ABL) from user-defined functions.

The following table provides the data type mappings for array parameters.

Table 35: Data type mappings between ABL and Java array types

ABL type with EXTENT	Java proxy return typefor values	Java proxy return typefor unknown
INTEGER	int[ ]	Integer[]
INT64	long[]	Long[]
DECIMAL	java.math.BigDecimal[ ]	
CHARACTER	java.lang.String[ ]	
LOGICAL	boolean[ ]	Boolean[]
DATE DATETIME DATETIME-TZ	java.util.GregorianCalendar[ ]	
RECID	long[ ]	
RAW	byte[ ][ ]	
ROWID	com.progress.open4gl.Rowid[ ]	
COM-HANDLE	com.progress.open4gl.COMHandle[ ]	
WIDGET-HANDLE	com.progress.open4gl.Handle[ ]	

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

```
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

```
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:

```
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 the table in Arrays as return values on page 195. 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[]`.

```
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 an `java.sql.ResultSet` parameter you must access the parameter as a data stream. For more information on accessing temp-tables as `java.sqlResultSet` parameters, see Passing Temp-tables as SQL ResultSet Parameters on page 221

To define the schema for a temp-table parameter passed as a **`java.sql.ResultSet`**:

1. Define a `com.progress.open4gl.ProResultSetMetaDataImpl` object.
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 Defining the schema for a temp-table parameter mapped to a ProDataGraph on page 205.

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

```
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 Passing Parameters on page 43. 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 Passing Temp-tables as SQL ResultSet Parameters on page 221.

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 TABLE and TABLE-HANDLE on page 191.

The following example adds a temp-table as an input parameter.

Example: OpenAPI fragment adding a temp-table parameter as an SQL ResultSet

```
// Create the ParamArray
ParamArray parms = new ParamArray(1);

// Set up the meta data
ProResultSetMetaDataImpl metaData1;
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 Accessing ABL ProDataSets on page 53.

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

```
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 Defining the schema for temp-tables in a ProDataSet on page 202. You can then add the temp-table meta data (`ProDataObjectMetaData`) to the `ProDataGraphMetaData` object using the following `ProDataGraphMetaData` method:

Syntax

```
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

```
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.

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 DATASET and DATASET-HANDLE on page 186.

The following example adds a temp-table parameter defined with no indexes.

Example: OpenAPI fragment setting a ProDataSet parameter

```
// 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 Accessing ABL ProDataSets on page 53.

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 Accessing ABL ProDataSets on page 53.

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

```
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 Accessing ABL ProDataSets on page 53.

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 Passing Parameters on page 43. 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 Accessing ABL ProDataSets on page 53.

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

```
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: OpenAPI fragment adding a temp-table parameter

```
// 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);
// 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 Accessing ABL ProDataSets on page 53.

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 Accessing ABL ProDataSets on page 53.

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

```
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 Defining the schema for a temp-table parameter mapped to a java.sql.ResultSet on page 197.

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 `setTable()` 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 Defining the schema for a ProDataSet parameter on page 199.

For more information on working with temp-table parameters mapped to a `ProDataGraph`, see Accessing ABL ProDataSets on page 53.

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 Creating variables for parameters on page 183).
2. Add the parameter to a `ParamArray` object (see Creating a parameter array on page 183>).
3. Run the procedure or user-defined function (see Running procedures and user-defined functions on page 179).

This example passes an `INPUT integer` parameter that does not support the `Unknown value (?)`.

Example: Passing an INPUT parameter using the Java OpenAPI

```
// 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: Passing an INPUT parameter as the Unknown value (?) using the Java OpenAPI

```
// 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 Creating variables for parameters on page 183).
2. Add the parameter to a `ParamArray` object (see Creating a parameter array on page 183).

3. Run the procedure or user-defined function (see Running procedures and user-defined functions on page 179).
4. Get the output value from the `ParamArray` (see Getting OUTPUT parameter values on page 209). 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: Passing an INPUT-OUTPUT parameter using the Java OpenAPI

```
// 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 Creating variables for parameters on page 183).
2. Add the parameter to a `ParamArray` object (see Creating a parameter array on page 183).
3. Run the procedure or user-defined function (see Running procedures and user-defined functions on page 179).
4. Get the value from the `ParamArray` (see Getting OUTPUT parameter values on page 209).

This example to passes an `OUTPUT` integer parameter that does not support the `Unknown value (?)`.

Example: Passing an OUTPUT parameter using the Java OpenAPI

```
// 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

```
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: Getting an OUTPUT parameter using the Java OpenAPI

```
// 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

```
public String _getProcReturnString() throws Open4GLException
```

- **`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

  ```
  public String _getProcReturnString() throws Open4GLException
  ```

- **`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

  ```
  public String getProcReturnString()
  ```

For more information on using these objects to execute procedures, see Running procedures and user-defined functions on page 179.

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

```
public Object getReturnValue()
```

You need to cast the return `Object` to the correct return type for the function. For example:

```
String retVal = (String)(parms.getReturnValue());
```

For information on accessing arrays see Arrays as return values on page 195.

Sample Java OpenAPI code

The following sections describe two types of sample code:

- Non-persistent procedure example on page 211

- Persistent procedure example on page 212
- Single-run/singleton procedure example on page 213

Note: These samples are not available on the OpenEdge product ESD or the PSDN Web site.

Non-persistent procedure example

Suppose you want to run a non-persistent procedure, `AddCustomer.p`, with the following definition.

Non-persistent procedure ABL for the OpenAPI

```
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.

OpenAPI code to run the non-persistent procedure

```
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);

      // Set up input parameters
      parms.addCharacter(0, CustName, ParamArrayMode.INPUT);
      parms.addCharacter(1, phone, ParamArrayMode.INPUT);
      parms.addCharacter(2, email, ParamArrayMode.INPUT);

      // Set up Out parameters - notice the value is null
      parms.addInteger(3, null, ParamArrayMode.OUTPUT);

      // Run the procedure
      dynAO.runProc("AddCustomer.p", parms);
```

```
      // Get output parameters - Returned as Object, so must cast
      CustomerNumber = (Integer) parms.getOutputParameter(3);

      // Get RETURN-VALUE - Will return null for AddCustomer() procedure
      retVal = (String)(parms.getProcReturnString());
      dynAO._release();

    } // try to catch all unexpected exceptions
    catch ( Exception e )
    {
        System.out.println("Exception in sample1()");
        System.out.println("Exception Message: " + e.getMessage());
        e.printStackTrace();
    }
  }
} // class
```

Persistent procedure example

Suppose you want to run a persistent procedure, `CustomerOrder.p`, with the following definition, containing a user-defined function.

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.

OpenAPI code to run the persistent procedure

```
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
```

Single-run/singleton procedure example

A procedure called `CustomerInfo.p` contains a user-defined function with the following definition.

User-defined function defined by CustomerInfo.p

```
FUNCTION GetTotalOrdersByNumber RETURNS INTEGER (CustNum AS INTEGER, threshold AS
DECIMAL):
...
END.
```

The version of `CustomerInfo.p` in the following code example (Persistent procedure ABL for the OpenAPI)cannot be run as a single-run (or singleton) because it has a parameter in the main block. Creating another internal procedure to set `CustNum` would also not work, because a single-run procedure is deleted from the AppServer after each call. Context must be established for each call. So in the previous code example (User-defined function defined by CustomerInfo.p), `CustNum` is passed as a parameter to the user-defined function `GetTotalOrdersByNumber` instead.

The OpenAPI code, in the following code example, runs `CustomerInfo.p` as a single-run procedure and then calls its user-defined function. The code also shows how to access the procedure type using the `getProcedureType()` method.

OpenAPI code to run the single-run procedure

```
import com.progress.open4gl.javaproxy.*;
import com.progress.open4gl.Parameter;
import com.progress.open4gl.ProcedureType;

public class sampleClient
{  private static final java.lang.String tName = new String( "KM_client" );

   public static void singleRunProcedure()
   {
    try  // To catch all exceptions
    {
      // Connect to the AppServer
      Connection myConn = new Connection("","","");
      OpenAppObject dynAO = new OpenAppObject(myConn, "asbroker2");

      // Run procedure as single-run
      OpenProcObject dynPO = dynAO.createPO("CustomerInfo.p",
                                            ProcedureType.SINGLE_RUN);

      // Display procedure type using getProcedureType method
      System.out.println("Procedure object procedure type: "
                         + dynPO.getProcedureType());
      // Call UDF
      // First set up parameters
      int custNum = 3;
      java.math.BigDecimal threshold = new java.math.BigDecimal(1000);
      Integer retVal;

      // Set up input parameters
      ParamArray parms = new ParamArray(2);
      parms.addInteger(0, custNum, ParamArrayMode.INPUT);
      parms.addDecimal(1, threshold, ParamArrayMode.INPUT);

      // Set up return type
      parms.setReturnType(Parameter.PRO_INTEGER);

      // Run UDF
      dynPO.runProc("GetTotalOrdersByNumber", parms);

      // Get return value
      retVal = (Integer)(parms.getReturnValue());

      // Even though the single-run procedure has been deleted on the
      // AppServer, use _release() to clean up the resources on the client.
      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
```

A singleton procedure can be run in the same way, with `ProcedureType.SINGLETON` substituted into the `createPO` method. Unlike a single-run procedure, a singleton procedure remains instantiated on the AppServer after execution, but as with a single-run procedure, you need to use `_release()` to clean up resources on the client side.

A

Accessing a SmartDataObject API Directly

This chapter describes how to access a SmartDataObject by defining a ProcObject that supports the interface.

For details, see the following topics:

- Introduction
- Directly accessing the SmartDataObject API in a proxy
- Defining a ProcObject that maps a SmartDataObject

Introduction

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 by defining a ProcObject 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.

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*.

B

Java Open Client Certificate Management Utility

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.

For details, see the following topics:

- 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_certout_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*.

C

Passing Temp-tables as SQL ResultSet Parameters

This section describes how to pass temp-tables as SQL `ResultSet` parameters.

For details, see the following topics:

- Introduction
- 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

Introduction

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 Passing Parameters on page 43

Note: You can also map and pass temp-table parameters as `ProDataGraph` objects. For more information, see Accessing ABL ProDataSets on page 53

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 Passing a TABLE or TABLE-HANDLE as an INPUT parameter on page 228.

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 Accessing TABLE or TABLE-HANDLE parameters as OUTPUT on page 233.

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. The following table shows the mappings.

Table 36: SQL ResultSet data type mappings for temp-table fields

Progress data type	Java data type

`BLOB`	`java.sql.Blob`
	Or
	`byte [ ]`
	Note: 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.

Progress data type	Java data type
CHARACTER	java.lang.String
CLOB	java.sql.Clob Or java.lang.String **Note:** 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.
COM-HANDLE	Long
DATE	java.sql.DateOrjava.util.GregorianCalendar **Note:** 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.
DATETIME	java.sql.Timestamp Or java.util.GregorianCalendar **Note:** 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.
DATETIME-TZ	java.util.GregorianCalendar
DECIMAL	java.math.BigDecimal
INT64	long
INTEGER	Integer
LOGICAL	Boolean
RAW	byte[]

Progress data type	Java data type
RECID	Long
ROWID	byte[]
WIDGET-HANDLE	Long

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 Accessing TABLE or TABLE-HANDLE parameters as OUTPUT on page 233.

The following examples use the Java flat model meta data methods `getColumnType()` and `getColumnCount()` and the array model methods `getFieldProType()` 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.

The following table shows how the call-back sequence works when passing an `INPUT TABLE` or `TABLE-HANDLE` parameter.

Table 37: Passing an INPUT temp-table as an SQL ResultSet

Client application	Proxy
1. Creates input `ResultSet` object **2.** Passes `ResultSet` object as a parameter to proxy	–
–	**3.** Uses methods on the input `ResultSet` object **4.** Passes input data to the AppServer

The following table shows how the call-back sequence works when passing an `OUTPUT TABLE` or `TABLE-HANDLE` parameter.

Table 38: Passing an OUTPUT temp-table as an SQL ResultSet

Client application	Proxy
1.Creates holder object for the output `ResultSet` **2.** Passes holder object to proxy	–
–	**3.** Gets output data from the AppServer **4.** Creates output `ResultSet` object and stores it in holder object **5.** Returns to client
6. Uses methods on the `ResultSet` contained in the holder object	–

The following table shows how the call-back sequence works when passing an `INPUT-OUTPUT TABLE` or `TABLE-HANDLE` parameter.

Table 39: Passing an INPUT-OUTPUT temp-table as an SQL ResultSet

Client Application	Proxy
1. Creates holder object **2.** Creates input `ResultSet` object and stores it in holder **3.** Passes holder object to proxy	–
–	**4.** Uses methods on the ResultSet in the holder object to get data **5.** Passes input data to the AppServer **6.** Gets output data from the AppServer **7.** Creates output `ResultSet` object and stores it in holder object **8.** Returns to client
9. Uses methods on the `ResultSet` in the holder object to get data	–

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

```
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

```
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

```
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:

Syntax

```
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 the following table.

Note: Because a standard `ResultSet` supports a standard `ResultSetMetaData` object and not the ABL extension (`ProResultSetMetaData`), there can be no array fields.

Table 40: Mapping between SQL and ABL schema data types for standard JDBC ResultSet

SQL type	ABL type
BIGINT	INT64
BIT	LOGICAL
BLOB	BLOB
CLOB	CLOB
DATE	DATE
DECIMAL	DECIMAL
INTEGER	INTEGER
LONGVARCHAR	CHARACTER
TIMESTAMP	DATETIME
VARBINARY	RAW

  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 the previous table 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 ProResultSetMetaDataImpl class on page 231.

- 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 ProResultSetMetaDataImpl class on page 231.

- 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.

There are only two methods of `ProResultSetMetaDataImpl` that the application must call:

1. The constructor:

   Syntax

   ```
   ProResultSetMetaDataImpl(int numFields)
   ```

2. The `setFieldMetaData()` method:

   Syntax

   ```
   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 Passing Parameters on page 43.

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 on page 232.
3. Create an instance of com.progress.open4gl.ProResultSetMetaDataImpl.
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 setMetaData() method to your InputResultSet implementation.
6. Make the proxy method call passing the com.progress.open4gl.InputResultSet instance as the parameter.

Note: Step 3 on page 232 and Step 4 on page 232 can be implemented in getMetaData() which eliminates the need for Step 5 on page 232.

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 Accessing TABLE or TABLE-HANDLE parameters as OUTPUT on page 233.

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

```
boolean next()
```

The following method closes the `ResultSet`:

Syntax

```
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 `columnNum`:

Syntax

```
Object getObject(int columnNum)
```

The data type of the returned object is the default data type of the column. For example, the default data type for an ABL `CHARACTER` field is `java.lang.String`. For more information, see Data type mapping for temp-table fields on page 222.

The following method returns the value of the column identified by the `fieldName`:

Syntax

```
Object getObject(String fieldName)
```

The data type of the returned object is the default data type of the field. For example, the default data type for an ABL `CHARACTER` field is `java.lang.String`. For more information, see Data type mapping for temp-table fields on page 222.

The following method returns the value of the column identified by the 1-based `columnNum`:

Syntax

```
DataType getDataType(int columnNum)
```

`DataType` is the name of the data type, for example, `String getString()`, `Integer getInteger()`, and so on.

The following method returns the value of the column identified by the `fieldName`:

Syntax

```
DataType getDataType(String fieldName)
```

`DataType` is the name of the data type; for example, `String getString()`, `Integer getInteger()`, and so on.

If the specified return *`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 *`DataType`*
- An exception is thrown with a message explaining that the conversion is not supported

For example, if the ABL data type is `INTEGER` and its value is 10, `getString(columnNum)` automatically converts the value from `Integer` to the `java.lang.String`, "10". If the application calls `getDate(columnNum)`, an exception is thrown with the message that an `Integer` value cannot be converted to `Date`. For information on the supported conversions, see Providing metadata for INPUT TABLE-HANDLE parameters on page 230.

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 Accessing temp-table array fields on page 224).

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)
```

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

```
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 *fieldName* are 1-based.

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:

```
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 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 `get`*DataType*`()` method you use to get the value of a temp-table field does not match the default Java data type of the field (see Preparing and managing temp-table parameters on page 222), a valid conversion occurs as long as the `get`*DataType*`()` method corresponds to the ABL data type, as shown in the following table. 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 41: ProResultSet type conversions for temp-table fields

ABL data type	Valid Java data type accessor methods
BLOB	byte[] getBytes() java.sql.Blob getBlob()
CHARACTER	java.lang.String getString()
CLOB	java.lang.String getString() java.sql.Clob getClob()
COM-HANDLE	java.lang.String getString() int getInt() long getLong()
DATE	java.lang.String getString() java.sql.Date getDate() java.util.GregorianCalendar getGregorianCalendar()
DATETIME	java.lang.String getString() java.util.GregorianCalendar getGregorianCalendar() java.sql.Timestamp getTimeStamp() java.util.GregorianCalendar getGregorianCalendar()
DATETIME-TZ	java.lang.String getString() java.util.GregorianCalendar getGregorianCalendar()
DECIMAL	java.lang.String getString() int getInt() long getLong() double getDouble() java.math.BigDecimal getBigDecimal()
INT64	java.lang.String getString() int getInt() long getLong() double getDouble() java.math.BigDecimal getBigDecimal()

ABL data type	Valid Java data type accessor methods
INTEGER	java.lang.String getString() int getInt() long getLong() double getDouble() java.math.BigDecimal getBigDecimal()
LOGICAL	java.lang.String getString() int getInt() boolean getBoolean()
RAW	byte[] getBytes()
RECID	java.lang.String getString() int getInt() long getLong()
ROWID	byte[] getBytes()
WIDGET-HANDLE	java.lang.String getString() int getInt() long getLong()

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:

```
// 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.

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`:

```
int getColumnCount() throws ProSQLException
```

```
String getColumnName(int columnNum) throws ProSQLException
```

```
int getColumnType(int columnNum) throws ProSQLException
```

The `getColumnType()` method returns an `int` constant for the standard SQL types as shown in the following table. The **SQL data type** column in the table shows the type names as specified by the `java.sql.Types` class. For example:

```
String getColumnTypeName(int columnNum) throws ProSQLException
```

The `getColumnTypeName()` method returns the corresponding ABL data type names shown in the following table under **ABL data type**.

Table 42: ABL to SQL data type mappings

ABL data type	SQL data type
BLOB	BLOB
CHARACTER	LONGVARCHAR
CLOB	CLOB
COM-HANDLE	BIGINT
DATE	DATE
DATETIME DATETIME-TZ	TIMESTAMP
DECIMAL	DECIMAL
INT64	BIGINT

ABL data type	SQL data type
INTEGER	INTEGER
LOGICAL	BIT
RAW	VARBINARY
RECID	BIGINT
ROWID	VARBINARY
WIDGET-HANDLE	BIGINT

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:

```
int getColumnProType(int columnNum) throws ProSQLException
```

  The following returns one of the constant values accessible from `com.progress.open4gl.Parameter`:

```
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 Preparing and managing temp-table parameters on page 222.

  For more information, see the section on specifying data type meta data for temp-tables in Passing Parameters on page 43

- The following methods provide access to the schema of temp-table parameters that contain array fields, viewed from an ABL viewpoint:

```
int getFieldCount() throws ProSQLException
```

```
String getFieldName(int fieldNum) throws ProSQLException
```

```
int getFieldExtent(int fieldNum) throws ProSQLException
```

```
int getFieldProType(int fieldNum) throws ProSQLException
```

```
String getFieldJavaTypeName(int fieldNum) throws ProSQLException
```

```
String getFieldTypeName(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 Accessing temp-table array fields on page 224.

Although you can access the data using either the SQL (flat) or ABL (array) model, both provide equivalent functionality. The following table shows the correspondence between the `ProResultSetMetaData` methods.

Table 43: Comparing some array and flat model methods

This array model methodreturns the same set of values . . .	As this flat model method . . .
`getFieldName()`	`getColumnName()`
`getFieldProType()`	`getColumnProType()`
`getFieldJavaTypeName()`	`getColumnJavaTypeName()`
`getFieldTypeName()`	`getColumnTypeName()`

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 Passing a TABLE or TABLE-HANDLE as an INPUT parameter on page 228. For more information on ProResultSet objects, see Accessing TABLE or TABLE-HANDLE parameters as OUTPUT on page 233.

Example Java client passing an SQL ResultSet parameter

The following procedure supports a small portion of an accounting system. The AccountInfo.p procedure is expected to run persistently and has an internal procedure called getPaymentsInfo:

Note: These samples are not available on the OpenEdge product ESD or the PSDN Web site.

AccountInfo.p

```
/* 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 the following example demonstrates how to create and use AppObjects and ProcObjects from a Java client. In this example, the AppObject was defined with ProxyGen as Account.

```
    // 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();
```

The following code example shows the `PayeeList` class for the Java client using proxy objects example. This class implements the passing of a temp-table parameter as an SQL ResultSet.

```
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 *23*
ABL unknown value, See Unknown value (?)
absolute( ) method
  SDOResultSet *162*, *169*
acceptChanges( ) method
  ProDataGraph *80*
Accessing
  ABL arrays in SQL ResultSet parameters *224*
  ProChangeSummary instances *74*
  ProDataObject instances
    using known temp-table name *64*
    using related table rows *76*
  Progress ProDataSets *54*
  proxy properties
    command line *120*
    Connection class *120*
    methods *121*
    RunTimeProperties class *120*
Accessor methods
  general property
    Connection class *122*
    RunTimeProperties class *121*
  purposed property *124*
addDataRelation( ) method
  ProDataGraphMetaData *99*
addParameter( ) method *191*
addProDataObject( ) method
  ProDataGraph *80*
addTable( ) method
  ProDataGraphMetaData *99*
Application services
  ABL methods *23*
  proxy objects *22*
AppObjects
  constructors *38*
  example *29*
  generation *22*
  instantiating *38*
  session-free method execution *28*
  SmartDataObject access *152*
AppServer
  establishing a connection *38*
  operating modes *40*
AppServer connection procedure *39*
Array model for SQL ResultSets
  array fields in temp-tables *224*
  behavior *225*
  methods compared to flat model *239*
  ProResultSet extended methods *234*
Array model for SQL ResultSets *(continued)*
  ProResultSetMetaData extended methods *239*
Arrays
  parameter mappings *45*
  Unknown value (?) *46*
attachToAppObj ( ) method
  SDOResultSet *161*

B

beforeFirst( ) method
  SDOResultSet *162*
beginLogging( ) method
  ProChangeSummary *96*
boolean wasNull( ) method
  SDOResultSet *169*
Building Java client applications *18*

C

Call-back sequence SQL ResultSets *225*
_cancelAllRequests( ) method *25*
cancelBatch( ) method
  SDOResultSet *167*
cancelRowUpdates( ) method
  SDOResultSet *166*
Certificate store files *217*
certificateStore property *133*
Change tracking
  ProDataGraph *57*
ChangeSummary interface
See also ProChangeSummary class
  Java SDO features *55*
  See also ProChangeSummary class
Class factory methods *24*
clear( ) method *183*
Client principal class
  attributes *35*
  methods *36*
Client programming
  ABL unknown values *45*
  connecting to an AppServer *33*
  exception handling *143*
  ProDataSet parameters *54*
  temp-table parameters
    ProDataGraphs *77*
    SQL ResultSets *221*
clientPrincipal property *134*
close( ) method
  ProResultSet *233*
  SDOResultSet *169*

Closing OUTPUT SQL ResultSets 227
Column properties 55
Common proxy methods 25
Compression
  Compression Enabled property 140
  Compression Level property 140
  Compression methods 129
  Compression Threshold property 140
Connecting to an AppServer
  using proxy objects 33
  using the OpenAPI 178
Connection
  class
    client principal property 33
    definition 120
    general accessor methods 122
    modifying object properties 131
    relationship to RunTimeProperties class 130
    releasing objects 40
    session-free AppObjects 29
  methods 23, 33
  pool
    session-free AppObject sharing 29
connection procedure 39
connectionLifetime property 134
Constructors
  AppObject 38
  example classes extending proxy objects 115
  Holder classes 46
  Java OpenAPI
    ProDataGraph 201
    ProDataGraphMetaData 199
    ProDataObjectMetaData 203
    ProDataRelationMetaData 200
    ProResultSetMetaDataImpl 197
  OpenAppObject 178
  ParamArray 183
  ProDataGraph 78
  ProDataGraphMetaData 98
  ProDataObjectMetaData 101
  ProDataRelationMetaData 107
  ProResultSetMetaDataImpl 231
  SDOAppObject 153
  SDOResultSet objects 154
CreatePO_ProcObject( ) method 24
createPO( ) method
  OpenAppObject 180
createProDataObject( ) method
  ProDataGraph 80
_CreateSDOResultSet ( ) 24
Creating
  ProDataObject instances 77

D

Data type mappings
  ABL to Java proxy 43
  ABL unknown values 45
  INPUT parameters 43
  OUTPUT and INPUT-OUTPUT parameters 46
  Progress to SQL ResultSet (OUTPUT) 238
  SQL ResultSet to Progress (INPUT) 230
  temp-table field defaults
    ProDataObject 62
    SQL ResultSet 222
Data-relations
  ProDataGraph 57
  ProDataSet 54
DataGraph interface
See also ProDataGraph class
  Java SDO features 55
  See also ProDataGraph class
DataObject interface
See also ProDataObject class
  Java SDO features 55
  property meta data 111
  See also ProDataObject class
Date conversions in parameters 45
deleteRow( ) method
  SDOResultSet 166
detachFromAppObj ( ) method
  SDOResultSet 161
Digital certificates
  managing 217

E

enableTracing property 134
endLogging( ) method
  ProChangeSummary 96
Error handling, proxies 143
  See also Exceptions
Examples
  ABL procedures mapped to Java proxy methods 23
  Java OpenAPI code
    non-persistent procedure 211
    persistent procedure 212
Exceptions
  descriptions 144
  general ABL
    class hierarchy 144
    definition 143
  output ResultSet 143, 147

F

findColumn( ) method
  SDOResultSet 169

first( ) method
SDOResultSet *162*
Flat model for SQL ResultSets
array fields in temp-tables *224*
behavior *224*
methods compared to array model *239*
ProResultSet standard methods *233*
ProResultSetMetaData standard methods *238*

G

Generating proxies *17*
get( ) method
ProDataObject *84*
getBigDecimal( ) method
ProDataObject *84*
getBImageFlag( ) method
ProDataObjectMetaData *103*
getBoolean( ) method
ProDataObject *85*
getBytes( ) method
ProDataObject *85*
getChangedDataObjects( ) method
ProChangeSummary *96*
getChanges( ) method
ProChangeSummary *98*
getChangeSummary( ) method
ProDataGraph *79*
getChildColumns( ) method
ProDataRelationMetaData *109*
getChildRows( ) method
ProDataObject *92*
getChildTable( ) method
ProDataRelationMetaData *109*
getClass( ) method
Type *111*
getColumnCount( ) method
SDOResultSetMetaData *172*
getColumnDisplaySize( ) method
SDOResultSetMetaData *172*
getColumnJavaTypeName( ) method
ProResultSetMetaData *239*
getColumnLabel( ) method
SDOResultSetMetaData *172*
getColumnName( ) method
SDOResultSetMetaData *172*
getColumnProType( ) method
ProResultSetMetaData *239*
getColumnType( ) method
SDOResultSetMetaData *172*
getColumnTypeName( ) method
SDOResultSetMetaData *172*
_getConnectionId( ) method *26*
getDataType( ) method
ProResultSet *233–234*
getDefault( ) method
Property *113*
getExtent( ) method
ProDataObjectMetaData *104*
getFetchSize( ) method
SDOParameters *156*
SDOResultSet *169*
getFieldCount( ) method
ProDataObject *93*
ProDataObjectMetaData *104*
ProResultSetMetaData interface *239*
getFieldExtent( ) method
ProResultSetMetaData *239*
getFieldJavaTypeName( ) method
ProResultSetMetaData *239*
getFieldName( ) method
ProDataObjectMetaData *104*
ProResultSetMetaData *239*
getFieldProType( ) method
ProResultSetMetaData *239*
getFields( ) method
SDOModificationException *174*
getFieldTypeName( ) method
ProResultSetMetaData *239*
getGregorianCalendar( ) method
ProDataObject *93*
getInstanceClass( ) method
Type *111*
getInt( ) method
ProDataObject *86*
getList( ) method
ProDataObject *86*
getLong( ) method
ProDataObject *87*
getMetaData( ) method
ProDataGraph *81*
SDOResultSetMetaData *169*
SQL ResultSet *227*
getName( ) method
Property *113*
getNoSchemaMarshal( ) method
InputResultSet *228*
ProDataObjectMetaData *104*
getNumRelations( ) method
ProDataGraphMetaData *99*
getNumTables( ) method
ProDataGraph *81*
ProDataGraphMetaData *100*
getObject( ) method
InputResultSet *228*
ProResultSet *233–234*
getOldValues( ) method
ProChangeSummary *97*
getOutputParameter( ) method
ParamArray *209*

getParentColumns( ) method
ProDataRelationMetaData *109*
getParentRow( ) method
ProDataObject *94*
getParentTable( ) method
ProDataRelationMetaData *110*
getPrefetchMaxRows( ) method
SDOParameters *156*
getProChangeSummary( ) method
ProDataGraph *81*
getProcReturnString( ) method
ParamArray *209*
_getProcReturnString( ) method
common proxy object *27*
OpenAppObject *209*
OpenProcObject *209*
getProDataGraphName( ) method
ProDataGraph *82*
getProDataObjects( ) method
ProDataGraph *81*
getProperties( ) method
Type *111*
getProperty( ) method
Type *111*
getProType( ) method
ProDataObjectMetaData *105*
getQuery( ) method
SDOResultSet *169*
getRelationMetaData( ) method
ProDataGraphMetaData *100*
getRelationName( ) method
ProDataRelationMetaData *110*
_getRequestId( ) method *27*
getReturnValue( ) method
ParamArray *210*
getRootObject( ) method
ProDataGraph *79*
getRow( ) method
SDOResultSet *162*
getRowErrorString( ) method
ProDataObject *94*
getRowIdentity( ) method
SDOParameters *156*
SDOResultSet *169*
getScrollingMode( ) method
SDOParameters *156*
getSDOInterface( ) method
SDOResultSet *169*
_getSSLSubjectName( ) method *28*
_getProcedureType( ) method *28*
getStateless( ) method
SDOParameters *156*
getString( ) method
ProDataObject *87*
getTable( ) method
SDOModificationException *174*
getTableErrorString( ) method
ProDataObject *94*
getTableMetaData( ) method
ProDataGraphMetaData *100*
getTableName( ) method
ProDataObject *94*
ProDataObjectMetaData *105*
SDOResultSetMetaData *172*
getTableNames( ) method
ProDataGraph *82*
ProDataGraphMetaData *100*
getText( ) method
SDOModificationException *174*
getType( ) method
ProDataObject *88*
Property *113*
getUserOrder( ) method
ProDataObjectMetaData *105*

H

hasError( ) method
ProDataGraph *82*
hasRowError( ) method
ProDataObject *95*
hasTableError( ) method
ProDataObject *95*
Holder classes
constructors *46*
super class definition *46*
type-specific
array mappings *46*
definitions *49*
mappings *46*
ProDataGraph *61*
HTTPS *217*
See also Secure Sockets Layer (SSL)

I

IDLE state
connection *41*
idleConnections property *135*
idleConnectionTimeout property *135*
Implementation classes *115*
inBatch( ) method
SDOResultSet *167*
INPUT parameters
Java OpenAPI *206*
ProDataGraph
adding data *71*
initializing meta data *69*
preparing *68*
INPUT SQL ResultSets *227*
INPUT-OUTPUT parameters
Java OpenAPI *207*

INPUT-OUTPUT parameters *(continued)*
ProDataGraph
adding data *71*
initializing meta data *69*
preparing *68*
InputResultSet class
definition *222*
getObject( ) method *228*
next( ) method *228*
using *228*
insertRow( ) method
SDOResultSet *166*
isAfterLast( ) method
SDOResultSet *162*
isAttached( ) method
SDOResultSet *161*
isBeforeFirst( ) method
SDOResultSet *162*
isCreated( ) method
ProChangeSummary *97*
isDeleted( ) method
ProChangeSummary *97*
isFirst( ) method
SDOResultSet *162*
isInstance( ) method
Type *111*
isMany( ) method
Property *113*
isNullable( ) method
SDOResultSetMetaData *172*
isReadOnly( ) method
SDOResultSetMetaData *172*
isSet( ) method
ProDataObject *88*
_isStreaming( ) method *26*
isWritable( ) method
SDOResultSetMetaData *172*

J

Java SDO, See Java Service Data Objects (Java SDO)
Java Service Data Objects (Java SDO)
See also DataGraph interface
Data Mediator Services *55*
DataGraph interface *55*
documentation *55*
ProDataGraph extensions *57*
property meta data *111*
reference implementation *55*
See also DataGraph interface
JSR 235
documentation *55*
overview *55*

L

last( ) method
SDOResultSet *162*
List of files in the certificate store *217*
logFileName property *135*
loggingLevel property *135*

M

Managing digital certificates *217*
maxConnections property *135*
Meta data
Java Service Data Objects (SDO)
underlying objects *111*
ProDataGraphs
accessing output parameters with known schema *64*
accessing output parameters with unknown schema *66*
initializing input parameters *69*
SmartDataObjects (SDOResultSets) *171*
SQL ResultSets
INPUT TABLE-HANDLE *230*
OUTPUT TABLE and TABLE-HANDLE *237*
temp-table ABL field data types *51*
Methods
absolute( ) *162*, *169*
acceptChanges( ) *80*
addDataRelation( ) *99*
addProDataObject( ) *80*
addTable( ) *99*
attachToAppObj ( ) *161*
beforeFirst( ) *162*
beginLogging( ) *96*
boolean wasNull( ) *169*
cancelBatch( ) *167*
cancelRowUpdates( ) *166*
class factory *24*
clear( ) *183*
close( )
ProResultSet *233*
SDOResultSet *169*
connection *23*
CreatePO_ProcObject( ) *24*
createPO( )
OpenAppObject *180*
createProDataObject( ) *80*
deleteRow( ) *166*
detachFromAppObj ( ) *161*
endLogging( ) *96*
findColumn( ) *169*
first( ) *162*
get( ) *84*
getBigDecimal( ) *84*
getBImageFlag( ) *103*

Methods *(continued)*
getBoolean( ) *85*
getBytes( ) *85*
getChangedDataObjects( ) *96*
getChanges( ) *98*
getChangeSummary( ) *79*
getChildColumns( ) *109*
getChildRows( ) *92*
getChildTable( ) *109*
getClass( ) *111*
getColumnCount( ) *172*
getColumnDisplaySize( ) *172*
getColumnJavaTypeName( ) *239*
getColumnLabel( ) *172*
getColumnName( ) *172*
getColumnProType( ) *239*
getColumnType( ) *172*
getColumnTypeName( ) *172*
getDataType( ) *233–234*
getDefault( ) *113*
getExtent( ) *104*
getFetchSize( )
SDOParameters *156*
SDOResultSet *169*
getFieldCount( )
ProDataObject *93*
ProDataObjectMetaData *104*
ProResultSetMetaData *239*
getFieldExtent( ) *239*
getFieldJavaTypeName( ) *239*
getFieldName( )
ProDataObjectMetaData *104*
ProResultSetMetaData *239*
getFieldProType( ) *239*
getFields( ) *174*
getFieldTypeName( ) *239*
getGregorianCalendar( ) *93*
getInstanceClass( ) *111*
getInt( ) *86*
getList( ) *86*
getLong( ) *87*
getMetaData( )
ProDataGraph *81*
SDOResultSetMetaData *169*
SQL ResultSet *227*
getName( ) *113*
getNoSchemaMarshal( )
InputResultSet *228*
getNumRelations( ) *99*
getNumTables( )
ProDataGraph *81*
ProDataGraphMetaData *100*
getObject( )
InputResultSet *228*
ProResultSet *233–234*
getOldValues( ) *97*

Methods *(continued)*
getOutputParameter( ) *209*
getParentColumns( ) *109*
getParentRow( ) *94*
getParentTable( ) *110*
getPrefetchMaxRows( ) *156*
getProChangeSummary( ) *81*
_getProcReturnString( )
common proxy object *27*
OpenAppObject *209*
OpenProcObject *209*
getProcReturnString( )
ParamArray *209*
getProDataGraphName( ) *82*
getProDataObjects( ) *81*
getProperties( ) *111*
getProperty( ) *111*
getProType( ) *105*
getQuery(*169*
getRelationMetaData( ) *100*
getRelationName( ) *110*
getReturnValue( ) *210*
getRootObject( ) *79*
getRow( ) *162*
getRowErrorString( ) *94*
getRowIdentity( )
SDOParameters *156*
SDOResultSet *169*
getScrollingMode( ) *156*
getSDOInterface( ) *169*
getStateless( ) *156*
getString( ) *87*
getTable( ) *174*
getTableErrorString( ) *94*
getTableMetaData( ) *100*
getTableName( )
ProDataObject *94*
ProDataObjectMetaData *105*
SDOResultSetMetaData *172*
getTableNames( )
ProDataGraph *82*
ProDataGraphMetaData *100*
getText( ) *174*
getType( )
ProDataObject *88*
Property *113*
getUserOrder( ) *105*
hasError( ) *82*
hasRowError( ) *95*
hasTableError( ) *95*
inBatch( ) *167*
insertRow( ) *166*
isAfterLast( ) *162*
isAttached( ) *161*
isBeforeFirst( ) *162*
isCreated( ) *97*

Methods *(continued)*
isDeleted( ) *97*
isFirst( ) *162*
isInstance( ) *111*
isMany( ) *113*
isNullable( ) *172*
isReadOnly( ) *172*
isSet( ) *88*
isWritable( ) *172*
Java OpenAPI
data type-specific add parameter *184*
generic add parameter *191*
cancelAllRequests( ) *25*
_CreateSDOResultSet ( ) *24*
_getConnectionId( ) *26*
_getProcedureType( ) *28*
_getRequestId( ) *27*
_getSSLSubjectName( ) *28*
_isStreaming( ) *26*
_release( ) *27*
moveToCurrentRow( ) *166*
moveToInsertRow( ) *166*
next( )
InputResultSet *228*
ProResultSet *233*
SDOResultSet *162*
nextMessage( ) *174*
proxy object
common *25*
Secure Sockets Layer property accessor *127*
server property accessor *126*
session property accessor *124*
thread control property accessor *126*
tracing property accessor *128*
relative( ) *162*
releaseConnection( ) *40*
releaseSDOInterface( ) *169*
remote ABL *23*
reOpenQuery( )
AppServer connections *161*
query management *169*
row fetch *159*
rowDeleted( ) *166*
rowInserted( ) *166*
rowUpdated( ) *166*
running on session-free AppObjects *28*
runProc( )
OpenAppObject *180*
OpenProcObject *182*
sendBatch( ) *161*
sendBatchAndReOpen( )
AppServer connections *161*
batch mode *167*
set( ) *88*
setBigDecimal( ) *89*
setBoolean( ) *89*

Methods *(continued)*
setBytes( ) *90*
setChildTableReferences( ) *83*
setColumns( ) *110*
setFetchSize( ) *156*
setFieldMetaData( )
ProDataObjectMetaData *106*
ProResultSetMetaDataImpl *231*
setGregorianCalendar( ) *95*
setInt( ) *90*
setList( ) *91*
setLong( ) *91*
setNoSchemaMarshal( )
InputResultSet *228*
setPrefetchMaxRows( )
SDOParameters *156*
SDOResultSet stateless mode *159*
setRowIdentity( ) *156*
setScrollingMode( ) *156*
setStateless( ) *156*
setString( ) *92*
startBatch( ) *167*
updateRow( ) *166*
minConnections property *136*
moveToCurrentRow( ) method
SDOResultSet *166*
moveToInsertRow( ) method
SDOResultSet *166*

N

next( ) method
InputResultSet *228*
ProResultSet *233*
SDOResultSet *162*
nextMessage( ) method
SDOModificationException *174*
noHostVerify property *136*
noSslSessionReuse property *136*
nsClientMaxPort property *136*
nsClientMinPort property *137*
nsClientPickListExpiration property *137*
nsClientPicklistSize property *137*
nsClientPortRetry property *137*
nsClientPortRetryInterval property *137*

O

Open Client Runtime package
supported security *15*
Open Client Runtime packages
supported security *15*
Open4GLException class *144*
OpenAPI
AppServer connections *178*
classes summary *177*

OpenAPI *(continued)*
parameters
adding with data type-specific methods *184*
adding with generic methods *191*
CHARACTER methods *186*
COM-HANDLE methods *186*
creating variables *183*
DATASET and DATASET-HANDLE methods *186*
DATE methods *187*
DATETIME methods *187*
DATETIME-TZ methods *187*
DECIMAL methods *188*
defining ProDataSet data-relations *200*
defining ProDataSet temp-tables *200*
HANDLE methods *188*
INPUT-OUTPUT, passing *207*
INPUT, passing *206*
INTEGER methods *188–189*
LOGICAL methods *189*
LONGCHAR methods *189*
MEMPTR methods *190*
OUTPUT, passing *208*
OUTPUT, value *209*
parameter array *183*
ProDataSet schema *199*
ProDataSet temp-table schema *202*
RAW methods *190*
RECID methods *190*
ROWID methods *191*
TABLE and TABLE-HANDLE methods *191*
temp-table instance (SQL ResultSet) *198*
temp-table schema (SQL ResultSet) *197*
temp-table schema and instance (ProDataGraph) *205*
UDF return types *194*
returned values
OUTPUT parameters *209*
RETURN-VALUE function from procedures *209*
user-defined function *210*
running
application services *179*
internal procedures and user-defined functions *182*
non-persistent procedures *180*
persistent procedures *180*
user-defined function return types *194*
OpenAppObject class
accessing ABL procedure return values *209*
constructors *178*
method parameters, See OpenAPI parameters
running
application services *179*
non-persistent procedures *180*
persistent procedures *180*
OpenEdge Reference Architecture (OERA)
preparing a ProDataGraph parameter *68*
ProDataSet parameter passing mode *74*
OpenProcObject class
accessing ABL procedure return values *209*
creating instances *180*
method parameters, See OpenAPI parameters
running internal procedures and UDFs *182*
OUTPUT parameters
Java OpenAPI *208*
ProDataGraph
accessing with a known schema *64*
accessing with an unknown schema *66*
OUTPUT SQL ResultSets
closing *227*
order of access *227*

P

ParamArray class
accessing ABL procedure return values *209*
constructor *183*
ParamArrayMode class
constants passed to data type-specific methods *184*
constants passed to generic methods *191*
Parameter class *51*
Parameters
date and time conversions *45*
holder classes *46*
INPUT mappings *43*
Java OpenAPI
adding with data type-specific methods *184*
adding with generic methods *191*
INPUT-OUTPUT, passing *207*
INPUT, passing *206*
OUTPUT, passing *208*
OUTPUT and INPUT-OUTPUT array mappings *46*
OUTPUT and INPUT-OUTPUT mappings *46*
passing *43*
Progress array mappings *45*
temp-tables and ProDataSets
ABL mechanisms for passing *50*
choosing a Java mechanism *51*
Java mechanisms for passing *50*
Unknown value (?) mappings *45*
Passing parameters *43*
See also Parameters
PRO_* data type constants *239*
procertm utility *217*
ProChangeSummary class
accessing instances *74*
methods
extended *98*
inherited from ChangeSummary *96*
tracking changes to a ProDataGraph *74*

ProcObjects
class factory method 24
example 32
generation 22
ProDataGraph class
See also ProDataSet parameters
accessing the meta data (unknown schema) 66
accessing with a known schema 64
adding data 71
change tracking 57
constructors
using an initial DataGraph 78
using meta data 78
Data Mediator Services requirements 55
data-relations 57
DataGraph extensions for ProDataSets 57
function and foundations 55
holder class for output parameters 61
implementation of DataGraph 55
initializing meta data 69
meta data 57
methods
extended 80
inherited from DataGraph 79
object model 58
overview 50
passing as a parameter
for INPUT or INPUT-OUTPUT 68
for OpenAPI 199, 202
for OUTPUT 63
modes 60
preparing 60
supporting OpenEdge classes 57
updating an instance
adding a row 77
deleting a row 77
error checking 75
finding a row 76
management cycle 74
See also ProDataSet parameters
ProDataGraphHolder class 61
ProDataGraphMetaData class
constructor
Java OpenAPI 199
defining data-relations, OpenAPI 200
defining temp-tables, OpenAPI 200
methods 99
ProDataObject class
accessing
instances using known temp-table name 64
instances using related table rows 76
property meta meta 111
temp-table meta data 66
creating instances 77
initializing meta data 69
ProDataObject class *(continued)*
methods
extended 92
inherited from DataObject 84
temp-table field data type mappings
array fields 63
initial values 63
single-valued fields 62
Unknown value (?) 63
ProDataObjectMetaData class
constructor 101
methods 103
ProDataRelationMetaData class
constructors
Java OpenAPI 200
specifying parent and child ProDataObject key fields 108
specifying ProDataObjectMetaData relations 108
methods 109
ProDataSet parameters
See also ProDataGraph class
ABL mechanisms for passing 50
accessing 54
accessing unknown schema (meta data) 66
accessing with a known schema 64
data-relation support 57
defining data-relations, OpenAPI 200
defining temp-tables, OpenAPI 200
Java mechanisms for passing 50–51
mapping to Java ProDataGraph 59
meta data support 57
overview 54
passing
as INPUT or INPUT-OUTPUT 68
as OUTPUT 63
modes 60
preparing to pass
Java OpenAPI 199, 202
static and dynamic 50
supporting Java classes 57
temp-table field data type mappings
array fields 63
initial values 63
single-valued fields 62
Unknown value (?) 63
updating 73
See also ProDataGraph class
Programming Open Clients
accessing a SmartDataObject API 215
Properties
certificateStore 133
clientPrincipal 134
column 55
connectionLifetime 134
enableTracing 134
idleConnections 135

Properties *(continued)*
idleConnectionTimeout *135*
logFileName *135*
loggingLevel *135*
maxConnections *135*
minConnections *136*
noHostVerify *136*
noSslSessionReuse *136*
nsClientMaxPort *136*
nsClientMinPort *137*
nsClientPickListExpiration *137*
nsClientPicklistSize *137*
nsClientPortRetry *137*
nsClientPortRetryInterval *137*
proxy
command-line access *120*
Connection class *120*
list of available *123*
method access *121*
modifying Connection object *131*
RunTimeProperties class *120*
proxyHost *138*
proxyPassword *138*
proxyPort *138*
proxyUserId *139*
reference *55*
requestWaitTimeout *139*
sessionModel *140*
waitIfBusy *141*
Property interface
Java SDO features *55*
methods *113*
property types *55*
ProResultSet interface
close( ) method *233*
definition *222*
extended methods *234*
getDataType( ) method *233–234*
getObject( ) method *233–234*
next( ) method *233*
standard methods *233*
using *233*
ProResultSetMetaData interface
extension methods *239*
getColumnJavaTypeName( ) method *239*
getColumnProType( ) method *239*
getFieldCount( ) method *239*
getFieldExtent( ) method *239*
getFieldJavaTypeName( ) method *239*
getFieldName( ) method *239*
getFieldProType( ) method *239*
getFieldTypeName( ) method *239*
standard ResultSetMetaData methods *238*
ProResultSetMetaDataImpl class
adding field descriptions for OpenAPI *198*
ProResultSetMetaDataImpl class *(continued)*
constructor
Java OpenAPI *197*
defining an instance for OpenAPI *197*
ProSQLException class *147*
Proxy
exceptions *143*
handling errors *143*
methods
ABL return values *23*
class factory *24*
common *25*
passing parameters, See Parameters
property accessor *121*
running on session-free AppObjects *28*
objects
AppObject instantiation *38*
extending in Java *115*
generated by ProxyGen *22*
objects and methods, See Methods, Proxy objects and methods
properties
command-line access *120*
Connection class *120*
list of available *123*
method access *121*
modifying Connection object *131*
RunTimeProperties class *120*
property accessor methods
Secure Sockets Layer *127*
server *126*
session *124*
thread control *126*
tracing *128*
requirements
generation on Windows *17–18*
using in applications *18*
sample
AppObject *29*
ProcObject *32*
SubAppObject *31*
proxyHost property *138*
proxyPassword property *138*
proxyPort property *138*
proxyUserId property *139*

Q

QUEUED state
request *25*

R

Reference properties *55*
refreshRow( ) method
SDOResultSet *169*

relative( ) method
SDOResultSet 162
_release( ) method 27
releaseConnection( ) method 40
releaseSDOInterface( ) method, SDOResultSet 169
Releasing proxy objects
closing OUTPUT SQL ResultSets 227
Remote ABL methods 23
reOpenQuery( ) method
SDOResultSet
AppServer connections 161
query management 169
row fetch 159
requestWaitTimeout property 139
ResultSetHolder class
definition 222
ResultSetMetaData objects
INPUT TABLE-HANDLE parameters 230
OUTPUT TABLE and TABLE-HANDLE parameters 237
Return values, ABL
Java OpenAPI 209
Returning connection string to a client 39
rowDeleted( ) method
SDOResultSet 166
rowInserted( ) method
SDOResultSet 166
rowUpdated( ) method
SDOResultSet 166
RUNNING state
connection 41
request 25
runProc( ) method
OpenAppObject 180
OpenProcObject 182
RunTimeProperties class
definition 120
general accessor methods 121
purposed accessor methods 124
relationship to Connection class 130

S

Sample Java client applications
OpenAPI example code
non-persistent procedure 211
persistent procedure 212
passing a temp-table as an SQL ResultSet 241
Schema, See Meta data
Scrolling modes, SDOResultSet 158
SDOAppObject class
connecting to AppServer 153
definition 152
SDOModificationException class
getFields( ) method 174
getTable( ) method 174
SDOModificationException class *(continued)*
getText( ) method 174
nextMessage( ) method 174
SDOParameters class
definition 152
getFetchSize( ) method 156
getPrefetchMaxRows( ) method 156
getRowIdentity( ) method 156
getScrollingMode( ) method 156
getStateless( ) method 156
passing 154
setFetchSize( ) method 156
setPrefetchMaxRows( ) method
SDOResultSet stateless mode 159
setRowIdentity( ) method 156
setScrollingMode( ) method 156
setStateless( ) method 156
SDOResultSet class
absolute( ) method 162, 169
attachToAppObj( ) method 161
batch mode 167
beforeFirst( ) method 162
boolean wasNull( ) method 169
cancelBatch( ) method 167
cancelRowUpdates( ) method 166
class factory method 24
close( ) method 169
comparing with JDBC 176
creating instance 154
definition 152
deleteRow( ) method 166
detachFromAppObj ( ) method 161
detaching
typical scenario 161
findColumn( ) method 169
first( ) method 162
getFetchSize( ) method 169
getQuery( ) method 169
getRow( ) method 162
getRowIdentity( ) method 169
getSDOInterface( ) method 169
getting column values 163
inBatch( ) method 167
insertRow( ) method 166
isAfterLast( ) method 162
isAttached( ) method 161
isBeforeFirst( ) method 162
isFirst( ) method 162
meta data
nonstandard methods 172
standard JDBC 2 methods 172
methods and operations 160
miscellaneous management methods 169
moveToCurrentRow( ) method 166
moveToInsertRow( ) method 166
multi-threading 173

SDOResultSet class *(continued)*
navigating rows *162*
next( ) method *162*
refreshRow( ) method *169*
relative( ) method *162*
releaseSDOInterface( ) method *169*
reOpenQuery( ) method
AppServer connections *161*
query management *169*
row fetch *159*
rowDeleted( ) method *166*
rowInserted( ) method *166*
rowUpdated( ) method *166*
scrolling modes *158*
sendBatch( ) method *161*
sendBatchAndReOpen( ) method
AppServer connections *161*
batch mode *167*
SmartDataObject management methods *169*
startBatch( ) method *167*
stateless mode *159*
transaction concurrency control *164*
transactions *164*, *166*
updateRow( ) method *166*
updating
columns *164*
rows *166*
visibility *168*
SDOResultSetMetaData interface
getColumnCount( ) method *172*
getColumnDisplaySize( ) method *172*
getColumnLabel( ) method *172*
getColumnName( ) method *172*
getColumnType( ) method *172*
getColumnTypeName( ) method *172*
getMetaData( ) method *169*
getTableName( ) method *172*
isNullable( ) method *172*
isReadOnly( ) method *172*
isWritable( ) method *172*
SDOScrollingMode class
FORWARD_ONLY *158*
KEEP_ROWS *158*
PREFETCH *158*
Secure Sockets Layer (SSL)
certificate store
file format conversion *219*
management *218*
client certificate management *217*
property accessor methods *127*
Security
Open Client packages *15*
sendBatch( ) method
SDOResultSet *161*
sendBatchAndReOpen( ) method
SDOResultSet
AppServer connections *161*
batch mode *167*
Session property
accessor methods *124*
sessionModel property *140*
set( ) method
ProDataObject *88*
setBigDecimal( ) method
ProDataObject *89*
setBoolean( ) method
ProDataObject *89*
setBytes( ) method
ProDataObject *90*
setChildTableReferences( ) method
ProDataGraph *83*
setColumns( ) method
ProDataRelationMetaData *110*
setFetchSize( ) method
SDOParameters *156*
setFieldMetaData( ) method
ProDataObjectMetaData *106*
ProResultSetMetaDataImpl *231*
setGregorianCalendar( ) method
ProDataObject *95*
setInt( ) method
ProDataObject *90*
setList( ) method
ProDataObject *91*
setLong( ) method
ProDataObject *91*
setNoSchemaMarshal( ) method
InputResultSet *228*
ProDataObjectMetaData *106*
setPrefetchMaxRows( ) method
SDOParameters *156*
SDOResultSet stateless mode *159*
setRowIdentity( ) method
SDOParameters *156*
setScrollingMode( ) method
SDOParameters *156*
setStateless( ) method
SDOParameters *156*
setString( ) method
ProDataObject *92*
Single-run procedure
OpenAPI
example *213*
ProxyGen *24*
Singleton procedure
OpenAPI
example *213*
ProxyGen *24*
SmartDataObjects (SDOResultSets)

SmartDataObjects (SDOResultSets) *(continued)*
See also SDOResultSet class
definition *150*
direct API access *215*
Java access
methods supported *169*
objects required *152*
requirements *150*
tools and documentation *151*
mapping from ProcObjects *216*
techniques for accessing *215*
See also SDOResultSet class
startBatch( ) method
SDOResultSet *167*
Stateless mode, SDOResultSet *159*
Streaming
connection state *41*
definition *227*
request state *25*
STREAMING state
connection *41*
request *25*
SubAppObjects
example *31*
generation *22*

T

Temp-table parameters
ABL field data types *51*
ABL mechanisms for passing *50*
choosing a Java mechanism *51*
field data types and meta data *51*
Java mechanisms for passing *50*
passing *49*
ProDataGraph mapping *50*
SQL ResultSet mapping *51*
static and dynamic *50*
Temp-table parameters as ProDataGraphs
default field data type mappings
array fields *63*
initial values *63*
single-valued fields *62*
Unknown value (?) *63*
INPUT and INPUT-OUTPUT *68*
Java OpenAPI
schema definition *205*
OUTPUT *63*
Temp-table parameters as SQL ResultSets
accessing ABL array fields *224*
accessing schemas *227*
application example *241*
call-back sequence *225*
data type conversions supported *235*
default field data type mappings *222*
Temp-table parameters as SQL ResultSets *(continued)*
INPUT
call-back sequence *225*
passing *228*
ResultSet objects *227*
TABLE-HANDLE meta data *230*
INPUT-OUTPUT
call-back sequence *225*
passing *241*
Java OpenAPI
adding *198*
adding field descriptions *198*
defining a ProResultSetMetaDataImpl object *197*
schema definition *197*
OUTPUT
accessing *233*
call-back sequence *225*
closing *227*
passing *232*
ResultSet objects *227*
TABLE and TABLE-HANDLE meta data *237*
preparing *222*
Progress data type constants *239*
Thread control property
accessor methods *126*
Threading model
session-free *28*
Time conversions in parameters *45*
Tracing property
accessor methods *128*
Transactions, SDOResultSet
concurrency control *164*
extended *166*
Type interface
See also ProDataObjectMetaData class
Java SDO features *55*
methods *111*
See also ProDataObjectMetaData class

U

Unknown value (?)
arrays *46*
Java client programming *45*
mapping to ProDataObject column properties *63*
updateRow( ) method
SDOResultSet *166*
Updating
ProDataGraph objects
adding a row *77*
deleting a row *77*
error checking *75*
finding a row *76*
management cycle *74*
SDOResultSet objects
batch mode *167*

Updating *(continued)*
SDOResultSet objects *(continued)*
column values *164*
rows *166*
visibility *168*

V

Viewing files in the certificate store *217*

W

waitIfBusy property *141*