

Progress Open Client Developer's Guide

© 2002 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress, Powered by Progress, Progress Fast Track, Progress Profiles, Partners in Progress, Partners en Progress, Progress en Partners, Progress in Progress, P.I.P., Progress Results, ProVision, ProCare, ProtoSpeed, SmartBeans, SpeedScript, and WebSpeed are registered trademarks of Progress Software Corporation in the U.S. and other countries. A Data Center of Your Very Own, Allegrix, Apptivity, AppsAlive, AppServer, ASPen, ASP-in-a-Box, Empowerment Center, Fathom, Future Proof, IntelliStream, OpenEdge, PeerDirect, POSSE, POSSENET, Progress Dynamics, Progress Software Developers Network, SectorAlliance, SmartObjects and WebClient are trademarks or service marks of Progress Software Corporation in the U.S. and other countries.

SonicMQ is a registered trademark of Sonic Software Corporation in the U.S. and other countries.

Vermont Views is a registered trademark of Vermont Creative Software in the U.S. and other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997, the IBM XML Parser for Java Edition, and software developed by the Apache Software Foundation (http://www.apache.org/).

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

The implementation of the MD5 Message-Digest Algorithm used by Progress Software Corporation in certain of its products is derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

May 2002



Product Code: 4525
Item Number: 89433;V9.1D

Contents

Preface . . . . . **xiii**
Purpose xiii
Audience xiii
Organization Of This Manual xiv
Typographical Conventions xv
Syntax Notation xvi
Example Procedures xx
Progress Messages xxiii
Other Useful Documentation xxiv
Getting Started xxiv
Development Tools xxvi
Reporting Tools xxvii
4GL xxviii
Database xxix
DataServers xxix
SQL-89/Open Access xxx
SQL-92 xxx
Deployment xxxi
WebSpeed xxxi
Reference xxxii

1. Overview **1–1**
1.1 Architecture 1–2
1.2 Developing and Running an Open Client Application 1–3
1.3 Accessing AppServer Functionality 1–5
1.3.1 Object Model 1–5

2. Configuration and Deployment 2–1
2.1 Selecting an Open Client Runtime Package 2–2
2.2 Preparing To Generate Proxies 2–6
2.2.1 Preparing To Generate Proxies For a Java Client Using GUI ProxyGen. 2–6
2.2.2 Preparing To Generate Proxies For a Java Client On UNIX Using Batch ProxyGen 2–7
2.2.3 Preparing To Generate Proxies For an ActiveX Client Using GUI ProxyGen. 2–8
2.3 Building an Open Client Application 2–11
2.3.1 Building a Client Application That Uses a Java Proxy 2–11
2.3.2 Building a Client Application That Uses an ActiveX Proxy 2–12
2.4 Deploying an Open Client Application 2–13
2.4.1 Deploying a Client Application That Uses a Java Proxy 2–13
2.4.2 Deploying a Client Application That Uses an ActiveX Proxy. . . 2–14
2.5 Managing Digital Certificates 2–15

3. Generating and Managing Proxies 3–1
3.1 Overview and Getting Started With ProxyGen 3–2
3.1.1 Starting and Using GUI ProxyGen 3–3
3.1.2 Defining and Editing a Proxy. 3–5
3.1.3 Specifying Generation Options 3–5
3.1.4 Saving a Project File 3–5
3.1.5 Validating and Generating a Proxy 3–5
3.1.6 Running Batch ProxyGen 3–6
3.2 Defining Your Proxy Using ProxyGen 3–6
3.2.1 Defining an AppObject Or SubAppObject. 3–7
3.2.2 Specifying 4GL Procedures and User-defined Functions 3–8
3.2.3 Customizing Procedure Access 3–13
3.2.4 Saving the Proxy Definition In a Project File 3–14
3.2.5 Validating and Generating the Proxy 3–14
3.3 Changing the Propath Setting Within a Project 3–17
3.4 Object Naming In ProxyGen 3–18
3.4.1 Proxy Naming Conventions. 3–19
3.4.2 Automatic Name Conversions. 3–20
3.4.3 Naming Side Effects and Restrictions. 3–21
3.5 Proxy Validation and Generation Rules 3–21
3.5.1 Proxy Validation 3–22
3.5.2 Proxy Generation 3–23

3.6 Managing Proxies During AppServer Development 3–24
3.6.1 Replacing a Lost Project File 3–24
3.6.2 A Sample Management Scenario 3–25
3.7 Using GUI and Batch ProxyGen Execution Options 3–26
3.7.1 GUI ProxyGen 3–26
3.7.2 Batch ProxyGen Platform Options 3–28

4. Programming Concepts. 4–1
4.1 Programming Overview 4–2
4.2 Connecting To a Progress AppServer 4–2
4.2.1 Connecting Over an Intranet 4–3
4.2.2 Connecting Over the Internet 4–5
4.3 Obtaining Run-time Information 4–7
4.4 Understanding Proxy Object Methods 4–8
4.4.1 Connection Methods 4–9
4.4.2 Remote 4GL Methods 4–9
4.4.3 Class Factory Methods 4–11
4.4.4 Common Methods 4–12
4.5 Using Proxy Objects 4–13
4.5.1 AppObjects 4–13
4.5.2 SubAppObjects 4–17
4.5.3 ProcObjects 4–19
4.6 Passing Parameters In Proxy Object Methods 4–22
4.6.1 Passing TEMP-TABLE And TABLE-HANDLE Parameters . . . 4–22
4.6.2 Result Set Schemas 4–27
4.6.3 Comparing the Flat Model and Array Model For Result Set Access 4–27
4.7 Accessing Proxy Run-time Properties 4–28
4.8 Handling Errors In an Open Client Application 4–29
4.8.1 Client-side Errors 4–29
4.8.2 Server-side Errors Before Remote 4GL Execution 4–30
4.8.3 Server-side Errors During Remote 4GL Execution 4–30

5. Programming Java Clients 5–1
5.1 Connecting To an AppServer 5–2
5.2 Data Type Mapping 5–2
5.2.1 Progress Unknown Values 5–4
5.2.2 INPUT-OUTPUT and OUTPUT Parameters 5–4
5.3 Passing TEMP-TABLE Or TABLE-HANDLE Parameters 5–7
5.3.1 Default Data-type Mapping For TEMP-TABLE Fields 5–8
5.3.2 Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT Parameter 5–10
5.3.3 Providing Metadata For INPUT TABLE-HANDLE Parameters 5–12

5.3.4 Passing a TEMP-TABLE Or TABLE-HANDLE As an OUTPUT Parameter 5–15
5.3.5 Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT-OUTPUT Parameter 5–15
5.3.6 Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output 5–16
5.3.7 Getting Metadata For OUTPUT TEMP-TABLE and TABLE-HANDLE Parameters 5–19
5.3.8 The Java Client Application Using AccountInfo 5–24
5.4 Extending Proxy Objects 5–28
5.5 Using Proxies In Multi-threaded Applications 5–31
5.5.1 Connection States 5–31
5.5.2 Request States 5–31
5.5.3 Cancelling a Request 5–32
5.5.4 Multiple-thread Access To ResultSets 5–32
5.6 Accessing Proxy Run-time Properties 5–33
5.6.1 Methods For Tracing 5–33
5.6.2 Methods For Thread Control 5–34
5.6.3 Methods For Specifying a Proxy Server 5–35
5.6.4 Methods For Managing Digital Certificates 5–36
5.7 Building Java Applets As Open Clients 5–37
5.8 Handling Errors 5–37

6. Using SmartDataObjects From Java Clients 6–1
6.1 What Is a SmartDataObject (SDO)? 6–2
6.2 Accessing Remote SmartDataObjects From Java 6–2
6.2.1 Requirements For Accessing SmartDataObjects 6–3
6.2.2 Example Java Access To a SmartDataObject 6–4
6.2.3 SmartDataObject Access Tools and Documentation 6–4
6.2.4 Objects Required To Access SmartDataObjects 6–5
6.2.5 Developing and Deploying a SmartDataObject-aware Application 6–6
6.2.6 Writing a SmartDataObject-aware Java Client 6–7
6.3 Connecting To an AppServer Using an SDOAppObject 6–7
6.4 Creating an SDOResultSet Object 6–8
6.5 Working With SDOResultSet Objects 6–16
6.5.1 Detaching SDOResultSet Objects [Extension] 6–17
6.5.2 Navigating SDOResultSet Rows 6–18
6.5.3 Getting SDOResultSet Column Values 6–20
6.5.4 Updating SDOResultSet Objects 6–21
6.5.5 Miscellaneous Management Methods 6–28
6.5.6 Metadata Methods 6–32

6.6 Multi-threading 6–35
6.7 Errors and Exceptions 6–35
6.7.1 SDOResultSet Exceptions 6–36
6.7.2 Typical Errors By Method Category 6–37
6.8 Choosing JDBC ResultSets Or SDOResultSets To Access Progress Data 6–38
6.9 Accessing the Sample Application 6–38

7. Programming ActiveX Controller Clients. 7–1
7.1 Accessing Proxies As Automation Objects 7–2
7.1.1 Using Early Binding 7–2
7.1.2 Using Late Binding 7–3
7.1.3 Managing Object Versions 7–3
7.2 Connecting To an AppServer 7–4
7.3 Data Type Mapping 7–4
7.3.1 Progress Data Type Requirements 7–5
7.3.2 Data Type Conversions 7–6
7.4 Passing TEMP-TABLE and TABLE-HANDLE Parameters 7–6
7.4.1 Passing a TEMP-TABLE OR TABLE-HANDLE As an Input Parameter 7–7
7.4.2 Providing Metadata For INPUT TABLE-HANDLE Parameters 7–11
7.4.3 Passing a TEMP-TABLE Or TABLE-HANDLE As an Output Parameter 7–14
7.5 Accessing Proxy Run-time Properties 7–20
7.5.1 Methods For Tracing 7–20
7.5.2 Methods For Thread Control 7–21
7.5.3 Properties For Specifying a Proxy Server 7–22
7.5.4 Properties For Managing Digital Certificates 7–23
7.6 Handling Errors 7–23
7.7 Payment Account Example 7–25
7.7.1 4GL Procedures 7–25
7.7.2 The ActiveX Client Application Using AccountInfo 7–25

8. Special Considerations For Java Applets 8–1
8.1 Asking For Permission In an Applet 8–2
8.1.1 Asking For Permission In Microsoft Internet Explorer 8–2
8.1.2 Asking For Permission In Netscape Version 4 8–3
8.1.3 Supporting Both Browsers In a Single Applet 8–5
8.1.4 Asking For Permission In Netscape Version 6 8–6
8.2 Signing an Open Client Applet 8–7
8.3 Adding Your Applet To an HTML Page 8–8
8.3.1 Microsoft Internet Explorer 8–9
8.3.2 Navigator 8–10
8.3.3 Supporting Both Internet Explorer and Navigator With the Same Page 8–11

A. Accessing a SmartDataObject API Directly A–1
A.1 Providing Access To the SmartDataObject API In a Proxy A–2
A.1.1 Defining a ProcObject That Maps a SmartDataObject A–2

B. Certificate Management Utility B–1
B.1 Managing Certificate Store Files B–2
B.2 Converting Digital Certificates B–4

Index Index–1

Figures

Figure 1–1:	Open Client Architecture	1–2
Figure 3–1:	Starting GUI ProxyGen On Windows	3–3
Figure 3–2:	ProxyGen Main Window	3–4
Figure 6–1:	Java Open Client Access To a Remote SmartDataObject	6–3

Tables

Table 1–1:	Types Of Proxy Objects	1–8
Table 2–1:	Open Client Package Options	2–3
Table 2–2:	Security Supported By Package	2–5
Table 2–3:	Root Digital Certificate Packages	2–16
Table 3–1:	Customizing GUI ProxyGen Startup Parameters	3–27
Table 4–1:	Proxy Common Methods	4–12
Table 4–2:	Passing an INPUT TEMP-TABLE Or TABLE-HANDLE Parameter	4–24
Table 4–3:	Passing an OUTPUT TEMP-TABLE Or TABLE-HANDLE Parameter	4–24
Table 4–4:	Passing an INPUT-OUTPUT TEMP-TABLE Or TABLE-HANDLE Parameter	4–25
Table 5–1:	Progress To Java Data Type Mapping For Parameters	5–3
Table 5–2:	Progress/Java Data Type Map For Output Parameters	5–5
Table 5–3:	Default Data-type Mapping For TEMP-TABLE Fields	5–8
Table 5–4:	Data Type Conversions For TEMP-TABLE Fields	5–9
Table 5–5:	Mapping Between SQL and Progress Data Types For Standard JDBC ResultSet	5–12
Table 5–6:	Progress-SQL ResultSet Data Type Mappings	5–20
Table 5–7:	Comparing Some Array and Flat Model Methods	5–23
Table 5–8:	General 4GL Exceptions For Java Proxies	5–38
Table 6–1:	Output Result Set Exceptions For SDOResultSet	6–36
Table 7–1:	Proxy Component Names Generated By ProxyGen	7–2
Table 7–2:	ActiveX Data Types In Progress and VB	7–4
Table 7–3:	Mapping Between Progress and DAO Data Types	7–11
Table 7–4:	Mapping Between Progress and ADO Data Types	7–12
Table 7–5:	ActiveX Data Types For TEMP-TABLE Output Parameters	7–18
Table 7–6:	HRESULT Values For ActiveX Proxies	7–24

Procedures

Java AppObject 4–15
Java SubAppObject 4–18
Java ProcObject 4–20
Java Client Using AccountInfo 5–24
PayeeList Class for Example 5–3 5–26
AccountInfo.p 5–27
Account Class (Delegating) 5–28
Tax Class Constructor 5–29
Account and Tax Class Extensions 5–30
Java Client Using an SDOAppObject 6–4
AccountInfo.p 7–25
ActiveX Client Using AccountInfo 7–26

Examples

Example 4–1: Java Account Class 4–15
Example 4–2: Java Tax Class 4–18
Example 4–3: Java AccountInfo Class 4–20
Example 5–1: Output Holder Class 5–4
Example 5–2: Holder Class Definitions 5–6
Example 5–3: Java Client Using Proxy Objects 5–25
Example 5–4: Java Input ResultSet Example 5–27
Example 5–5: AppServer Application 5–27
Example 5–6: Delegating Classes 5–28
Example 5–7: SubAppObject/ProcObject Constructors 5–29
Example 5–8: Class Extensions 5–30
Example 7–1: ActiveX TEMP-TABLE Input Parameter Using ADO 7–9
Example 7–2: ActiveX TEMP-TABLE Input Parameter Using DAO 7–10
Example 7–3: ActiveX TEMP-TABLE Input Parameter Using IClientTempTable 7–10
Example 7–4: ActiveX TEMP-TABLE Output Parameter Using ProTempTable 7–19
Example 7–5: AppServer Application 7–25
Example 7–6: ActiveX Client Using AppServer Objects 7–27

Preface

Purpose

The Progress AppServer excels at encapsulating business functions written in the Progress 4GL and making them available to 4GL client applications anywhere on an enterprise network. With the Open Client Toolkit, you can now extend your departmental applications, for example, written in Visual Basic, or your Web-enabled applications written in Java to work with the same business functions used by your Progress enterprise applications.

Using the Open Client Toolkit, you can build or modify an existing Java application, Java applet, or ActiveX Automation Controller to access the business functions on a Progress AppServer. This guide describes how to use the tools provided with the Open Client Toolkit to accomplish this task.

Audience

This guide assumes that you are experienced in programming Java applications, Java Applets, or ActiveX Controllers for your development and deployment environment, and that you have access to documentation on Progress Version 9, especially *Building Distributed Applications Using the Progress AppServer* and its prerequisites.

Before building an application with the help of the Open Client Toolkit, you should be familiar with the information provided by the first two chapters of *Building Distributed Applications Using the Progress AppServer*.

Organization Of This Manual

Chapter 1, “Overview”

Describes the basic mechanisms provided to access Progress AppServer functionality from a non-Progress (*Open Client*) application.

Chapter 2, “Configuration and Deployment”

Details the configuration requirements for developing and deploying Open Client applications.

Chapter 3, “Generating and Managing Proxies”

Explains how to use the Open Client Proxy Generator (ProxyGen) to encapsulate any 4GL API available on an AppServer for access by an Open Client application, and how to manage the result for the application environment.

Chapter 4, “Programming Concepts”

Describes the mechanisms and techniques available to access the functionality encapsulated by ProxyGen from all Open Client applications.

Chapter 5, “Programming Java Clients”

Describes the mechanisms and techniques available to access the functionality encapsulated by ProxyGen from Java applications.

Chapter 6, “Using SmartDataObjects From Java Clients”

Describes how to access remote Progress SmartDataObjects from a Java Open Client using either a prebuilt proxy or a customized proxy that you can build with ProxyGen.

Chapter 7, “Programming ActiveX Controller Clients”

Describes the mechanisms and techniques available to access the functionality encapsulated by ProxyGen from all ActiveX Controllers, with emphasis on Visual Basic applications.

Chapter 8, “Special Considerations For Java Applets”

Describes the considerations you must make, and the tasks you must perform when you develop Java applets for Open Client.

Appendix A, "Accessing a SmartDataObject API Directly"

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

Appendix B, "Certificate Management Utility"

Describes the Certificate Management Tool provided by Progress, and documents how to import, export and remove certificates to and from `.jar` and `.zip` files.

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, GET, GO
  ALT, CTRL, SPACEBAR, TAB

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

  ```
  FOR EACH Customer:
     DISPLAY Name.
  END.
  ```

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

  In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

SYNTAX

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In some instances, square brackets are not a syntax notation, but part of the language.

For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

SYNTAX

```
INITIAL [ constant [ , constant ] ... ]
```

NOTE: The ellipsis (**...**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

  SYNTAX

  ```
  { BY expression [ DESCENDING ] }
  ```

  In some cases, braces are not a syntax notation, but part of the language.

  For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

  SYNTAX

  ```
  { &argument-name }
  ```

- A vertical bar (**|**) indicates a choice.

  In this example, EACH, FIRST, and LAST are optional, but you can only choose one:

SYNTAX

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must select one of *logical-name* or *alias*:

SYNTAX

```
CONNECTED ( { logical-name | alias } )
```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

  In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

  ```
  MAXIMUM ( expression , expression [ , expression ] ... )
  ```

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

  SYNTAX

  ```
  MESSAGE { expression | SKIP [ (n) ] } ...
  ```

  In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

  SYNTAX

  ```
  { include-file
      [ argument | &argument-name = "argument-value" ] ... }
  ```

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

  In this example, WITH is followed by several optional items:

  SYNTAX

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

  In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

  SYNTAX

```
ASSIGN {   { [ FRAME frame ]
               { field [ = expression ] }
               [ WHEN expression ]
           } ...
         | { record [ EXCEPT field ... ] }
       }
```

Example Procedures

This manual provides numerous example procedures that illustrate syntax and concepts. Examples use the following conventions:

- They appear in boxes with borders.
- If they are available online, the name of the procedure appears above the left corner of the box and starts with a prefix associated with the manual that references it, as follows:
  - `e-` — *Progress External Program Interfaces*, for example, `e-ddeex1.p`
  - `lt-` — *Progress Language Tutorials*, `lt-05-s3.p`
  - `p-` — *Progress Programming Handbook*, for example, `p-br01.p`
  - `r-` — *Progress Language Reference*, for example, `r-dynbut.p`

  If the name does not start with a listed prefix, the procedure is not available online.
- If they are not available online, they compile as shown, but might not execute for lack of completeness.

Accessing Files In Procedure Libraries

Documentation examples are stored in procedure libraries, `prodoc.pl` and `prohelp.pl`, in the `src` directory where Progress is installed.

You **must** first create all subdirectories required by a library before attempting to extract files from the library. You can see what directories and subdirectories a library needs by using the PROLIB -list command to view the contents of the library. See the *Progress Client Deployment Guide* for more details on the PROLIB utility.

Creating a Listing Of the Procedure Libraries

Creating a listing of the source files from a procedure library involves running PROENV to set up your Progress environment, and running PROLIB.

1 ♦ From the Control Panel or the Progress Program Group, double-click the Proenv icon.

2 ♦ The Proenv Window appears, with the proenv prompt.

Running Proenv sets the DLC environment variable to the directory where you installed Progress (by default, `C:\Program Files\Progress`). Proenv also adds the DLC environment variable to your PATH environment variable and adds the bin directory (`PATH=%DLC%;%DLC%\bin;%PATH%`).

3 ♦ Enter the following command at the proenv prompt to create the text file `prodoc.txt` which contains the file listing for the prodoc.pl library:

```
PROLIB %DLC%\src\prodoc.pl -list > prodoc.txt
```

Extracting Source Files From Procedure Libraries On Windows Platforms

Extracting source files from a procedure library involves running PROENV to set up your Progress environment, creating the directory structure for the files you want to extract, and running PROLIB.

1 ♦ From the Control Panel or the Progress Program Group, double-click the Proenv icon.

2 ♦ The Proenv Window appears, with the proenv prompt.

Running Proenv sets the DLC environment variable to the directory where you installed Progress (by default, `C:\Program Files\Progress`). Proenv also adds the DLC environment variable to your PATH environment variable and adds the bin directory (`PATH=%DLC%;%DLC%\bin;%PATH%`).

3 ♦ Enter the following command at the proenv prompt to create the `prodoc` directory in your Progress working directory (by default, `C:\Progress\Wrk`):

```
MKDIR prodoc
```

4 ♦ Create the `langref` directory under `prodoc`:

```
MKDIR prodoc\langref
```

5 ♦ To extract all examples in a procedure library directory, run the PROLIB utility. Note that you must use double quotes because "Program Files" contains an embedded space:

```
PROLIB "%DLC%\src\prodoc.pl" -extract prodoc\langref\*.*
```

PROLIB extracts all examples into `prodoc\langref`.

To extract one example, run PROLIB and specify the file that you want to extract as it is stored in the procedure library:

```
PROLIB "%DLC%\src\prodoc.pl" -extract prodoc/langref/r-syshlp.p
```

PROLIB extracts `r-syshlp.p` into `prodoc\langref`.

Extracting Source Files From Procedure Libraries On UNIX Platforms

To extract `p-wrk1.p` from `prodoc.pl`, a procedure library, follow these steps at the UNIX system prompt:

1 ♦ Run the PROENV utility:

```
install-dir/dlc/bin/proenv
```

Running proenv sets the DLC environment variable to the directory where you installed Progress (by default, `/usr/dlc`). The proenv utility also adds the bin directory under the DLC environment variable to your PATH environment variable (`PATH=$DLC/bin:$PATH`).

2 ♦ At the proenv prompt, create the `prodoc` directory in your Progress working directory:

```
mkdir prodoc
```

3 ♦ Create the `proghand` directory under `prodoc`:

```
mkdir prodoc/proghand
```

4 ♦ To extract all examples in a procedure library directory, run the PROLIB utility:

```
prolib $DLC/src/prodoc.pl -extract prodoc/proghand/*.*
```

PROLIB extracts all examples into `prodoc/proghand`.

To extract one example, run PROLIB and specify the file that you want to extract as it is stored in the procedure library:

```
prolib $DLC/src/prodoc.pl -extract prodoc/proghand/p-wrk-1.p
```

PROLIB extracts `p-wrk-1.p` into `prodoc/proghand`.

Progress Messages

Progress displays several types of messages to inform you of routine and unusual occurrences:

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify, or that are assumed, as part of the procedure. This is the most common action taken following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

Progress messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

On the Windows platform, use Progress online help to get more information about Progress messages. Many Progress tools include the following Help menu options to provide information about messages:

- Choose Help→Recent Messages to display detailed descriptions of the most recent Progress message and all other messages returned in the current session.
- Choose Help→Messages, then enter the message number to display a description of any Progress message. (If you encounter an error that terminates Progress, make a note of the message number before restarting.)
- In the Procedure Editor, press the **HELP** key (**F2** or **CTRL-W**).

On the UNIX platform, use the Progress PRO command to start a single-user mode character Progress client session and view a brief description of a message by providing its number. Follow these steps:

1 ♦ Start the Progress Procedure Editor:

```
install-dir/dlc/bin/pro
```

2 ♦ Press **F3** to access the menu bar, then choose Help→Messages.

3 ♦ Type the message number, and press **ENTER**. Details about that message number appear.

4 ♦ Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose File→Exit.

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows and Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A bulletin that provides a list of new and changed features by release, a list and description of changes to documentation by release, and critical information about product changes that might require changes to existing code and configurations.

This bulletin also provides information about where to go for detailed information about the new and changed features and documentation.

Progress Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Language Tutorial for Windows and *Progress Language Tutorial for Character*

Platform-specific tutorials designed for new Progress users. The tutorials use a step-by-step approach to explore the Progress application development environment using the 4GL.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup and shutdown commands that you use at the command line, and the startup parameters that you use for Progress processes. This guide also provides information about parameter usage and parameter files.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Progress Basic Development Tools (Character only; information for Windows is in online help)

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for UNIX*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Building Distributed Applications Using the Progress AppServer

A guide that provides comprehensive information about building and implementing distributed applications using the Progress AppServer. Topics include basic product information and terminology, design options and issues, setup and maintenance considerations, 4GL programming details, and remote debugging.

Progress External Program Interfaces

A guide to the external programming interfaces supported by Progress. This manual covers the Host Language Call (HLC) Interface, the system clipboard, named pipes, shared libraries and DLLS, Windows Dynamic Data Exchange (DDE), COM objects, ActiveX Automation, ActiveX controls, sockets, XML, SAX, and the SonicMQ 4GL Adapter.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress on the Web

A manual that describes how to use the new WebClient, AppServer Internet Adapter, SmartObjects, and SonicMQ Adapter to create applications tailored for Internet, intranet, and extranet environments.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial.

Each DataServer has its own guide as follows:

- *Progress/400 Product Guide*
- *Progress DataServer for Microsoft SQL Server Guide*
- *Progress DataServer for ODBC Guide*
- *Progress DataServer for ORACLE Guide*

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/products/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Developer's Guide

Provides a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

WebSpeed Product Update Bulletin

A booklet that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Welcome to WebSpeed (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

Pocket WebSpeed (Hard copy only)

A reference that lets you quickly look up information about the SpeedScript language or the WebSpeed programming environment.

1

Overview

The Progress Open Client Toolkit exposes Progress AppServer functionality to *Open Clients* (non-Progress clients). The Open Client Toolkit allows you to generate proxy objects, tailored for your particular type of Open Client, that encapsulate the remote 4GL procedures and functions supported on an AppServer. Your Open Client application can then access these AppServer procedures and functions through methods of the generated proxy objects.

The Open Client supports both intranet and Internet access to the AppServer. When you use the Internet, you can use HTTP/S to communicate through firewalls and optionally send secure encrypted data transmissions.

This chapter describes the general Open Client architecture and how ProxyGen encapsulates AppServer procedures map to proxy objects for access by Open Clients and has the following sections:

- Architecture
- Developing and Running an Open Client Application
- Accessing AppServer Functionality

The other chapters provide information about how to develop and deploy Open Client applications to access generated proxy objects.

1.1 Architecture

The Open Client architecture allows Open Clients to access Progress AppServer functionality that is organized into separate 4GL source files (usually procedure `.p` `files`, `.w` files or both).

There are several components involved in exposing AppServer functionality for Open Clients. Figure 1–1 shows the basic components of an Open Client application and provides a high level illustration of the different communication paths an Open Client can use to access an AppServer on a company intranet or over the Internet.

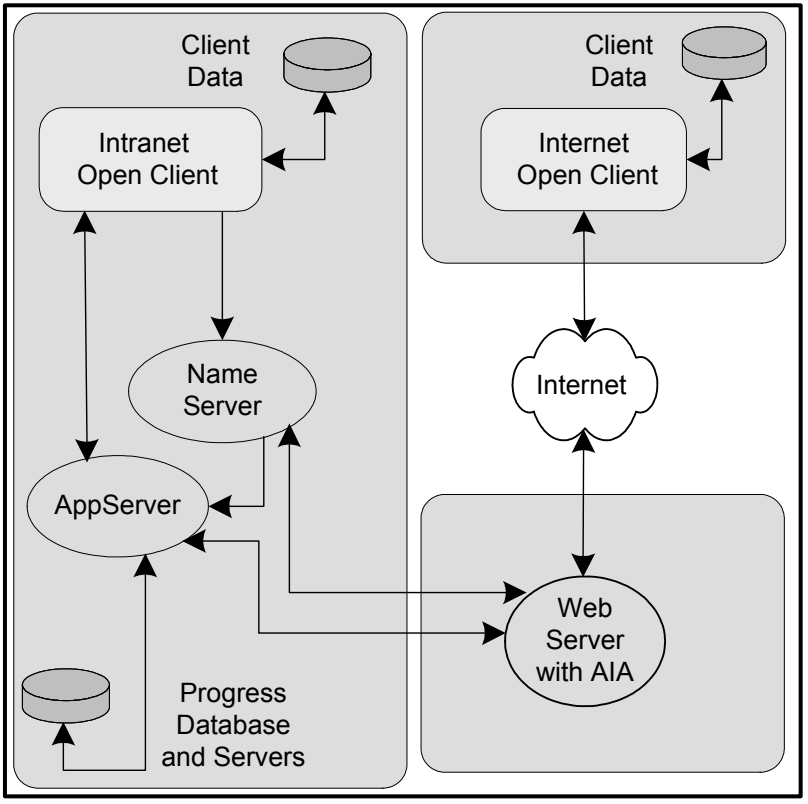

Figure 1–1: Open Client Architecture

The client programmer writes the client application in Java, Visual Basic, or a compatible language. This application executes remote procedures and functions in an AppServer session through methods on a proxy object that you generate using the Open Client Proxy Generator (ProxyGen). ProxyGen is an Open Client tool that generates proxies for Java and ActiveX Open Client applications. The proxy objects are implemented in Java and use the Open Client Runtime to access the AppServer. For more information about using ProxyGen to generate proxies, see "Preparing To Generate Proxies" in Chapter 2, "Configuration and Deployment."

A key feature of the Open Client is full support for relational data exchange. This allows 4GL TEMP-TABLE parameters to be passed between the AppServer and the client, and exposed in the client as result set objects. In this way, Open Client applications can access any Progress database or DataServer that is connected and exposed by the AppServer application.

The types of supported Open Clients include Java applications, Java applets, and ActiveX Automation Controllers. For Java, ProxyGen generates proxy objects as Java classes. For ActiveX, ProxyGen generates proxy objects as Automation objects. Proxies packaged as Automation objects are accessible from any ActiveX Automation Controller. However, this manual presents all ActiveX examples using Visual Basic source code.

If you want to run your Open Client application across the Internet using HTTPS (SSL), you must include digital certificates with your application. The Open Client Toolkit includes the Certificate Management Tool (`procertm` utility) that provides a way to import, export and remove certificates to and from `.jar` and `.zip` files. For more information, see Appendix B, "Certificate Management Utility."

1.2 Developing and Running an Open Client Application

Preparing an Open Client application to access an AppServer requires completing the following tasks using the Open Client Toolkit:

- **Run the Proxy Generator (ProxyGen)** — A Progress development tool that allows a 4GL developer to identify AppServer functionality and expose that functionality as proxy objects to Open Clients.
- **Select an Open Client Runtime package** — A software package that allows your Open Client application to connect to an AppServer and interact with the proxy objects generated by ProxyGen.

The process for developing and running an Open Client application involves several tools and procedures that are summarized in the following steps:

1 ♦ Ensure that your development environment meets the requirements for developing Open Client applications. See Chapter 2, "Configuration and Deployment."

2 ♦ Use ProxyGen to generate Java or ActiveX proxy objects for your AppServer 4GL application. See Chapter 3, "Generating and Managing Proxies".

NOTE: You can skip this step if you want to access only remote Progress SmartDataObjects from a Java application. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."

3 ♦ Write a Java application, Java applet, or ActiveX Controller that uses the proxy objects generated in Step 2 or the predefined SmartDataObject proxy objects. For more information, see Chapter 4, "Programming Concepts," Chapter 5, "Programming Java Clients," and Chapter 7, "Programming ActiveX Controller Clients."

If you want to access remote Progress SmartDataObjects, also see Chapter 6, "Using SmartDataObjects From Java Clients," and Appendix A, "Accessing a SmartDataObject API Directly."

4 ♦ Manage Digital Certificates if your Open Client uses HTTPS (SSL) to communicate with the AppServer. For more information, see the "Managing Digital Certificates" sections in Chapter 2, "Configuration and Deployment."

5 ♦ Deploy the proxy, Open Client application, your optional digital certificates, and supporting software to your client machine (see Chapter 2, "Configuration and Deployment.").

6 ♦ Start any DataServers or Database servers that are accessed by the AppServer. See the *Progress DataServer Guides* and the *Progress Database Administration Guide and Reference*.

7 ♦ Start the AppServer (see *Building Distributed Applications Using the Progress AppServer*).

8 ♦ Start the Java Servlet Engine (JSE) to run the AppServer Internet Adapter (AIA) if you are using HTTP/S.

9 ♦ Run your Open Client application as you have designed it.

1.3 Accessing AppServer Functionality

The Progress AppServer functionality is organized into procedure files. You must compile and save the r-code for these files before ProxyGen can access them, even if your deployment strategy ultimately relies on the source files. (*R-code* is the binary run code that is executed by the Progress interpreter.) Once the AppServer r-code files are available, you can use ProxyGen to identify and customize Open Client access to the procedures and functions defined by these files. To make the r-code files available to ProxyGen, you can:

- Run ProxyGen on the same machine where the r-code resides.
- Run ProxyGen on a machine that has access to a network file system where the r-code resides.
- Copy the r-code files (maintaining the directory structure underneath the AppServer PROPATH) to the machine where you run ProxyGen.

1.3.1 Object Model

ProxyGen requires that you organize Progress AppServer functionality into three possible types of proxy objects:

- Application objects (*AppObjects*)
- Sub-application objects (*SubAppObjects*)
- Procedure objects (*ProcObjects*)

All of these proxy objects organize 4GL procedures on the AppServer, but in different ways. The following sections describe the different ways of organizing proxy objects and contains information about:

- 4GL Procedures
- AppObjects and SubAppObjects
- ProcObjects
- Object Relationships

4GL Procedures

Before describing the three types of proxy objects, it helps to understand a few basic concepts that govern the operation of 4GL procedures. First, any single file that is executable by Progress is an external procedure. An external procedure can contain one or more internal procedures or user-defined functions that execute within and share the context of the external procedure. You can execute an external procedure in two basic ways:

- As a *non-persistent procedure*
- As a *persistent procedure*

When you execute a non-persistent procedure, the procedure executes and returns to the caller, removing all trace of its context from memory after it returns. Any internal procedures and functions that it defines can only be executed by the procedure itself. As a result, a non-persistent procedure executes and returns as a unit without exposing any of its context to the caller.

However, when you execute a persistent procedure, the procedure leaves its context active after it completes execution. Internal procedures and functions then remain available for future execution. Thus, a persistent procedure essentially instantiates a procedure object that exposes its context to the caller through the internal procedures and functions that it defines. For more information on 4GL procedures, see the *Progress Programming Handbook*.

The Progress AppBuilder, which is part of the Progress Application Development Environment (ADE), also allows you to build preprogrammed persistent procedures that adhere to the Progress Application Development Model (ADM). The Progress ADM allows you to build persistent procedures as reusable application building blocks known as SmartObjects. The Open Client Toolkit allows you to access a type of SmartObject known as a SmartDataObject, which dynamically accesses and updates data. As a result, you can run a SmartDataObject on the Progress AppServer like any persistent procedure, and, using the Open Client Runtime, access the SmartDataObject from a Java or ActiveX application, or Java applet. In Java you can also access the SmartDataObject as an extended Java Database Connectivity (JDBC) 2 ResultSet. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients." For more information on SmartDataObjects, see the *Progress AppBuilder Developer's Guide*.

AppObjects and SubAppObjects

AppObjects and SubAppObjects are similar in that both expose the functionality of a group of 4GL external procedures as methods to the Open Client. However AppObjects create and maintain a connection to the AppServer, while a SubAppObject uses the connection already created by an associated AppObject.

Although you can encapsulate an AppServer's entire functionality within one AppObject, dividing a large AppServer application into one AppObject and several SubAppObjects offers several advantages. Multiple objects provide better logical organization and separate name spaces for each of the objects. Additionally, they help avoid the instantiation of large objects that are slow to load and difficult to maintain.

ProcObjects

A ProcObject encapsulates the functionality of a single persistent procedure on the AppServer. It exposes non-PRIVATE internal procedures and functions as methods to the Open Client. A ProcObject always uses its associated AppObject's connection to remotely execute the persistent procedure.

Object Relationships

In an Open Client application, you create the AppObject first in order to access AppServer functionality. You can then create a SubAppObject through its associated AppObject, and you can create a ProcObject through its associated AppObject or SubAppObject. You use a special method, a *Class Factory method*, to create a SubAppObject or ProcObject. Thus, the total package comprised of one AppObject and all SubAppObjects and ProcObjects that share the same AppServer connection is a single *proxy*.

Although there is a creation hierarchy among the different types of proxy objects, there is no functional hierarchy among them. All objects share the AppServer connection with equal status. The objects can be *released* (have their context removed from client memory) in any order with no effect on the objects that remain in use. The connection to the AppServer remains intact until the last object that uses it is released.

In summary, each type of proxy object encapsulates and accesses AppServer functionality as shown in Table 1–1.

Table 1–1: Types Of Proxy Objects

The Proxy Object ...	Provides ...
AppObject	• A connection to an AppServer • Methods to execute Progress non-persistent procedure files • Class factory methods that create SubAppObjects • Class factory methods that create ProcObjects • (Java Only) A special class factory method that creates a built-in ProcObject for accessing a remote Progress SmartDataObject as an extended JDBC 2 ResultSet
SubAppObject	• A shared AppServer connection through an AppObject • Methods to execute Progress non-persistent procedure files • Class factory methods that create ProcObjects • (Java Only) A special class factory method that creates a built-in ProcObject for accessing a remote Progress SmartDataObject as an extended JDBC 2 ResultSet
ProcObject	• A shared AppServer connection through an AppObject • Execution of a remote Progress persistent procedure • Methods to execute Progress internal procedures and user-defined functions

Once you define an AppObject, its methods and any related SubAppObjects and ProcObjects, ProxyGen can generate the proxy that exposes these objects to Open Clients.

The client code accesses the proxy in the client's native environment. As a result, a Java client sees the proxy (and the AppServer functionality) as a Java class definition, while a Visual Basic client sees the proxy as an ActiveX Automation object. To use proxy objects, you must deploy the generated proxy along with the Open Client Runtime to all client application development and deployment systems. For more information about development and deployment, see Chapter 2, "Configuration and Deployment."

The Open Client Toolkit installation includes an SDOAppObject. This is a prebuilt proxy that allows you to access any remote Progress SmartDataObject from a Java application without needing to run ProxyGen. Therefore, an SDOAppObject is essentially an AppObject that provides the built-in interface to a remote SmartDataObject, but not to any other remote 4GL procedure.

If you want to access other remote 4GL procedures in addition to SmartDataObjects, you can include the additional 4GL 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 that you have built.

2

Configuration and Deployment

Each step in the development and deployment of an Open Client application has a separate set of configuration requirements. You must fulfill each of these requirements before beginning the specified task.

This chapter describes the configuration requirements for performing the following Open Client development and deployment tasks:

- Selecting an Open Client Runtime Package
- Preparing To Generate Proxies
- Building an Open Client Application
- Deploying an Open Client Application

2.1 Selecting an Open Client Runtime Package

The Open Client Runtime is part of the client installation that you must supply with your application. Progress packages and distributes the Open Client Runtime in a number of file formats that contain different network protocol support combinations. This distribution method reduces the overhead and complexity for both you the developer and the Open Client deployer.

Based on the run-time configuration you want to support for your Open Client application, you can select one of the following Open Client Runtime distribution packages:

- **Basic Open Client Runtime Packages** — Supports the AppServer and AppServerDC URL connection protocol on an intranet.
- **HTTP Open Client Runtime Packages —** Supports the AppServer, AppServerDC, and HTTP URL connection protocols. These packages support user-implemented authentication to a Web server and Proxy servers.
- **HTTPS Lite Open Client Runtime Packages** — Supports the AppServer, AppServerDC, HTTP, and HTTPS URL connection protocols. These packages support a minimal set of HTTPS (SSL) functionality to achieve a faster download time. Progress software recommends that you use these packages for applets when download time is critical.

  While these packages support many SSL enabled Web servers, they do not support all of them. The SSL configuration for the individual Web server and the Digital Certificate used to authenticate the Web server identity determines whether you can use these packages.

  These packages support user-implemented authentication to a Web server, data encryption, and Proxy servers.
- **HTTPS Standard Open Client Runtime Packages** — Supports the AppServer, AppServerDC, HTTP, and HTTPS URL connection protocols. These packages provide a full set of HTTPS (SSL) functionality and you should use these packages when download time is not critical. These packages support most SSL enabled Web servers. The individual Web server's SSL configuration and the Digital Certificate used to authenticate the identity of the Web server determines whether you can use these packages.

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

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

As you develop your HTTP/S applications, you should consider that the application deployer might require certain data encryption and digital signature algorithms when they configure their Web server's SSL support. These requirements might exceed the capabilities of the HTTPS Lite package you select. Deployers might also require the use of SSL version 2 or TLS protocols instead of the standard SSL version 3. If any of these requirements exists, you must use the HTTPS Standard packages. For Java applets, Progress software recommends using the HTTP Lite package because of its download size.

For more information about selecting a package, see Table 2–1. For more information about the security supported, see Table 2–2.

The following table lists the types of supported applications, the supported protocols, and the Open Client Runtime packages you would use depending on the run-time configuration of your application and the protocols you want to support.

Table 2–1: Open Client Package Options *(1 of 2)*

If your application run-time configuration is...	And the supported protocol is...	The Open Client Runtime package you use is...
An ActiveX Automation controller, such as Visual Basic	AppServer or AppServerDC	Basic (`o4glrt.zip`)[1]
A Java application or servlet	HTTP	HTTP (`o4glrth.zip`)
	HTTPS	HTTPS Lite (`o4glrthsl.zip`)
	HTTPS	HTTPS Standard (`o4glrths.zip`)

Table 2–1: Open Client Package Options *(2 of 2)*

If your application run-time configuration is...	And the supported protocol is...	The Open Client Runtime package you use is...
A Java applet running in the Netscape browser	AppServer or AppServerDC	Basic (`o4glrt.jar`)[1]
	HTTP	HTTP (`o4glrth.jar`)
	HTTPS	HTTPS Lite (`o4glrthsl.jar`)
	HTTPS	HTTPS Standard (`o4glrths.jar`)
Java applet running in the Internet Explorer browser	AppServer or AppServerDC	Basic (`o4glrt.cab`)[1]
	HTTP	HTTP (`o4glrth.cab`)
	HTTPS	HTTPS Lite (`o4glrthsl.cab`)
	HTTPS	HTTPS Standard (`o4glrths.cab`)

[1] Use only for Intranet network configurations

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

For example, based on the options described in Table 2–1, if you have:

- An Intranet UNIX Java application that only requires AppServer support, you would use `o4glrt.zip`.
- An ActiveX application that requires HTTP and full HTTPS support, you would use `o4glrths.zip`.
- An Internet Explorer Java Applet that requires HTTP and limited HTTPS support, you would use `o4glrthsl.cab`.
- A Java application that uses only AppServer, AppServerDC, or HTTP, you would use `o4glrth.zip`.

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

Table 2–2: Security Supported By Package

For This Package...	The Following Security is Supported...
AppServer and AppServerDC	User-implemented authentication using a userid and password
HTTP	• Authentication that includes: – User-implemented authentication using a userid and password – Basic authentication to the AIA's Web server – Basic Proxy server authentication • Proxy servers
HTTPS Lite (limited security)	• Authentication that includes: – User-implemented authentication using a userid and password – Basic Proxy server authentication – HTTP Basic authentication to the AIA's Web server – HTTPS using SSL V3, RSA with MD5 digital signatures, and RSA Key Exchange • Data Encryption using RC4-128 bit key and RC4-40 bit key • Proxy servers
HTTPS Standard (full security)	• Authentication that includes: – User-implemented authentication using a userid and password – Basic Proxy server authentication – HTTP basic authentication to the AIA's Web server – HTTPS using SSL V2, SSL V3, and TLS V1, Digital Signatures: RSA w. MD5, RSA w. SHA1, DSA 2. SHA1, and Key Exchange: RSA, Diffie-Hellman. • Data Encryption using RC4-128 bit key, RC4-40 bit key, DES-56 bit key, DES-40 bit key, Triple-DES-168 bit key, RC2-40 bit key • Proxy servers

2.2 Preparing To Generate Proxies

The Open Client Proxy Generator (ProxyGen) utility allows you to define and generate Open Client proxies. On Windows platforms, ProxyGen provides a graphical user interface (GUI ProxyGen) that you can use to define and generate both ActiveX and Java proxies. On Unix platforms, ProxyGen provides a batch interface (Batch ProxyGen) that you can use to generate Java proxies you have already defined using GUI ProxyGen.

Before you run ProxyGen to generate proxies, you must follow specific preparation procedures based on your environment and client type. The following sections provide information about:

- Preparing To Generate Proxies For a Java Client Using GUI ProxyGen
- Preparing To Generate Proxies For a Java Client On UNIX Using Batch ProxyGen
- Preparing To Generate Proxies For an ActiveX Client Using GUI ProxyGen

2.2.1 Preparing To Generate Proxies For a Java Client Using GUI ProxyGen

To configure your environment to generate a Java proxy using GUI ProxyGen:

1 ♦ Install one of these Progress products on a Windows system where you plan to run GUI ProxyGen:

- Open Client Toolkit
- ProVision Plus

With these products, Progress 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, you must install the appropriate JDK.

NOTE: If you install an alternate JDK for ProxyGen to compile Java proxies, you must specify your compiler location and CLASSPATH to ProxyGen. For more information, see Chapter 3, "Generating and Managing Proxies."

2.2.2 Preparing To Generate Proxies For a Java Client On UNIX Using Batch ProxyGen

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

1 ♦ Make sure that you have a JDK on your system.

NOTE: If you have already installed a Progress product that requires the platform JDK, you do not need to perform this step.

2 ♦ Specify the compiler location and CLASSPATH using the GUI ProxyGen. For more information see Chapter 3, "Generating and Managing Proxies."

3 ♦ If any of the following files are not on your system from another Progress installation, copy them from a Windows system where the Open Client Toolkit is installed to any directory that preserves the directory structure under *Progress-Install-Directory*.

Text files:

a) *Progress-Install-Directory*\bin\bproxygen (Make sure execute mode is set on this file using the chmod command.)

b) *Progress-Install-Directory*\properties\international.properties.

Binary files:

- *Progress-Install-Directory*\java\progress.jar
- *Progress-Install-Directory*\java\proxygen.zip

4 ♦ Set the following environment variables:

a) DLC — Set DLC to the UNIX *Progress-Install-Directory* (from Step 2).

b) JVM — Set JVM to the full pathname of the platform JVM. For example, /jdk/bin/java.

c) PATH — If you do not specify a path to the Java compiler in your proxy ProxyGen project file, you must add the compiler directory to PATH.

d) JDKCP — Only use for Version 1.1.x of Java. Set JDKCP to the full pathname containing classes.zip or the equivalent. For example, /jdk1.1.8/lib/classes.zip.

5 ♦ Copy the ProxyGen project file (.pxg) to any directory.

6 ♦ Make sure that the Progress r-code for the proxy is accessible to your system.

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

NOTE: In the Propath on UNIX, Windows drive letters are ignored and back slashes (\) are automatically changed to forward slashes (/). You can also use dot (.) as a Propath component on UNIX.

2.2.3 Preparing To Generate Proxies For an ActiveX Client Using GUI ProxyGen

To configure your environment to generate an ActiveX proxy, you must first perform the following steps in the specified order:

1 ♦ Install one of the following Progress products on a Windows system where you plan to run GUI ProxyGen:

- Open Client Toolkit
- ProVision Plus

2 ♦ Download the Microsoft Java SDK Version 3.2 (or later) and the Java Virtual Machine (which might not be included in the SDK). Follow the instructions on the Microsoft Web site to complete the download and any subsequent installations.

3 ♦ Ensure the PATH environment variable contains: *Microsoft-Java-SDK-Install-Directory*\bin. See Step 2.

4 ♦ Ensure that ProxyGen can access and execute the MIDL compiler, which it needs to create a Type Library. For instructions on completing this step, see the "Setting Up the MIDL Compiler" section and the "Setting Up the Environment To Run the MIDL Compiler" section.

Setting Up the MIDL Compiler

To set up the MIDL compiler for access by ProxyGen, make sure that one of the following products is installed on the machine:

- Visual J++ Version 1.1
- Visual C++ Version 5
- Visual C++ Version 6

1 ♦ If one of these products is already on your machine, no further installation work is necessary. Skip to the "Setting Up the Environment To Run the MIDL Compiler" section for additional instructions.

2 ♦ Choose which of these products to install. Here are some considerations:

a) By default, ProxyGen uses the files provided by Visual J++ 1.1. An additional switch is required when you run ProxyGen to use MIDL from Visual C++. For more information, see the "Setting Up the Environment To Run the MIDL Compiler" section.

b) The Progress Installation CD contains a copy of Visual J++ Version 1.1 Trial Edition, which you can install directly. For more information, move on to Step 3.

c) Later versions of Visual J++ (for example, Version 6) do *not* contain the MIDL compiler and therefore are not sufficient for this requirement.

d) If you choose to purchase Visual C++, both the Standard Edition and the Professional Edition contain the necessary files.

3 ♦ If you choose to use Visual J++ 1.1 Trial Edition, follow these steps to install it from the Progress Installation CD:

a) Copy the self-extracting executable file `VJTrial.exe` from the Progress Installation CD onto the machine (this file is at the root level on the CD).

b) Execute `VJTrial.exe`.

Setting Up the Environment To Run the MIDL Compiler

Once you have selected and installed one of the recommended products containing the MIDL compiler, you must set up the environment for ProxyGen to run the MIDL compiler.

For Visual J++ Version 1.1, you must perform the following additional steps:

1 ♦ Provide access to the `*.idl` files using one of these techniques:

a) Copy the `*.idl` files from *`Microsoft-Visual-J++-Install-Directory`*`\DevStudio\Vj\include` to the Progress working directory (specified during Progress product installation).

NOTE: You can determine the Progress working directory by viewing the startup options in the `proxygen.prm file` located in the directory where you installed Progress. You can view the `proxygen.prm` file with a text editor.

b) Add the *Microsoft-Visual-J++-Install-Directory*\DevStudio\vj\include to your INCLUDE environment variable.

2 ♦ Ensure that the PATH environment variable contains:

a) *Microsoft-Visual-J++-Install-Directory*\DevStudio\Vj\bin

b) *Microsoft-Visual-J++-Install-Directory*\DevStudio\SharedIDE\bin

CAUTION: The PATH settings in Step 2 **must** be on the PATH **after** the Microsoft Java SDK `bin` directory.

For Visual C++, you must perform the following additional steps:

1 ♦ Add the following startup option to the `proxygen.prm` file located in the directory where you installed Progress:

```
/d:ProxyGen.UseMidlFromC++=yes
```

Surround this option with spaces and place it immediately before `com.progress.open4gl.proxygen.gui.ProxyGen` line in the `proxygen.prm` file. For more information about adding information to the `proxygen.prm` file, see the "Using GUI and Batch ProxyGen Execution Options" section in Chapter 3, "Generating and Managing Proxies."

2 ♦ Update the PATH environment variable to include:

a) Visual C++ `bin` directory — This directory contains `midl.exe`.

For example, for Visual C++ 5.0, you would enter:

VC++-Install-Dir\DevStudio\Vc\bin

or

VC++-Install-Dir\DevStudio\SharedIDE\bin

b) The common directory shared by the IDE for Visual C++ and other products — This is the directory containing the `*.dll` files associated with `midl.exe`.

For example, for Visual C++ 6.0, you would enter:

VC++-Install-Dir`\Microsoft Visual Studio\Vc98\bin`

or

VC++-Install-Dir`\Microsoft Visual Studio\Common\msdev98\bin`

3 ♦ For Windows 98 only, the INCLUDE environment variable must contain the Visual C++ `include` directory.

For example, for Visual C++ 5.0, you would include the following:

- *VC-Install-Dir*`\DevStudio\Vc\include`

For example, for Visual C++ 6.0, you would include the following:

- *VC-Install-Dir*`\Microsoft Visual Studio\Vc98\include`

For more information on using ProxyGen to generate these files, see Chapter 3, "Generating and Managing Proxies."

2.3 Building an Open Client Application

Before you build your application, you must prepare your environment. The preparation depends on the build environment and the client type. The following sections provide information about:

- Building a Client Application That Uses a Java Proxy
- Building a Client Application That Uses an ActiveX Proxy

2.3.1 Building a Client Application That Uses a Java Proxy

To build 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 build the application:

1 ♦ To build the client application on machine other than the proxy machine, you must:

a) Copy the proxy `.class` files to a directory that mirrors the package hierarchy specified in ProxyGen during the generation of the proxy. The package name implies a directory structure that you must maintain when you copy the proxy.

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 for documentation purposes only. Do not attempt to modify and rebuild your proxy from these files.

c) Copy the Open Client Runtime package to any directory on the system. For more information, see the "Selecting an Open Client Runtime Package" section.

2 ♦ Update the CLASSPATH setting for the client application's development environment to include the:

a) Directory containing the proxy (`.class` files) that you copied in Step 1. This is the directory above the package hierarchy.

b) Open Client Runtime package you copied in Step 1.

3 ♦ Write, compile, and execute the client application. For more information, see Chapter 4, "Programming Concepts," and Chapter 5, "Programming Java Clients."

NOTE: If you installed either the Progress Open Client Toolkit or the Progress ProVision Plus products on a Windows system, this also installs the JavaSoft JDK. However, it does not set up the system for general use of the JDK. If you want to use the JDK for your client application development, see the JavaSoft documentation.

2.3.2 Building a Client Application That Uses an ActiveX Proxy

To build a client application that uses an ActiveX proxy (Automation Controller), you must first perform the following steps in the specified order on the system where you plan to build the application:

1 ♦ To build the client application on machine other than the proxy machine, you must:

a) Copy the proxy (`.dll` file) to any directory on the system.

b) Copy the Open Client Runtime package to any directory on the system. For more information, see the "Selecting an Open Client Runtime Package" section.

c) Copy the ActiveX Open Client support library, *Progress-Install-Directory*\bin\o4glActiveX.dll, to any directory on the system.

d) If you do not have Version 5.00.3182 or later of the Microsoft JVM on your machine, run msjavx86.exe to install the Microsoft JVM and related Java classes. You can find msjavx86.exe on the Progress installation CD or you can obtain it from the Microsoft Web site.

2 ♦ Update the CLASSPATH environment variable to include the Open Client Runtime package you copied in Step 1.

3 ♦ Register the proxy by running Microsoft's RegSvr32.exe on the proxy .dll file.

4 ♦ Register the ActiveX Open Client support library by running Microsoft's RegSvr32.exe on the o4glActiveX.dll file (located in *Progress-Install-Directory*\bin or in the directory where you copied this file in part c of Step 1).

5 ♦ Write, compile, and execute the client application (Automation Controller). For more information, see Chapter 4, "Programming Concepts," and Chapter 7, "Programming ActiveX Controller Clients."

2.4 Deploying an Open Client Application

After completing the steps in "Building an Open Client Application", you can deploy your client application according to its Open Client type, as specified in the following sections.

2.4.1 Deploying a 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 ♦ 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 in ProxyGen during the generation of the proxy. The package name implies a directory structure that you must maintain when you copy the proxy. You can also put the class files into a .zip or .jar file, as with any other Java classes.

c) 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 the in the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

d) Copy the Open Client Runtime package to any directory on your system. For more information, see the "Selecting an Open Client Runtime Package" section.

2 ♦ If you are using HTTPS, copy your digital certificates to the location required by the Open Client application. For more information, see the "Managing Digital Certificates" section.

3 ♦ Update the CLASSPATH setting for the Java environment that will run the client application and proxy to include the:

a) Client application classes.

b) Directory containing the proxy (`.class` files) or the `.zip` or `.jar` file. Remember that for class files, this is the directory above the package hierarchy.

c) Open Client Runtime package that you copied in Step 1.

NOTE: The mechanism to update the CLASSPATH setting depends on the Java environment that you are using. For instructions on updating the CLASSPATH, see the documentation for your Java environment.

4 ♦ Run your Java Open Client application as designed.

2.4.2 Deploying a Client Application That Uses an ActiveX Proxy

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

1 ♦ Copy the required files for the client application and proxy to the deployment system:

a) Copy the client application.

b) Copy the proxy (`.dll` file) to any directory on the system.

c) Copy the Open Client Runtime package to any directory on the system. For more information, see the "Selecting an Open Client Runtime Package" section.

d) Copy the ActiveX Open Client support library (*Progress-Install-Directory*\bin\o4glactivex.dll) to any directory on the system.

2 ♦ If you are using digital certificates, copy your digital certificates to the location required by the Open Client application. For more information, see the "Managing Digital Certificates" section.

3 ♦ Update the CLASSPATH environment variable to include the Open Client Runtime package you copied in Step 1.

4 ♦ If you do not have Version 5.00.3182 or later of the Microsoft JVM on your machine, run `msjavx86.exe` to install the Microsoft JVM and related Java classes. You can find `msjavx86.exe` on the Progress installation CD or you can obtain it from the Microsoft Web site.

5 ♦ Register the proxy by running Microsoft's `RegSvr32.exe` on the proxy `.dll` file.

6 ♦ Register the ActiveX Open Client support library by running Microsoft's `RegSvr32.exe` on the `o4glActiveX.dll` file.

7 ♦ Run your ActiveX Open Client application as designed.

2.5 Managing Digital Certificates

When a client connects to an SSL enabled Web server, the Web server sends its digital certificate to the client to prove its identity. The client is responsible for authenticating that identity. Authentication is done using Root Digital Certificates. The client does this by locating the local Root Digital Certificate on the client machine that corresponds to the Web server certificate issued by the CA for the Web server. This local certificate is then validated against the certificate sent by the Web server to authenticate the identity of the Web server.

To meet the demands of the worldwide software distribution that Progress Software supports, a set of international CA Root Digital Certificates is distributed with the Progress installation. Though these Root Digital Certificates can be distributed and used as is, the size might make it impractical to use. For example, you might not want to use these Root Digital Certificates with applets due to the download time required.

Progress Software recommends that your application or applet be distributed with a minimal set of Root Digital Certificates from the set provided. You can then provide the application or applet deployer the capability of adding one or more Root Digital Certificates with the `setCertificateStore()` method in order to satisfy their specific requirements. This provides the deployer with a way to develop their specific method of distributing their own Root Digital

Certificates to their users and then dynamically configuring the Open Client application or applet to use them.

NOTE: If the Web server is configured to use a Digital Certificate issued by a private Certificate Authority (CA), it would never be included in the set distributed by Progress.

The compressed file names of the Root Digital Certificates do not indicate which certificates are included. However, Progress provides a certificate management utility (`procertm.exe`) that you can use for viewing and managing the files. For more information about the certificate management utility, see Appendix B, "Certificate Management Utility."

The following table lists the Open Client application type, run-time environments, and Root Digital Certificate packages that Progress distributes.

Table 2–3: Root Digital Certificate Packages

If you have....	Progress supplies this root digital certificate package..
An ActiveX or Java application	`psccerts.zip`
Java applet running in the Netscape browser only	`Psccertsn.jar`
Java applet running in the Internet Explorer browser only	`Psccerts.zip`

In all cases, the Root Digital Certificates that Progress distributes are in binary (DER) format. All of the certificate files use compressed file names and have `.cer` file extensions. The exception is for the Netscape Internet Browser where the Root Digital Certificates are in files with a `.txt` file extension.

3

Generating and Managing Proxies

This chapter provides a general guide to each step required to define and generate proxies for your Java or ActiveX Open Client application. This chapter also describes how to use the Open Client Proxy Generator (ProxyGen) tool. For more information on the options available in ProxyGen, see the ProxyGen online help. If you only need to access Progress SmartDataObjects on the AppServer (and no other 4GL procedures) from a Java application, you do not need to use ProxyGen. Instead, use the prebuilt SDOAppObject proxy to access the SmartDataObjects. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."

The sections in this chapter describe:

- Overview and Getting Started With ProxyGen
- Defining Your Proxy Using ProxyGen
- Changing the Propath Setting Within a Project
- Object Naming In ProxyGen
- Proxy Validation and Generation Rules
- Managing Proxies During AppServer Development
- Using GUI and Batch ProxyGen Execution Options

For information about writing Java and ActiveX applications using Open Client proxies, see Chapter 4, "Programming Concepts," Chapter 5, "Programming Java Clients," and Chapter 7, "Programming ActiveX Controller Clients." For information about writing Java applets, see Chapter 4, "Programming Concepts," Chapter 5, "Programming Java Clients," and Chapter 8, "Special Considerations For Java Applets."

3.1 Overview and Getting Started With ProxyGen

ProxyGen comes installed in two versions, a GUI tool that runs on Windows platforms and a batch version that runs on either Windows or UNIX. You must use the GUI version to define a proxy that accesses AppServer 4GL procedures. You can use either the GUI or batch version of ProxyGen to generate the actual proxy.

Thus, the basic tasks that you can complete with GUI ProxyGen include:

1. Defining and editing a proxy
2. Specifying proxy generation options
3. Saving the proxy definition in a project file
4. Validating and generating the proxy

You can only use Batch ProxyGen to complete the fourth task, validating and generating the proxy.

NOTE: While this chapter provides basic guidance on how to use the ProxyGen tool, see the online help for complete information on each ProxyGen window. Also, for information on additional options for starting ProxyGen, see the "Using GUI and Batch ProxyGen Execution Options" section.

3.1.1 Starting and Using GUI ProxyGen

You can start GUI ProxyGen on Windows by opening the Proxy Generator icon in the Progress program group:

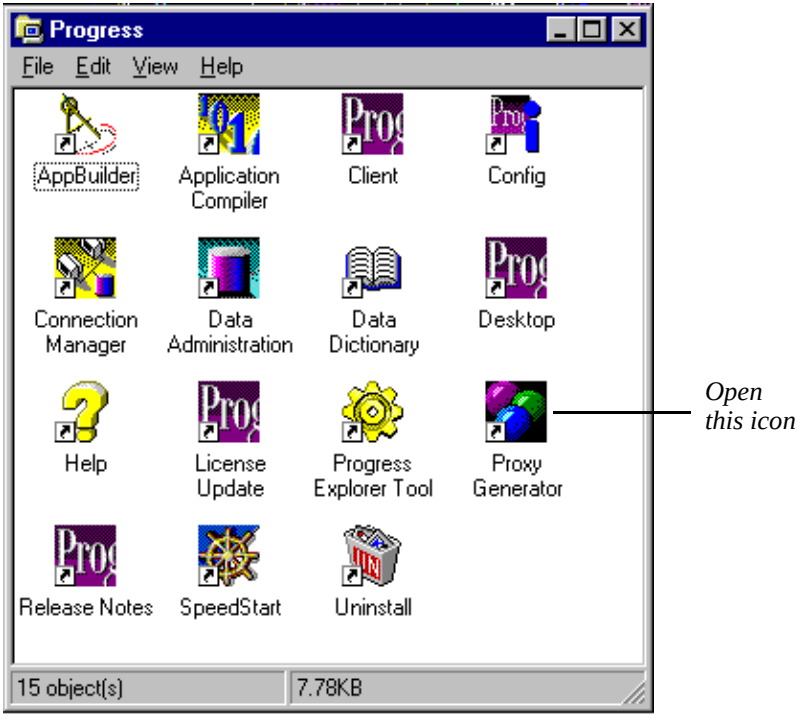

Figure 3–1: Starting GUI ProxyGen On Windows

This opens the ProxyGen main window, like the example shown in Figure 3–2.

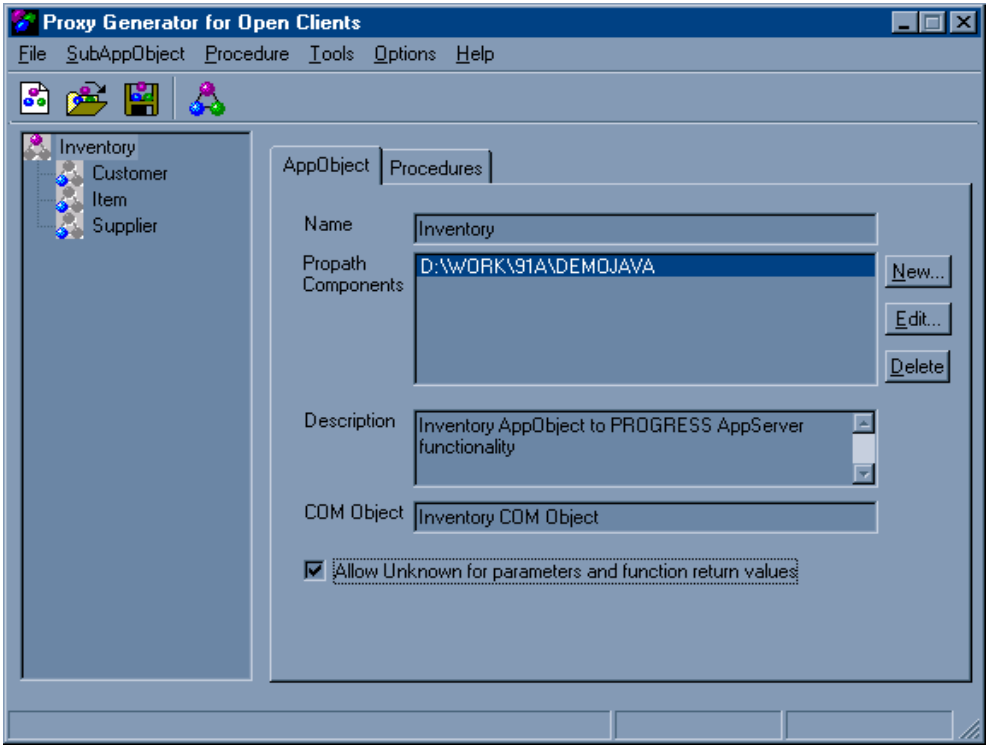

Figure 3–2: ProxyGen Main Window

The tree control in the figure shows a previously-defined proxy consisting of the Inventory AppObject and three SubAppObjects-Customer, Item, and Supplier. The Inventory AppObject is selected with its definition showing in the AppObject tab folder.

The following sections provide information about:

- Defining and Editing a Proxy
- Specifying Generation Options
- Saving a Project File
- Validating and Generating a Proxy
- Running Batch ProxyGen

3.1.2 Defining and Editing a Proxy

The ProxyGen main window provides a tree control to navigate through the proxy objects as you define them. The root (top) object in the tree control is always the AppObject for the current proxy (for example, Inventory in Figure 3–2). Any child nodes in the tree are SubAppObjects associated with this AppObject (for example, Customer in Figure 3–2). Depending on your selection in the tree control, ProxyGen displays tab folders that show the definition for the selected object. You complete the proxy definition by entering information in the tab folders and using the options available from the menu bar and toolbar.

To edit an existing proxy, you open the *project file* (with extension `.pxg`) containing the proxy definition. You can do this using the ProxyGen File menu or by opening the project file directly in the Windows Explorer.

3.1.3 Specifying Generation Options

Among the options that you can specify to complete a proxy definition are a set of options that determine how you want to generate the proxy. This information becomes part of the proxy definition and is specified using ProxyGen. You can set and change these generation options at several points during a ProxyGen session. For more information, see the "Validating and Generating the Proxy" section.

3.1.4 Saving a Project File

The minimum output from ProxyGen is a proxy definition that you save in a *project file* (with extension `.pxg`). You have the option to save the project file either explicitly or when you validate or generate the proxy. Once you save the project file, you can use either GUI ProxyGen or Batch ProxyGen to generate the actual proxy. For more information on saving project files, see the "Saving the Proxy Definition In a Project File" section.

3.1.5 Validating and Generating a Proxy

Proxy generation validates the information in the project file against the AppServer r-code and generates the proxy. If you make subsequent changes to the AppServer code after the project file is saved, ProxyGen automatically incorporates some of the changes in the next version of the proxy you generate based on a set of rules.

If this automatic update does not produce acceptable results, you must run the GUI version of ProxyGen and update the information in the project file manually. For example, as AppServer procedure files are created or deleted, you must update the definition of the AppObject in ProxyGen. Then, you can generate the proxy from the newly saved project file using either version of ProxyGen.

3.1.6 Running Batch ProxyGen

To run Batch ProxyGen, enter a command in the Windows Run dialog or at the UNIX command prompt using the following syntax:

```
bproxygen -pxgfile project-filename.pxg
```

This validates the proxy defined in `project-filename.pxg` and generates the proxy. For more information on Batch ProxyGen execution options, see the "Using GUI and Batch ProxyGen Execution Options" section.

Differences On UNIX and Windows Platforms

Batch ProxyGen runs on UNIX in the same way as it runs on Windows platforms, with the following limitations:

- It works only for projects that define Java proxies and that do not also define ActiveX proxies. You can only generate ActiveX proxies on Windows platforms.
- It ignores all drive letter references in project files generated by ProxyGen on Windows platforms. As a result, all absolute pathnames in the project descend from root (/).
- It automatically changes back slashes (\) to forward slashes (/). You can also use dot (.) as a Propath component.

3.2 Defining Your Proxy Using ProxyGen

You can follow these steps in the GUI version of ProxyGen to expose AppServer functionality as a proxy:

1 ♦ Define an AppObject for the proxy, and optionally one or more SubAppObjects.

2 ♦ Select 4GL procedures (non-persistent and persistent) to include in these object definitions.

3 ♦ Optionally, customize access to the selected procedures in the proxy.

4 ♦ Save the proxy definition in the project file.

5 ♦ Validate and generate the proxy.

The following sections provide an overview of these steps. For specific information on ProxyGen options for defining and generating proxies, see the ProxyGen online help.

The following sections provide information about:

- Defining an AppObject Or SubAppObject
- Specifying 4GL Procedures and User-defined Functions
- Customizing Procedure Access
- Saving the Proxy Definition In a Project File
- Validating and Generating the Proxy

3.2.1 Defining an AppObject Or SubAppObject

You must define one AppObject for a proxy. You can also define one or more SubAppObjects depending on how you choose to organize the available AppServer functionality. As you specify AppObject and SubAppObject names and select 4GL procedures to define the proxy, ProxyGen applies a set of rules to convert and validate the various object and method names that you specify. For more information on these naming rules, see the "Object Naming In ProxyGen" section.

The following sections provide information about:

- Creating AppObject and SubAppObject Entries
- Specifying AppObject and SubAppObject Definitions

Creating AppObject and SubAppObject Entries

To create a new AppObject definition, choose File→ New from the ProxyGen menu bar or the New button from the ProxyGen toolbar. ProxyGen closes any open project, and creates a project for the new proxy, adding an unnamed AppObject entry at the root of the navigation tree control. ProxyGen also displays the tab folders for this AppObject with the entry selected in the tree control.

To create a SubAppObject, choose SubAppObject→ New from the menu bar. ProxyGen adds an unnamed SubAppObject entry to the navigation tree control and selects it, displaying the tab folders to define the SubAppObject.

The tab folders for defining both the AppObject and any SubAppObjects are identical. However, the definitions that you enter apply only to the entry currently selected in the tree control. Also, as you select an entry in the tree control, the field values in the AppObject and Procedures tab folders change to the current settings for the selected AppObject or SubAppObject.

Specifying AppObject and SubAppObject Definitions

As part of defining an AppObject or SubAppObject, you must specify the following information in the AppObject tab folder:

- The proxy object name (Name field)
- The PROPATH (Propath Components list box), set to the locations of AppServer procedures for this proxy object. (See the "Specifying 4GL Procedures and User-defined Functions" section.)

You can optionally specify:

- Proxy object descriptive information (Description edit box)
- If generating an ActiveX proxy, the name of the Automation object (COM Object field, used for example in Visual Basic's Project References dialog box)
- Whether to allow access to the 4GL unknown value (?) in parameters and return values (Allow Unknown field). The check box here applies to the entire proxy object. Other dialog boxes can be used to allow access to the 4GL unknown value for individual procedures and parameters.

  NOTE: If you are using the proxy to access a remote Progress SmartDataObject, you must allow 4GL unknown values for the entire proxy object. Check this box and make sure that the boxes in all ProcObject and method definitions remain checked.

You must also specify any AppServer procedure files you want to include, and whether they are non-persistent or persistent procedures (ProcObjects), using the Procedure menu on the menu bar. These procedures are listed on the Procedures tab folder. For more information on specifying the PROPATH setting and AppServer procedure files, see the "Specifying 4GL Procedures and User-defined Functions" section.

3.2.2 Specifying 4GL Procedures and User-defined Functions

To specify 4GL procedures for an AppObject or SubAppObject definition, you must:

1 ♦ Set the Propath Components list box to allow ProxyGen to locate the available r-code files.

2 ♦ Add the r-code files for this object from the set of files made available under the directories specified as Propath Components. Add each procedure according to whether it is executed non-persistently or persistently.

3 ♦ Optionally, customize each procedure entry to configure the corresponding method (if non-persistent) or ProcObject (if persistent).

4 ♦ Optionally, for ProcObjects only, customize the method definitions for all specified internal procedures and user-defined functions.

The following sections provide information about:

- Setting the Propath Components List Box
- Adding and Deleting Procedure Files
- Changing the Execution Mode (Type)
- Customizing Method and ProcObject Definitions
- Customizing Method Definitions In a ProcObject

Setting the Propath Components List Box

To access the procedures for AppObject and SubAppObject definitions, you must first specify, in the Propath Components list box, the directories that contain the r-code for these procedures. Typically, these directories correspond to a portion of the AppServer PROPATH environment setting. You can add, edit, or remove a Propath Component using the New, Edit, and Delete buttons on the AppObject tab folder.

You can only add 4GL procedures to a proxy that ProxyGen can locate using the Propath Components setting. The PROPATH environment variable itself might be different between the development and deployment machines, but the relative path for the r-code files below a specified PROPATH directory must remain unchanged. Thus, if you move the r-code files to a separate development machine for access by ProxyGen, make sure that you maintain the directory structure used for relative pathnames under the Propath Components setting.

Adding and Deleting Procedure Files

You can add any 4GL procedure file to an AppObject or SubAppObject definition, provided that the compiled (r-code) version of the file is available to ProxyGen through the Propath Components setting. For information on compiling 4GL procedures, see the *Progress Programming Handbook*.

NOTE: If you are generating a Java proxy that accesses a Progress 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 Chapter 6, "Using SmartDataObjects From Java Clients." However,

if you are defining a ProcObject in ProxyGen to directly access the SmartDataObject as a Progress persistent procedure, you must make the compiled r-code available to ProxyGen, like any other 4GL procedure.

To add procedure files to your object definition:

- **For non-persistent execution** — Choose Procedure→ Add→ Non-persistent from the menu bar.
- **For persistent execution** — Choose Procedure→ Add→ Persistent from the menu bar.

These choices display either the Add Procedures dialog box or the Add Persistent Procedures dialog box for your current proxy object. The user interface is almost identical for the two dialog boxes. Both dialog boxes allow you to:

- Select r-code files under a pathname selected from your Propath Components setting (Propath Selection list box).
- Specify whether to use the relative pathname of selected procedures as part of the corresponding method name in the proxy (Use Path in Method Name check box).
- Request that any directories that you select be searched recursively (Perform Recursive Add check box).
- Add the selected files to the Procedures tab folder for the current AppObject or SubAppObject definition (Add button). The procedures that you add depend on what files and directories you have selected in the Folders tree and Folder Contents list.

The difference between the two dialog boxes is in the execution type of the procedure file that is added. A method is generated in the proxy for each non-persistent procedure. A ProcObject class factory method is generated for each persistent procedure. (For information on proxy methods, see Chapter 1, "Overview," and Chapter 4, "Programming Concepts.")

After selecting and adding the respective procedure files, any non-persistent procedures appear in the Procedures list and any persistent procedures appear in the Persistent Procedures list of the Procedures tab folder on the main window.

To delete procedures from either list in the Procedures tab folder:

1 ♦ Select one or more procedures in one list.

2 ♦ Press the **Delete** key or choose Procedure→ Delete from the menu bar.

Changing the Execution Mode (Type)

You can change the execution mode type of any procedure files already added to a proxy object definition using the Procedures tab folder. This folder allows you to select and move procedures between non-persistent execution (Procedures list) and persistent execution (Persistent Procedures list).

To move procedures between modes:

1 ♦ Select one or more procedures in one list.

2 ♦ Choose the associated arrow button or Procedure→ Change Type from the menu bar to move the selected procedures to the opposite list.

Customizing Method and ProcObject Definitions

You can customize method and ProcObject definitions for added procedure files according to your application requirements. From the Procedures tab folder:

1 ♦ Select a procedure file in one of the lists.

2 ♦ Double-click the selection or choose Procedure→ Customize from the menu bar.

This displays the corresponding Customize Procedure or Customize Persistent Procedure dialog box for your proxy object. Both dialog boxes provide:

- A Procedure tab folder to modify the name and description of the associated method or ProcObject
- A Parameters tab folder to modify the behavior of method or ProcObject parameters
- A check box (Use AppObject Unknown Setting) that specifies whether to use the AppObject or SubAppObject setting to allow the 4GL unknown value (?) in parameters and return values

The Customize Persistent Procedure dialog box also has an Internal Procs tab folder to customize methods (internal procedures and user-defined functions) of the specified ProcObject.

In the Procedure tab folder, you can:

- Rename the method or ProcObject (Method Name or ProcObj Name field).
- Enter descriptive information for the method or ProcObject (Description edit box).

- For ActiveX proxies, specify a label to identify the method or ProcObject in the Type Library.

In the Parameters tab folder, you can set check boxes to indicate whether one or more method or ProcObject parameters allow the 4GL unknown value (?). If you check the Use AppObject Unknown Setting box, these parameter check boxes are disabled.

In the Internal Procs tab folder for ProcObjects (Customize Persistent Procedure dialog box):

- You can set check boxes to indicate whether one or more internal procedures or user-defined functions are included as methods of the ProcObject.
- You can also customize the method definition for a selected internal procedure or user-defined function included in the ProcObject (Customize button).

Customizing Method Definitions In a ProcObject

You can customize the definitions for methods included in a ProcObject according to your application requirements. From the Customize Persistent Procedure dialog box:

1 ♦ Select an included internal procedure or user-defined function from the Internal Procedures/User-Defined Functions list.

2 ♦ Double-click the selection or choose the Customize button.

The Customize Internal Procedure/UDF dialog box appears. This dialog box provides:

- A Procedure tab folder to modify the name and description of the method associated with the selected internal procedure or user-defined function.
- A Parameters tab folder to modify the behavior of any method parameters
- A check box (Use AppObject Unknown Setting) that specifies whether to use the AppObject or SubAppObject setting to allow the 4GL unknown value (?) in method parameters and return values
- For user-defined functions, a check box (Allow Unknown Return) that specifies whether to allow the 4GL unknown value (?) in function return values.

In the Procedure tab folder, you can:

- Rename the method (Method Name field).
- Enter descriptive information for the method (Description edit box).
- For ActiveX proxies, specify a label to identify the method in the Type Library.

In the Parameters tab folder, you can set check boxes to indicate whether one or more method parameters allow the 4GL unknown value (?). If you check the Use AppObject Unknown Setting box, these parameter check boxes are disabled.

3.2.3 Customizing Procedure Access

To manage client access to remote procedures, ProxyGen supports three levels of access control for remote procedures:

1. The AppServer developer can define internal procedures and user-defined functions as PRIVATE in the 4GL. ProxyGen does not display private procedures and user-defined functions for use in a ProcObject definition.
2. You can further select what public internal procedures and user-defined functions you want ProxyGen to include in a ProcObject.
3. The AppServer developer can use the business logic on the AppServer to further restrict what procedures can be run remotely by using the Progress EXPORT statement. This statement restricts access at run time without any affect on the generated proxy.

For information on how the second level of remote procedure access control works in ProxyGen, see the "Specifying 4GL Procedures and User-defined Functions" section. For information on how AppServer applications can use the EXPORT statement to control remote procedure access at run time, see *Building Distributed Applications Using the Progress AppServer*.

3.2.4 Saving the Proxy Definition In a Project File

At any point during definition of an AppObject or SubAppObject, you can create or save the project (`.pxg`) file that defines the proxy. This file includes the list of all selected AppServer procedures and any customizations you have applied to those procedures for each proxy object definition.

Setting the Project Work Directory

By default, ProxyGen saves all project files in a *work directory*. You specify the default path for this work directory during Progress installation of the Open Client Toolkit. The Progress install then saves the path for the ProxyGen working directory in the `proxygen.prm` file and that path is then used at the default.

The `proxygen.prm` file is located in the directory where you installed Progress. You can view `proxygen.prm` with a text editor and change the ProxyGen work directory. For more information, see the "GUI ProxyGen" section.

NOTE: You cannot set the project directory using the Start in option in the ProxyGen shortcut properties. Changes to the Start in option have no effect on ProxyGen operation.

ProxyGen allows you to specify a separate directory to save proxies and log files. For more information, see the "Validating and Generating the Proxy" section.

Saving the Project File

To save the project file, choose File→ Save from the menu bar. ProxyGen opens a Save As file dialog box in the current project directory. You can save the project file in this directory or select a different project directory to save the file. Each time during the same ProxyGen session that you save the project file, the Save As file dialog box opens to the last directory that you selected as the project directory.

Also, if the preference settings specify Save on Validate/Generate, ProxyGen automatically saves the project file to the current project directory each time you validate or generate a proxy. For information on setting preferences, see the "Validating and Generating the Proxy" section.

3.2.5 Validating and Generating the Proxy

ProxyGen allows you to generate and validate a proxy in one step or validate the proxy definition without generating the proxy. At almost any point during proxy definition, but before proxy validation or generation, you can specify options for proxy generation. These options specify such information as an Application Service name for the AppServer connection and compiler options for proxy generation.

Specifying Proxy Generation Preferences

You can specify proxy generation options at the default or AppObject level. You can specify the same options at each level as follows:

1. **Default** — Default settings for all proxies that you define. Choose Options→ Preferences from the menu bar, and select the Default tab folder in the displayed Preferences dialog box. ProxyGen initializes these settings from and saves them to the *Progress-Install-Directory*\properties\ProxyGen.preferences file.

2. **AppObject** — Specific settings for the AppObject you are defining. ProxyGen provides two ways to access this information:

   a) Choose Options→ Preferences from the menu bar, and select the AppObject tab folder in the displayed Preferences dialog box.

   b) Choose File→ Generate from the menu bar or choose the Generate button from the toolbar. This displays the Generate dialog box for the current AppObject before you generate the proxy.

   ProxyGen initializes these settings for a new AppObject from the ProxyGen.preferences file. Thereafter (for an existing AppObject), ProxyGen initializes these settings from and saves them to the project file for the proxy.

Setting the Proxy Directory

ProxyGen places both the proxy files (.class and .dll) and the activity log (*AppObject-name*.log) in a *proxy directory*. You can set the proxy directory to a different path by changing the Proxy Dir option in the Preferences dialog box or the Generate dialog box.

NOTE: ProxyGen does not clean the proxy directory. If there are existing files in the directory with the same name as a generated file, ProxyGen overwrites the existing files. Similarly, ProxyGen ignores any obsolete proxy files, which you can remove manually from the directory.

Specifying the Compiler

ProxyGen uses two default compilers, JavaSoft's javac (installed by Progress) and Microsoft's jvc. You can also specify an alternate compiler for proxy generation.

For Java proxies, you can choose from the following compilers:

- Default javac
- Default jvc
- Custom

For ActiveX proxies, you can choose from the following compilers:

- `Default (jvc)`
- `Custom (jvc)`

Specifying Compiler Options

You can use the Advanced Compiler Options dialog box to specify the following compiler options for both Java and ActiveX compilers:

- **Classpath switch** — Use to customize the Classpath.
- **Classpath** — The Classpath value.

  CAUTION: The Classpath initially displayed in this field is the one required for proxy generation. You should never change this portion of the Classpath. You should only add values to the end of this Classpath.

- **Classpath settings for Java Open Clients**
  - For Microsoft Java SDK, `progress.jar`, `messages.jar`, and `prowin.jar` are part of the Classpath.
  - Sun-Compatible JDK, `progress.jar` is part of the Classpath.
- **Compiler options** — All options and values accepted by the complier other than the Classpath.

Proxy Validation and Generation

Once a project file is saved, you can generate the proxy by choosing File→ Generate from the menu bar or the Generate button from the toolbar (which displays the Generate dialog box), then choosing the OK button. During generation, the list of procedures is validated against the r-code files available on disk. If any procedure is deleted, ProxyGen automatically removes the procedure from the AppObject definition and records this action in an activity log file (*`AppObject-name`*.log). ProxyGen picks up all prototype changes and reconciles these changes (such as new or removed parameters) with existing customizations. ProxyGen records all such significant reconciliations in the activity log. This continues until all the procedures have been validated. For the available r-code files, the proxy is then generated with any specified customizations.

CAUTION: On Windows platforms, a compiler error might result if you generate a Java proxy, change the case of an AppObject, SubAppObject, or ProcObject name, and then regenerate the proxy without first deleting the old proxy files.

Proxy Validation Without Generation

If you want, you can also validate the proxy before any attempt to generate the code by choosing File→ Validate from the menu bar or the Validate button from the toolbar. This causes ProxyGen to complete the same validation procedure that occurs during code generation without generating any code. Similarly, any significant validation actions appear in the activity log.

For more information on proxy validation and generation, see the "Proxy Validation and Generation Rules" section.

3.3 Changing the Propath Setting Within a Project

You might find it necessary to change the Propath setting for existing procedures of an AppObject or SubAppObject. You can accomplish this by deleting and adding these procedures again within your project. However, this results in the loss of any customizations. To accomplish this without losing data, the Open Client Toolkit provides the Change Propath tool.

On Windows platforms, you can use this tool through the Change Propath dialog box in GUI ProxyGen. On non-Windows platforms, such as UNIX, you can use the tool by running the `ChangePropath` utility. `ChangePropath` is a Java utility installed with the Open Client Toolkit that you execute with the following syntax:

SYNTAX

```
java com.progress.open4gl.proxygen.ChangePropath project-file propath-file
```

`project-file`

The project (.pxg) file for the proxy whose Propath settings you want to change.

`propath-file`

A text file containing a pair of strings for each Propath component you want to change, using the following syntax, where *`OldPropathDirectory`* is the current Propath component string and *`NewPropathDirectory`* is the replacement component string:

SYNTAX

```
{ OldPropathDirectory newline
  NewPropathDirectory newline }
{ OldPropathDirectory newline
  NewPropathDirectory newline }
...
```

NOTE: Each Propath component string must reside on a separate line

Thus, suppose you have a proxy whose project file is `Employees.pxg`. You might create a text file named `Propath.txt` with the following content:

Propath.txt

```
/appsv1/SimpleQueries
/appsv2/SimpleQueries
/appsv1/ComplexQueries
/appsv2/ComplexQueries
```

Then run the following command:

```
java com.progress.open4gl.proxygen.ChangePropath Employees.pxg Propath.txt
```

The component `/appsv2/SimpleQueries` replaces `/appsv1/SimpleQueries` and `/appsv2/ComplexQueries` replaces `/appsv1/ComplexQueries` in the Propath for every procedure included in `Employees.pxg`

NOTE: This utility is not recommended for use with Windows operating systems, even though it is accessible on these platforms. On Windows platforms, use the Change Propath dialog box in GUI ProxyGen instead.

3.4 Object Naming In ProxyGen

Proxy generation requires the mapping of 4GL identifier names to names appropriate for the proxy. This includes restricting the use of certain characters and reserved words.

ProxyGen determines the default names of proxy methods and parameters from the r-code filename and its contents. These names conform to the proxy naming conventions and automatic conversions described in the following sections.

NOTE: You can override the default names for proxy methods in GUI ProxyGen.

3.4.1 Proxy Naming Conventions

ProxyGen uses the following default naming conventions for proxy objects and methods:

- **AppObject, SubAppObject, and ProcObject class names**

  You must specify the name for each AppObject and SubAppObject class. Unless you override it, each ProcObject class name is the r-code filename of the mapped persistent procedure, matching the case and spelling without the extension. ProxyGen also performs any automatic name conversions.

  Each object must be uniquely named.

  For Java proxies, these object names are used as the actual Java class names.

  For ActiveX proxies, these object names are preceded by a prefix that you see in the VB Object Browser for each class as follows:

  – **AppObject** — *CAppObject*, where *AppObject* is the AppObject name.

  – **SubAppObject** — *CAO_SubAppObject*, where *SubAppObject* is the SubAppObject name.

  – **ProcObject** — *CPO_ProcObject*, where *ProcObject* is the ProcObject name.

- **Methods in an AppObject or SubAppObject that execute an external, non-persistent AppServer procedure**

  The method name is the procedure filename without the extension. Method names are case sensitive. ProxyGen also performs automatic name conversions. See the "Automatic Name Conversions" section.

- **Methods in a ProcObject that execute an internal procedure or user-defined function defined in the persistent procedure**

  The method name match the corresponding 4GL names found in the procedure file. Method names are case sensitive. ProxyGen also performs any automatic name conversions.

- **Class factory methods in an AppObject that create a SubAppObject**

  The method name depends on the type of Open Client:

  – **Java** — `createAO_`*`SubAppObject`*

  – **ActiveX** — `CreateAO_`*`SubAppObject`*

  `SubAppObject` is the SubAppObject name.

- **Class factory methods in an AppObject or SubAppObject that create a ProcObject**

  The method name depends on the type of Open Client:

  – **Java** — `createPO_`*`ProcObject`*

  – **ActiveX** — `CreatePO_`*`ProcObject`*

  `ProcObject` is the ProcObject name.

NOTE: On Windows platforms, the case of the r-code filename is affected by the name entered during 4GL compilation. For example if you execute the following COMPILE statement in the Progress Procedure Editor:

```
COMPILE aBCdEf.r SAVE.
```

The generated r-code filename is `aBCdEf.r`, even though `Abcdef.p` might be the actual procedure filename. Thus, the generated method name is `aBCdEf` or `CreatePO_aBCdEf` for a persistent procedure.

3.4.2 Automatic Name Conversions

During name generation, ProxyGen does some automatic conversion that capitalizes the character following each restricted character, and removes all restricted characters from the name. The restricted characters include:

- Pound (#)
- Ampersand (&)
- Percent (%)
- Dash ()
- Dot (.)

NOTE: Once you edit a method name, ProxyGen no longer does this automatic conversion. If your edit includes a restricted character, ProxyGen displays an error and you must edit the name again.

3.4.3 Naming Side Effects and Restrictions

Automatic name conversions can cause two methods or parameters in the same name space to acquire the same name, even when the original names in a procedure file are different. This can also occur if you have two procedure files with the same name under different directories. You can override method names in ProxyGen to handle method name conflicts, but you must change any conflicting parameter names directly in the procedure files because you cannot customize parameter names in ProxyGen.

When ProxyGen generates a proxy, a separate `.java` file is created for each AppObject, SubAppObject, and ProcObject. In order to avoid file conflicts, ProxyGen does not allow you to generate a proxy if any two objects in the proxy have the same name.

For example, if you have two persistent procedures with the same name but in different directories, you must customize the ProcObject name for at least one of these procedures to prevent a filename conflict. You can do this by editing the name or by including the relative path in the name. Similarly, you must ensure that your AppObject, SubAppObject, and ProcObject names are all unique within a single proxy.

3.5 Proxy Validation and Generation Rules

Proxy generation takes the proxy definition and performs the following operations to generate the requested proxy:

- **Proxy Validation**

  The proxy definition is reconciled against the actual r-code on the system because the r-code can change independently of the saved proxy definition. This process throws away obsolete information and picks up new information while preserving customizations that are still valid with regard to these changes. This process generates an activity log (`.log`) file that records any significant validation events. This file is created in the proxy directory specified in the generation preferences.

- **Proxy Generation**

  Once the proxy definition is validated and updated as appropriate, the proxy can be generated. A separate proxy is created for Java and ActiveX clients. The proxy for the Java client produces a set of `.java` and `.class` files. The proxy for the ActiveX client produces a `.dll` that defines the specified Automation objects, including the Type Library information for the proxy. In both cases, the activity log records any significant generation events and errors.

3.5.1 Proxy Validation

Proxy validation occurs when:

- You choose File→ Validate from the ProxyGen menu bar or the Validate button from the toolbar.
- You choose File→ Generate from the ProxyGen menu bar or the Generate button from the toolbar.
- You run Batch ProxyGen on the saved project file.
- You edit a method or ProcObject definition (procedure or persistent procedure). Validation occurs only for the edited procedure.

Proxy validation does the following:

- Saves the proxy definition to the project file if the preference settings specify Save on Validate/Generate.
- Verifies that the selected procedure files exist on disk.
- Compares any customizations made for procedures against the actual r-code and synchronizes them according to the following rules:
  - If a parameter for a procedure in the project file matches a parameter in the r-code by name and data type, any customizations for the parameter are maintained. Otherwise the parameter is assumed to be new and the default setting is used for allowing the 4GL unknown value.
  - If an internal procedure or user-defined function in the project file matches an internal procedure or user-defined function in the r-code by name and type (where type is whether it is a procedure or function), any customizations are maintained. Otherwise the internal procedure or user-defined function is assumed to be new and is assigned all default definition settings (method name, description, and so on).

- Validates the method and parameter names and verifies that all method names are unique within each proxy object and that parameter names are unique for each method.

  NOTE: While you can override method names in ProxyGen, you cannot override parameter names.

- Verifies that all object names are unique within the proxy.

- Logs discrepancies, errors, and reconciliations to the activity log.

3.5.2 Proxy Generation

ProxyGen generation performs the following actions, putting any output into the proxy directory that you specify in the generate preferences:

- Saves the proxy definition to the project file if the preference settings specify Save on Validate/Generate.

- Performs proxy validation if it has not already been done.

- For a Java Client, generates implementation and delegating classes for each proxy object as follows:

  NOTE: For more information on these two types of classes, see Chapter 5, "Programming Java Clients."

  – Generates two Java classes for the AppObject. For example, if the AppObject was named Inventory, these classes would be `Inventory.class` and `InventoryImpl.class`. The source for the public (delegating) class is also available (`Inventory.java`).

  – Generates two Java classes for each SubAppObject. For example, if the SubAppObject was named Item, these classes would be `Item.class` and `ItemImpl.class`. The source for the public (delegating) class is also available (`Item.java`).

  – Generates two Java classes for each ProcObject. For example, if the ProcObject was named AddItem, these classes would be `AddItem.class` and `AddItemImpl.class`. The source for the public (delegating) class is also available (`AddItem.java`).

  – Generates an activity log with status and error information (`Inventory.log`).

- For an ActiveX Client:
  - Generates the Type Library and Automation objects for each proxy object in a single `.dll` file.
  - Generates an activity log with status and error information.

3.6 Managing Proxies During AppServer Development

You can run ProxyGen either at the end of a development life cycle or as part of a build process. As AppServer procedure files are created or deleted, you must update the definition of the proxy in ProxyGen. If a procedure file is modified, you might need to update the proxy if any modifications you previously made in ProxyGen no longer makes sense.

3.6.1 Replacing a Lost Project File

If you lose an existing project (`.pxg`) file and have to use ProxyGen to recreate it and regenerate an ActiveX proxy again, you must rebuild any Visual Basic or C++ executable that uses the generated proxy (`.dll`) file.

When you create a new project in ProxyGen, it generates a new class ID that it stores in the proxy file generated for the project. The ActiveX Controller references the proxy using the same class ID. If you try to use the newly-generated proxy file with the existing executable, the two class IDs will not match and the ActiveX Controller fails trying to locate the proxy during execution.

For any proxy that you generate with an existing project, rebuilding the associated ActiveX Controller executable is not necessary. ProxyGen maintains the same class ID for all versions of the proxy that you generate from the same project unless you explicitly tell it to do otherwise.

3.6.2 A Sample Management Scenario

A Progress developer at company ABCPets designs and develops an inventory system for pet shops using the AppBuilder (.w files). These source files are in the directory `c:\ABCPets\Source\Inventory`. The files are compiled and the r-code files are placed in `c:\ABCPets\Bin\Inventory`. ABCPets wants to package this functionality for Java clients. You run GUI ProxyGen and supply the following information:

- AppObject name — `Inventory`
- Propath — `c:\ABCPets\Bin`
- Allow Unknown Values — checked
- Procedures — `Inventory\InvAdd.r`
- Persistent Procedures — `Inventory\p-Cust.r`, with internal procedures:
  - `AddInventory()`, which takes two INTEGER parameters
  - `ValidateInventory()`
  - `UpdateInternals()`, which is defined in `Inventory\p-Cust.r` as PRIVATE

You customize the method definition for `AddInventory()` to not allow the 4GL unknown value (?) for either parameter. You also explicitly exclude `ValidateInventory()`. Note that the internal procedure `UpdateInternals()` is defined with the PRIVATE keyword and does not appear in ProxyGen as a procedure to include or customize.

When you choose the Generate button, you supply the following information:

- AppService — `com.ABCPets.Inventory`
- Package — `com.ABCPets`
- Proxy Dir — `c:\Proxies`

Proxy generation places the Java proxy (the `.class` files) into `c:\Proxies\com\ABCPets`, and creates the activity log as `c:\Proxies\Inventory.log`. The generated Java files include:

- `Inventory.java`
- `Inventory.class`
- `InventoryImpl.class`
- `pCust.java` (note the removal of the dash)

- pCust.class
- pCustImpl.class.

You then change p-Cust.w but do not change the signature of AddInventory() (AddInventory(INTEGER, INTEGER)). You add new internal procedures to the persistent procedure p-Cust.w. These changes only require you to regenerate the proxy as long as you want them included. New non-PRIVATE internal procedures are added as methods automatically with default definitions. In order to exclude them, you have to run GUI ProxyGen. Also any changes in or the removal of ValidateInventory() do not cause validation errors because it is excluded from the proxy definition and is not validated.

You decide to run Batch ProxyGen as part of a regular build cycle so that any similar changes in the 4GL are automatically picked up.

Later in the development cycle, you add a new procedure file for inventory called Orders.w. This procedure file is saved in c:\ABCPets\Source\Inventory and the r-code file is placed in c:\ABCPets\Bin\Inventory. You then run GUI ProxyGen before the scheduled build and add this procedure file to the Inventory AppObject.

You make another change to the AddInventory() procedure, which now takes a third parameter (CHARACTER). Because this is a new parameter, it automatically gets the default (AppObject) setting to allow unknowns. If you want to disallow the unknown value for this parameter, you must run the GUI version of ProxyGen before the scheduled build and update the customizations for this procedure.

3.7 Using GUI and Batch ProxyGen Execution Options

This section describes options that you can use to modify operation of both the GUI and batch versions of ProxyGen.

3.7.1 GUI ProxyGen

All ProxyGen startup options are in the proxygen.prm file located in the directory where you installed Progress. You can view and edit proxygen.prm with a text editor. If you edit this file, for example to modify the work directory, make sure you do not change the format of the file. All of the /d: lines must be kept together.

A copy of an installed `proxygen.prm` file is shown below:

```
-c @{JAVA\PROGRESSCP}
wjview.exe
/d:Install.Dir=@{Startup\DLC}
/d:ProxyGen.UseMidlFromC++=no
/d:Work.Dir=@{WorkPath}
/d:JavacCP=@{Java\JDKCP}
com.progress.open4gl.proxygen.gui.ProxyGen
```

The registry references denoted by "@{...}" should not normally be changed. It is only appropriate that you customize the following lines in a ProxyGen Parameter file:

```
/d:ProxyGen.UseMidlFromC++=
/d:Work.Dir=
```

A new line for a local preference file can also be added after the `/d:Work.Dir=` line. For example:

```
/d:Work.Dir=
/d:PrefFile="c:\myfiles\proxygen.preferences"
```

You can modify the parameters shown in Table 3–1.

Table 3–1: Customizing GUI ProxyGen Startup Parameters *(1 of 2)*

Parameter	Description
`/d:Work.Dir=`*`path`*	Specifies the project work directory for ProxyGen. This is the default location for ProxyGen project files. (Defaults to the Progress work directory that you specify during installation of the Open Client Toolkit.)

Table 3–1: Customizing GUI ProxyGen Startup Parameters *(2 of 2)*

Parameter	Description
/d:ProxyGen.UseMidlFromC++=yes	A yes value tells ProxyGen to use the MIDL compiler from a Visual C++ product on the system. A no value tells ProxyGen to use MIDL from Visual J++. (Defaults to no.) For more information, see Chapter 2, "Configuration and Deployment."
/d:PrefFile=*filename*	Specifies the file for ProxyGen generation preferences. (Defaults to *Progress-Install-Directory*\properties\ProxyGen.preferences.)

3.7.2 Batch ProxyGen Platform Options

The options for Batch ProxyGen vary depending on whether the platform is Windows or UNIX.

Windows Platforms

For Batch ProxyGen, one optional command-line switch is available to generate ActiveX proxies on Windows platforms only:

```
-usemidlfromc++
```

This tells ProxyGen to use the MIDL compiler from a Visual C++ product on the system, for example:

```
bproxygen -pxgfile c:\proxy\inventory.pxg -usemidlfromc++
```

UNIX

Batch ProxyGen also runs on supported UNIX platforms. This allows you to generate code on UNIX for a proxy that you have defined in a project file created using GUI ProxyGen.

Batch ProxyGen runs on UNIX using the bproxygen command the same way it runs on Windows platforms, with the following limitations:

- It generates code only for projects that define Java proxies and that do not also define ActiveX proxies. You can only generate ActiveX proxies on Windows platforms.
- It does not support and ignores the -usemidlfromc++ command-line switch. This switch applies to ActiveX proxies only.

- It automatically converts compatible Windows pathnames to UNIX pathnames, with the exceptions described in the following paragraphs.

When you generate project (`.pxg`) files for use with Batch ProxyGen on UNIX systems, make sure all pathnames that you specify are portable to your UNIX machine. This includes the proxy directory, compiler path, and PROPATH settings.

In general, `bproxygen` applies certain translation rules when reading a project file on UNIX:

- It ignores all drive letters.
- It converts all backslashes (`\`) to forward slashes (`/`).

Using any other Windows-specific naming convention results in a non-portable project file.

To make your code more portable, you can use dot (`.`) or some other relative path as your Proxy Dir setting for all application development. You might also consider using your PATH environment variable for finding Java compiler by executable name only. Doing this also allows you to install the JDK for each platform in vendor-specified default directories without causing your application code to fail when moving from one platform to another.

4

Programming Concepts

Both Java clients and ActiveX Controller clients use equivalent mechanisms to access AppServer functionality. This chapter describes the common features of each and how they differ between the Java and ActiveX environments. For more detailed information about a specific client, see Chapter 5, “Programming Java Clients,” Chapter 7, “Programming ActiveX Controller Clients,” and Chapter 8, “Special Considerations For Java Applets.”

This chapter includes the following sections:

- Programming Overview
- Connecting To a Progress AppServer
- Obtaining Run-time Information
- Understanding Proxy Object Methods
- Using Proxy Objects
- Passing Parameters In Proxy Object Methods
- Accessing Proxy Run-time Properties
- Handling Errors In an Open Client Application

4.1 Programming Overview

In any Open Client application, you must follow these steps to access a Progress AppServer:

1 ♦ Instantiate each proxy object before accessing its methods.

2 ♦ Connect to an AppServer that supports the required Application Service by instantiating an AppObject.

3 ♦ Execute proxy methods that run remote procedures and functions on the AppServer.

4 ♦ Disconnect from the AppServer when it is no longer needed.

The remaining sections in this chapter describe how to fulfill the requirements of these steps.

4.2 Connecting To a Progress AppServer

To specify an AppServer connection for an Open Client application, you provide the same connection information as for a 4GL client application. How you provide the connection information depends on your type of client. For Java clients, you pass the connection information as a set of parameters to the AppObject constructor. For ActiveX clients, you specify the connection information as a set of parameters to a connect method on an AppObject. For more information on the connection mechanisms for a specific type of Open Client, see:

- Chapter 5, "Programming Java Clients" for Java Open Clients.
- Chapter 7, "Programming ActiveX Controller Clients" for ActiveX Open Clients.

The connection information includes character-string parameters that Progress passes to the AppServer. While the application can require any character-string information, you typically pass the following parameters:

- **AppServer URL** — AppServer connection information depends on the connection type. See "Connecting Over an Intranet" and "Connecting Over the Internet."
- **User-ID** — AppServer user identification
- **Password** — AppServer password
- **AppInfo** — Application-specific information

The Following sections provide information about:

- Connecting Over an Intranet
- Connecting Over the Internet

4.2.1 Connecting Over an Intranet

The AppServer environment provides a number of options for connecting to an AppServer over a company intranet. You can connect through a NamerServer or directly to an AppServer.

When you connect over a company intranet using a NameServer, the NameServer process coordinates access to multiple AppServers. Each AppServer identifies the application procedures that it supports using an Application Service name, which it registers with a particular NameServer. Therefore, the AppServer connection parameter provides the network address of a NameServer and the name of an Application Service. The specified NameServer locates an AppServer that supports the specified Application Service and provides the network address of that AppServer to the Progress Open Client Runtime. The Open Client Runtime then connects your application to this AppServer. Using the NameServer also provides location transparency and load balancing capabilities for an AppServer.

Connecting directly to an AppServer without a NameServer is desirable if the configuration for your application is simple and static, and as a result, you do not need location transparency or load balancing. For these configurations, taking advantage of the optional NameServer means having less to configure and manage.

Connecting directly to an AppServer also eliminates conflicts that might arise when you do not want to use the User Datagram Protocol (UDP) protocol that the NameServer uses. If you have restrictions that prohibit using UDP, you can connect directly to an AppServer to avoid these restrictions.

Both Java and ActiveX clients accept the AppServer connection parameter as a single string containing a universal resource locator (URL).

In general, all connection parameters are optional, depending on the AppServer configuration and the application requirements. For any parameter that you do not specify, Progress uses standard default values. However, you must ensure that any defaults are valid for your application environment. For more information, see *Building Distributed Applications Using the Progress AppServer*.

When you connect to the AppServer, you have the option of connecting through a NameServer or of connecting directly to the AppServer. The following shows the AppServer URL syntax for connecting to an AppServer through a NameServer over an intranet:

SYNTAX

```
[[AppServer:][//name-server-host][:name-server-port][service-name]]
```

AppServer

Indicates that the Open Client should connect using a NameServer.

name-server-host

TCP/IP host address of the NameServer that holds the service registration.

name-server-port

TCP/IP port address for the NameServer specified with *name-server-port.*

service-name

Name of the Progress service that supports the application.

The following shows the AppServer URL syntax for connecting directly to an AppServer over an intranet:

SYNTAX

```
AppServerDC:[//AppServer-host][:AppServer-port][service-name]
```

AppServerDC

Indicates the Open Client should connect directly to the AppServer.

AppServer-host

AppServer TCP/IP host address that holds the service.

AppServer-port

AppServer TCP/IP port address for the AppServer specified with *AppServer-host.*

service-name

Name of the Progress service that supports the application.

4.2.2 Connecting Over the Internet

When you distribute your application over the Internet, you can connect to an AppServer using either the HTTP or HTTPS protocol. Using HTTP/S allows clients to communicate through firewalls and optionally send secure encrypted data transmissions.

Connecting Using the HTTP Protocol

The HTTP protocol specifies that Progress use HTTP to connect the client to the AppServer indirectly through a Web server running the AppServer Internet Adapter (AIA).

The following shows the AppServer URL syntax for the HTTP protocol:

SYNTAX

```
http://[user-id[:password]@]web-server-host[:web-server-port]
/aia-path?AppService=service-name
```

Connecting Using the HTTPS Protocol

The HTTPS protocol specifies that Progress is to use HTTPS to connect the client to the AppServer through a Web server running AIA. HTTPS works just like HTTP except that it encrypts all communications between the client and AppServer using SSL.

The following shows the AppServer URL syntax for the HTTPS protocol:

SYNTAX

```
https://[user-id[:password]@]web-server-host[:web-server-port]
/aia-path?AppService=service-name
```

user-id

User account name to use to connect to a Web server that hosts AIA.

password

Password to use with the *user-id*.

web-server-host

TCP/IP host address of the Web server that hosts AIA.

web-server-port

TCP/IP port address of the Web server that hosts the AIA. The default port is 80 for HTTP and 443 for HTTPS.

aia-path

URL path on the Web server where AIA is installed. The *aia-path* identifies the virtual path and servlet name used to invoke the JSE and AIA and must include the following information:

- A virtual path that the JSE has configured within the Web server to recognize and pass directly to the JSE instead of looking for that directory on the Web server
- Name of the Java servlet to be invoked by the JSE

The specification of the path depends on your Web server, JSE, and AIA configuration. For more information, see the *Progress Version 9 Product Update Bulletin* and the documentation for your Web server and servlet engine.

service-name

Name of the Progress service that supports the application. The *service-name* identifies the application service to which the AIA connects. If you do not specify an application service, the AIA uses the default application service registered with ProxyGen.

The following shows some examples of the AppServer URL format where `external/aia1` is the path to the AIA:

```
http://aia.acme.com/external/aia1?AppService=Partners
http://aia.acme.com:2045/external/aia1?AppService=Partners
http://jones@aia.acme.com/external/aia1?AppService=Partners
http://jones:brownie@aia.acme.com/external/aia1?AppService=Partners
```

During the connection process, if Progress determines that the client application is attempting to connect to a secure Web server and that the `httpsEnabled` property is enabled, the protocol is automatically converted to HTTPS.

NOTE: Converting the connection to HTTPS requires that the client be redirected to the port on which the Web server is listening for HTTPS connections. This port is specified using the secure-port property for an AIA instance. For more information, see the *Progress Version 9 Product Update Bulletin*.

4.3 Obtaining Run-time Information

If your Open Client application supports HTTPS, the Open Client Runtime must validate the Web server's digital certificate. Validation is done using a Root Digital Certificate located on the client machine. Because the location of this certificate depends on where the Root Digital Certificate was installed, the location of this certificate must be obtained from the end user at run time. Therefore, the application must obtain this information from the end user and then pass the information to the Open Client Runtime using the RunTimeProperties object.

In addition, if an application must go through a proxy server to connect to AIA, the application must prompt the end user for the proxy host and port and for any required user Id and password information. Again, because the proxy server was configured on the end-user machine, the user using the application must provide the proxy server information. Therefore, the application must obtain the information from the end user and then pass the information to the Open Client Runtime using the methods and properties of the RuntimeProperties object.

Your Open Client application might need to obtain information from the end user when:

- The host address, port, userid, and password are required for a Proxy Web server.
- You need to provide the location of Root Digital Certificates to validate the Web server's identity. Every Web server that hosts an AIA servlet and that supports the HTTPS protocol must have the Root Digital Certificate of the Certificate Authority who issued the digital certificate for the Web server.

Your Open Client application might need to obtain URL connection information from the end user when:

- The URL requires a userid and password for authentication to the Web server hosting the AIA when you use the HTTP or HTTPS protocols.
- The connection URL format differs between the AppServer and AppServerDC protocol and the HTTP and HTTPS protocols.
- You communicate with more than one AIA, you might need to use:
  - Different URL file paths
  - More than one AppService= *name*
- When you use the HTTPS protocol, you might need to supply the names and locations of the Root Digital Certificate files to load and use for authenticating the Web server hosting the AIA Java servlet.

The application might automatically prompt the user for the information when it starts, or it might chose to dynamically handle the Connection Error status returns specific to failed Web server authentication and retry the connection after getting the information from the end user.

For more information, see"Methods For Managing Digital Certificates" in Chapter 5, "Programming Java Clients" and "Methods For Managing Digital Certificates" in Chapter 7, "Programming ActiveX Controller Clients."

4.4 Understanding Proxy Object Methods

As described in previous chapters, proxy objects provide a number of methods that allow you to create other proxy objects and access AppServer procedures. In all, proxy objects support four different types of methods that you can call from an Open Client application:

- **Connection** — Methods that establish a connection to an AppServer
- **Remote 4GL** — Methods that invoke non-persistent procedures, internal procedures, or user-defined functions on the AppServer
- **Class factory** — Methods that create SubAppObjects and ProcObjects
- **Common** — Methods for managing a proxy object and its AppServer connection

In addition to these methods, the proxy provides run-time properties that you can access for specialized purposes. For more information, see the "Accessing Proxy Run-time Properties" section.

The following sections provide information about:

- Connection Methods
- Remote 4GL Methods
- Class Factory Methods
- Common Methods

4.4.1 Connection Methods

You establish a connection to an AppServer through the AppObject constructor for a Java proxy and through a separate method, `OC_Connect(`*`parameter-list`*`)`, for an ActiveX proxy. For more information about connection to an AppServer, see the "Connecting To a Progress AppServer" section. For more information about establishing a connection for a specific type of Open Client, see Chapter 5, "Programming Java Clients," and Chapter 7, "Programming ActiveX Controller Clients."

4.4.2 Remote 4GL Methods

ProxyGen maps each 4GL procedure (internal or external) and user-defined function that you expose on the AppServer to a Remote 4GL method. When invoked, each Remote 4GL method executes the procedure (or user-defined function) on the AppServer, passes any parameter values, and makes any return values available to the Open Client.

NOTE: If the 4GL procedure has a super procedure, the proxy also contains Remote 4GL methods for the super procedure as long as the remote procedure declares the super procedure prototypes. For more information, see the information on programming for Open Client applications in *Building Distributed Applications Using the Progress AppServer*.

The following sections provide information about:

- Identifying the Generated Methods
- Passing Parameters
- Handling Return Values

Identifying the Generated Methods

ProxyGen generates method names using automatic conversions and conventions. For more information, see Chapter 3, "Generating and Managing Proxies." In addition, you can customize the method names in ProxyGen. Thus, each method name might not exactly match the name of the corresponding 4GL procedure or function. Use the generated Java source files for Java clients and the Type Library for ActiveX clients as a reference for method names in your proxy.

Passing Parameters

ProxyGen maps 4GL data types to equivalent data types in the Open Client for 4GL INPUT, OUTPUT, and INPUT-OUTPUT parameters.

For more information on data type mapping for your type of Open Client, see for:

- **Java Open Clients** — Chapter 5, "Programming Java Clients"
- **ActiveX Open Clients** — Chapter 7, "Programming ActiveX Controller Clients"

Handling Return Values

For any remote 4GL procedure, the corresponding proxy method always returns `void`. After calling this method, the client can access the current value of the 4GL RETURN-VALUE function by calling the following common methods for each type of Open Client:

- **Java** — `_getProcReturnString` that returns a `String`, which can be null, so the client can check for a 4GL unknown value(?).
- **ActiveX** — `OC_GetProcReturnString` that returns a `String Variant`, which can be null, so the client can check for a 4GL unknown value(?).

These methods return the current value of RETURN-VALUE as set by the RETURN statement most recently executed on this connection. AppObjects, SubAppObjects, and ProcObjects all support this method.

NOTE: The RETURN-VALUE function is set in the 4GL using the RETURN *`string`* or the RETURN ERROR *`string`* statement. For more information on these statements, see the *Progress Language Reference*, and for how they function in an AppServer application, see *Building Distributed Applications Using the Progress AppServer*.

For any remote user-defined function, ProxyGen defines the corresponding method to return a value that is compatible with the 4GL data type returned by the user-defined function.

4.4.3 Class Factory Methods

As described in the previous chapters, ProxyGen generates class factory methods that allow you to create:

- SubAppObjects that share an AppServer connection with an existing AppObject. Executing a SubAppObject class factory method does not execute anything on the AppServer.
- ProcObjects that share an AppServer connection with an existing AppObject or SubAppObject. Executing a ProcObject class factory method executes the corresponding persistent procedure on the AppServer.
- SDOResultSet is a built-in method that creates a predefined ProcObject, SDOResultSet. SDOResultSet provides Java client access to a specified remote Progress SmartDataObject as an extended JDBC 2 ResultSet. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."

ProxyGen generates class factory methods using standard naming conventions for each type of Open Client. For more information, see Chapter 3, "Generating and Managing Proxies."

4.4.4 Common Methods

AppObjects, SubAppObjects, and ProcObjects all support the common methods described in Table 4–1.

Table 4–1: Proxy Common Methods *(1 of 2)*

Common Method[1]	Description
Java: • `public void _cancelAllRequests() throws Open4GLException` **ActiveX:** • `Sub OC_CancelAllRequests()`	Stops or cancels any outstanding requests to the AppServer by this object or one that shares its connection. This method is most useful for multi-threaded clients.
Java: • `public String _getConnectionId() throws Open4GLException` **ActiveX:** • `Function OC_GetConnectionId() As String`	Returns a string containing a unique identifier for this connection that is used to identify entries in AppServer log files.
Java: • `public String _getProcReturnString()` **ActiveX:** • `Function OC_GetProcReturnString() As String`	Returns the current value of the AppServer RETURN-VALUE function. For more information, see the discussion on handling return values in the "Remote 4GL Methods" section.
Java: • `public boolean _isStreaming() throws Open4GLException` **ActiveX:** • `Function OC_IsStreaming() As Boolean`	Returns true if there is an open output TEMP-TABLE for this connection.[2]

Table 4–1: Proxy Common Methods *(2 of 2)*

Common Method[1]	Description
Java: • `public void _release()` **ActiveX:** • `Sub OC_Release()`	Make this object unavailable for further use. If you execute this method on the last available proxy object that has shared a particular connection, the client disconnects the AppServer. If you execute this method on a ProcObject, the associated persistent procedure on the AppServer is deleted.

[1] ActiveX method prototypes follow the conventions used by the VB Object Browser to display method prototypes.

[2] For more information on streaming, see the "Passing TEMP-TABLE And TABLE-HANDLE Parameters" section.

4.5 Using Proxy Objects

The following sections describe each proxy object in detail, with information on how to use it for each Open Client type.

4.5.1 AppObjects

Each AppObject represents the encapsulation of some set of business logic deployed at a particular AppServer and it establishes a connection to that AppServer.

Methods

An AppObject has the following types of methods:

- **Connection:**
  - **Java** — Constructors that connect the Open Client to an AppServer.
  - **ActiveX** — An `OC_Connect()` method on the AppObject that connects the Open Client to an AppServer.
- **Remote 4GL** — Invoke non-persistent procedures, internal procedures, or user-defined functions on the AppServer.
- **SubAppObject Class Factory** — Create SubAppObjects that share the AppServer connection from this AppObject.

- **Procedure Class Factory** — Create ProcObjects that share the AppServer connection from this AppObject and run an external procedure on the AppServer persistently.
- **SDOResultSet Class Factory**— (Java only) A built-in method that creates a predefined ProcObject, SDOResultSet. SDOResultSet provides Java client access to a specified remote Progress SmartDataObject as an extended JDBC 2 ResultSet. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."
- **Common** — Provide management functions.

Example Java AppObject

Example 4–1 shows a partial sample Java class definition for an AppObject, `Account`. This proxy contains Remote 4GL methods to run `Add.p` and `Remove.p`, and a Class Factory method that runs `AccountInfo.p` persistently on the AppServer.

Java AppObject

```
public class Account
{
  // -------------------------Constructors--------------------------------
  // Connects to an Application Service using a NameServer and
  // Application Broker or directly to an AppServer.

  public Account(String url, String userId, String password, String
         appServerInfo) throws ConnectException, SystemErrorException {}
  public Account(String userId, String password, String
         appServerInfo) throws ConnectException, SystemErrorException {}
  public Account() throws ConnectException, SystemErrorException {}

  // ------------------------Remote 4GL Methods---------------------------

  public void Add(int accountNum, String name) {} // runs Add.p
  public void Remove(int accountNum) {}          // runs Remove.p

  // ------------------SubAppObject Class Factory Methods-----------------

  public Tax createAO_Tax() throws Open4GLException, NotRunException,
         RunTime4GLException, SystemErrorException {}

  // -------------------ProcObject Class Factory Methods------------------

  public AccountInfo createPO_AccountInfo (int accountNum)
         throws Open4GLException, NotRunException, RunTime4GLException,
         SystemErrorException  {}  // runs AccountInfo.p persistently

  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 {}

  // -------------------------Common Methods-----------------------------

  public void _cancelAllRequests() throws Open4GLException {}
  public String _getConnectionId() throws Open4GLException {}
  public String _getProcReturnString() {}
  public boolean _isStreaming() throws Open4GLException {}
  public void _release() {}
}
```

Example 4–1: Java Account Class

For more information on how to specify the parameters for the AppObject constructors for a particular AppServer connection, see the"Connecting To a Progress AppServer" section in this chapter and in Chapter 5, "Programming Java Clients." For more information on the other types of methods, see the "Understanding Proxy Object Methods" section in this chapter.

Example ActiveX AppObject

The following Visual Basic subroutines and functions comprise the definition for a sample ActiveX AppObject, `Account`, where each method appears as it is displayed by the VB Object Browser as `AccountLib.CAccount`:

- The Connection method that connects to an Application Service:

```
Sub OC_Connect ([url As String], [userId As String],
   [password As String], [appServerInfo As String])
```

- A SubAppObject Class Factory method that creates a SubAppObject called tax:

```
Function CreateAO_Tax() As CAO_Tax
```

- A ProcObject Class Factory method that runs and instantiates the persistent procedure `AccountInfo.p` on the AppServer:

```
Function CreatePO_AccountInfo() As CPO_AccountInfo
```

- Remote 4GL methods to run the internal procedures `Add` and `Remove` (defined by `AccountInfo.p`) on the AppServer:

```
Sub Add (accountNum As Long, name As String)
Sub Remove (accountNum As Long)
```

- Common methods in all ActiveX proxy objects:

```
Sub OC_CancelAllRequests()
Function OC_GetConnectionId() As String
Function OC_GetProcReturnString() As String
Function OC_IsStreaming() As Boolean
Sub OC_Release()
```

For more information on how to specify the parameters for the connection method, `OC_Connect()`, for a particular AppServer connection, see the"Connecting To a Progress AppServer" section in this chapter and in Chapter 7, "Programming ActiveX Controller Clients." For more information on the other types of methods, see the "Understanding Proxy Object Methods" section in this chapter.

4.5.2 SubAppObjects

Each SubAppObject represents the encapsulation of a particular business function deployed at a particular AppServer. It differs from an AppObject only in the fact that it does not establish its own connection to an AppServer but shares the one from its associated AppObject.

The associated AppObject defines a class factory method for creating each SubAppObject that shares its AppServer connection. This method exchanges no communications with the AppServer. Rather, it only performs operations to instantiate the SubAppObject on the client.

Methods

A SubAppObject has the following types of methods:

- **Remote 4GL** — Invoke non-persistent procedures, internal procedures, or user-defined functions on the AppServer.
- **Procedure Class Factory** — Create ProcObjects that share the AppServer connection from the associated AppObject and run an external procedure on the AppServer persistently.
- **SDOResultSet Class Factory**— (Java only) A built-in method that creates a predefined ProcObject, SDOResultSet. SDOResultSet provides Java client access to a specified remote Progress SmartDataObject as an extended JDBC 2 ResultSet. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."
- **Common** — Provide management functions.

Example Java SubAppObject

Example 4–2 shows a partial sample Java class definition for a SubAppObject, `Tax`. This proxy contains a Remote 4GL method to run `SetStatus.p` and a Class Factory method that runs `TaxInfo.p` on the AppServer persistently.

Java SubAppObject

```
public class Tax
{
  // ------------------------Remote 4GL Methods--------------------------

  public void SetStatus(int status) {}   // runs SetStatus.p

  // -------------------ProcObject Class Factory Methods-----------------

  public TaxInfo createPO_TaxInfo (int accountNum)
         throws Open4GLException, NotRunException, RunTime4GLException,
         SystemErrorException  {}  // Runs TaxInfo.p persistently

  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 {}

  // --------------------------Common Methods-----------------------------

  public void _cancelAllRequests() throws Open4GLException {}
  public String _getConnectionId() throws Open4GLException {}
  public String _getProcReturnString() {}
  public boolean _isStreaming() throws Open4GLException {}
  public void _release() {}
}
```

Example 4–2: Java Tax Class

For more information on the three types of methods in a SubAppObject, see the "Understanding Proxy Object Methods" section in this chapter.

Example ActiveX SubAppObject

The following Visual Basic subroutines and functions comprise the definition for a sample ActiveX SubAppObject, `Tax`, where each method appears as it is displayed by the VB Object Browser:

- A ProcObject Class Factory method that runs and instantiates the persistent procedure `TaxInfo.p` on the AppServer:

```
Function CreatePO_TaxInfo() As CPO_TaxInfo
```

- Remote 4GL method to run the internal procedure `SetStatus` (defined by `TaxInfo.p`) on the AppServer:

```
Sub SetStatus(status As Long)
```

- Common methods in all ActiveX proxy objects:

```
Sub OC_CancelAllRequests()
Function OC_GetConnectionId() As String
Function OC_GetProcReturnString() As String
Function OC_IsStreaming() As Boolean
Sub OC_Release()
```

For more information on the three types of methods in a SubAppObject, see the "Understanding Proxy Object Methods" section in this chapter.

4.5.3 ProcObjects

Each ProcObject represents a particular persistent procedure running on an AppServer. The ProcObject exposes all non-PRIVATE internal procedures and user-defined functions to the client that you do not explicitly omit in ProxyGen when you define the ProcObject.

A ProcObject also shares a connection established by an associated AppObject. You can create the ProcObject using a class factory method on the AppObject or SubAppObject to which this procedure was added in ProxyGen

When you call the ProcObject class factory method on the AppObject or SubAppObject, it creates the ProcObject on the client and executes the corresponding procedure persistently on the AppServer. Any parameters required by this procedure are required as parameters on the class factory method. The ProcObject saves a handle to the persistent procedure for future calls to its internal procedures and functions and for releasing and disconnecting the object from the AppServer application.

Methods

A ProcObject has the following types of methods:

- **Remote 4GL** — Run internal procedures and user-defined functions on the AppServer that are defined in the associated remote persistent procedure
- **Common** — Provide management functions

Example Java ProcObject

Example 4–3 shows a partial sample Java class definition for a ProcObject, `AccountInfo`. This proxy contains Remote 4GL methods to run `getPaymentsInfo()`, `setDirectDeposit()`, and `getDirectDeposit()` on the AppServer.

Java ProcObject

```
public class AccountInfo
{
  // ------------------------Remote 4GL Methods---------------------------

  public void getPaymentsInfo(Date fromDate, ResultSet payeeList,
         int paymentsNum, ResultSet paymentsInfo) {}
                            // runs internal proc getPaymentsInfo()
  public void setDirectDeposit(ResultSet ddData) {}
                            // runs internal proc setDirectDeposit()
  public void getDirectDeposit(ResultSetHolder ddData) {}
                            // runs internal proc getDirectDeposit()

  // --------------------------Common Methods-----------------------------

  public void _cancelAllRequests() throws Open4GLException {}
  public String _getConnectionId() throws Open4GLException {}
  public String _getProcReturnString() {}
  public boolean _isStreaming() throws Open4GLException {}
  public void _release() {}
}
```

Example 4–3: Java AccountInfo Class

For more information on the two types of methods in a ProcObject, see the"Understanding Proxy Object Methods" section in this chapter.

Example ActiveX ProcObject

The following Visual Basic subroutines and functions comprise the definition for a sample ActiveX ProcObject, `AccountInfo`, where each method appears as it is displayed by the VB Object Browser:

- Remote 4GL methods to run the internal procedures `getPaymentsInfo`, `setDirectDeposit`, and `getDirectDeposit` (defined by `AccountInfo.p`) on the AppServer:

```
Sub getpaymentsInfo(fromDate As Double, payeeList As Unknown,
   paymentsNum As Long, paymentsInfo As Unknown)
Sub setDirectDeposit(ddData As Unknown)
Sub getDirectDeposit(ddData As Unknown)
```

NOTE: *Unknown* corresponds to a ProTempTable object in this example.

- Common methods in all ActiveX proxy objects:

```
Sub OC_CancelAllRequests()
Function OC_GetConnectionId() As String
Function OC_GetProcReturnString() As String
Function OC_IsStreaming() As Boolean
Sub OC_Release()
```

For more information on the two types of methods in a ProcObject, see the"Understanding Proxy Object Methods" section in this chapter.

For information on TEMP-TABLE parameters, first see the"Passing TEMP-TABLE And TABLE-HANDLE Parameters" in this chapter and then in Chapter 7, "Programming ActiveX Controller Clients."

4.6 Passing Parameters In Proxy Object Methods

The Open Client passes parameters when the corresponding 4GL procedure or user-defined function takes parameters. The type of mechanism used to send and receive parameters depends on the type of Open Client. Although the types of Open Client affects parameter passing, a common model underlies the mechanisms that both types of Open Clients use.

The following sections describe this model and provide information about:

- Passing TEMP-TABLE And TABLE-HANDLE Parameters
- Result Set Schemas
- Comparing the Flat Model and Array Model For Result Set Access

For information about how a specific Open Client type passes parameters, see:

- **Java Open Clients** —Chapter 5, "Programming Java Clients"
- **ActiveX Open Clients** — Chapter 7, "Programming ActiveX Controller Clients"

4.6.1 Passing TEMP-TABLE And TABLE-HANDLE Parameters

4GL procedures can pass TEMP-TABLE or TABLE-HANDLE parameters to an AppServer. This is a convenient and efficient way to pass relational data between the client and the AppServer code. The Open Client environment provides the same capability using result sets. A result set is an object that provides access to relational data in an Open Client application. An Open Client passes a 4GL TEMP-TABLE parameter using a result set parameter that is mapped to the specified TEMP-TABLE.

TABLE–HANDLE parameters allow the transfer of TEMP–TABLE objects between Open Clients and the AppServer without requiring a complete, static definition of the table on each side of the transfer (as with TEMP-TABLE parameters). The TEMP-TABLE and TABLE-HANDLE parameters also use a result set parameter.

INPUT TEMP-TABLE and TABLE-HANDLE Parameters

When the client application passes a static TEMP–TABLE as an INPUT parameter, the client provides an object that contains data for the table. For a Java client, this object is a `java.sql.ResultSet` and for an ActiveX client, this object is an ADO, DAO or `IClientTempTable`. In this instance, the proxy already knows the schema (metadata) 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 (metadata) for the table. The client must provide both the schema and the data t for the TABLE–HANDLE parameters. If the application does not provide metadata, the method call fails.

For more information about how metadata is provided to the a specific client type, see:

- **Java Open Clients** —Chapter 5, "Programming Java Clients"
- **ActiveX Open Clients** — Chapter 7, "Programming ActiveX Controller Clients"

OUTPUT TEMP-TABLE and TABLE-HANDLE Parameters

An application uses the same model when working with an OUTPUT TABLE–HANDLE parameter or a TEMP–TABLE parameter, by having the client pass a holder object. For Java this is a `ResultSetHolder`, and for ActiveX, this is an instance of `ProTempTable`.

On return from the method, the holder has access to the output result set data and the metadata. In Java, the data is returned through a `ResultSet` object from which a metadata object can be obtained. In ActiveX, the data and metadata is returned through `ProTempTable`.

Although the model is the same, TABLE–HANDLEs are different in that they can be Unknown or undefined. As a result, you must ensure that the client can handle these cases.

INPUT-OUTPUT TEMP-TABLE and TABLE-HANDLE Parameters

INPUT–OUTPUT TEMP-TABLE and TABLE-HANDLE parameters are a combination of INPUT and OUTPUT parameters. For Java, the input `ResultSet` is stored in a `ResultSetHolder` and the holder is passed as the parameter rather than passing the `ResultSet` directly. For ActiveX, the input side is the same for both INPUT and INPUT–OUTPUT because the client must set the `DataSource` for `ProTempTable` in both instances.

TEMP-TABLE and TABLE-HANDLE Call-back Sequence

Open Client support for TEMP-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 must always requests data from the opposite side, rather than having the opposite side set or send the data. As a result, the client application calls back to the proxy for output parameters and the proxy calls back to the client application for input parameters.

Table 4–2 shows how the call-back sequence works when passing an INPUT TEMP-TABLE or TABLE-HANDLE parameter.

Table 4–2: Passing an INPUT TEMP-TABLE Or TABLE-HANDLE Parameter

Client Application	Proxy
1) Creates input result set object **2)** Passes result set object as a parameter to proxy	–
–	**3)** Uses methods on the input result set object to get data **4)** Passes input data to AppServer

Table 4–3 shows how the call-back sequence works when passing an OUTPUT TEMP-TABLE or TABLE-HANDLE parameter.

Table 4–3: Passing an OUTPUT TEMP-TABLE Or TABLE-HANDLE Parameter

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

Table 4–4 shows how the call-back sequence works when passing an INPUT-OUTPUT TEMP-TABLE Or TABLE-HANDLE parameter.

Table 4–4: Passing an INPUT-OUTPUT TEMP-TABLE Or TABLE-HANDLE Parameter

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

The input result set object is an object that is written by the Java or ActiveX client application. This can also be a standard result set object available from each respective environment as the result of an SQL query (a `java.sql.ResultSet` in Java, or a DAO or ADO `Recordset` object in Visual Basic). In contrast, the output result set object is always provided by the generated proxy.

Input Result Sets

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 result set object (that 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 Result Sets

On output (or the output side of input/output parameters), the result set is not necessarily finished being transmitted to the client by the time the method call returns. The proxy buffers data for the result set and makes it available sequentially to the client.

This process of buffering an output result set from the AppServer and making it sequentially available to the client is called *streaming*. An AppServer connection that is streaming a result set is said to be in a STREAMING state. This STREAMING state remains in effect until all data has been received by the client or the result set is closed.

When the method containing an output result set parameter returns, the client gets an object that 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, you can invoke the `_isStreaming()` method (Java client) or the `OC_IsStreaming()` method (ActiveX client) for any proxy object using the connection.

While both types of Open Clients handle output result sets using streaming, you must make special provisions for streaming in a multi-threaded Java client. For more information on managing streaming in Java clients, see the information on multi-threaded applications in Chapter 5, "Programming Java Clients."

Order of Access For Output Result Sets

This stream of data 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 to go backwards. When getting the data from a single TEMP-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 TEMP-TABLE or TABLE-HANDLE parameter for a method, they must be perused in sequence. Data from the first must be completely fetched or the result set closed before any data can be retrieved from the second TEMP-TABLE or TABLE-HANDLE parameter, and so forth. 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 Result Sets

To close an output result set, you generally execute the close method on the result set as appropriate for your Java or ActiveX Open Client. However, you can also close the result set by executing the release method on all proxy objects running over the same AppServer connection that returned the result set. Releasing the last such AppObject, SubAppObject, or ProcObject, disconnects the Open Client from the AppServer and closes any output result sets that it provides.

For more information on the close and release methods for your type of Open Client, see for:

- **Java Open Clients** — Chapter 5, "Programming Java Clients"
- **ActiveX Open Clients** — Chapter 7, "Programming ActiveX Controller Clients"

4.6.2 Result Set Schemas

Every result set 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 is available to the Open Client application. For the Java client, it is available through the `getMetaData()` method of the `ResultSet` object. For ActiveX clients, it is exposed through the methods of the `ProField` 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.

For more information on accessing result set schemas, see the information on passing TEMP-TABLE parameters for your type of Open Client:

- **Java Open Clients** — Chapter 5, "Programming Java Clients"
- **ActiveX Open Clients** — Chapter 7, "Programming ActiveX Controller Clients"

4.6.3 Comparing the Flat Model and Array Model For Result Set Access

An array field in a Progress TEMP-TABLE can be viewed as a result set 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 Progress handles array fields.

For both Java and ActiveX clients, there are access methods to support both models. These methods access field values and metadata of an output result set.

The following examples use the Java flat model metadata methods `getColumnType()` and `getColumnCount()` and the array model methods `getFieldProType()` and `getFieldCount()`. For more information on the result set access methods for both Java and ActiveX clients, see Chapter 5, "Programming Java Clients," Chapter 6, "Using SmartDataObjects From Java Clients," and Chapter 7, "Programming ActiveX Controller Clients."

These examples illustrate the use of a Progress 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 like this:

- `getColumnCount()` returns the value 14.
- `getColumnType(4)` gets the data type of the third element of the second TEMP-TABLE field (which is the array field).
- `getColumnType(14)` gets the data type of the third TEMP-TABLE field.

Since all elements of an array field have the same metadata information, `getColumnType`(2) through `getColumnType(13)` return the same value.

Using the Array Model

The array model methods behave like this:

- `getFieldCount()` returns the value 3.
- `getFieldProType(2)` gets the data type of the second TEMP-TABLE field.
- `getFieldProType(3)` gets the data type of the third TEMP-TABLE field.

In this model, since all elements of an array field have the same metadata information, it is only necessary to pass in the field index (without an array item index) to these metadata 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:

```
getObject(int fieldNum, int arrayIndex)
```

4.7 Accessing Proxy Run-time Properties

There are a number of properties that are provided to govern behavior across the entire application. These properties do not affect any particular object, but affect the behavior of all objects created by the application. These are referred to as run-time properties and include: tracing, thread control, Proxy server support, and digital certificate management.

For more information on how to access these properties in your type of Open Client, see:

- Chapter 5, "Programming Java Clients" for Java Open Clients
- Chapter 7, "Programming ActiveX Controller Clients" for ActiveX Open Clients

4.8 Handling Errors In an Open Client Application

A remote 4GL request can generate three different types of errors:

- Client-side Errors
- Server-side Errors Before Remote 4GL Execution
- Server-side Errors During Remote 4GL Execution

A *client-side error* is an error that occurs on the client machine before the proxy sends a particular request to the AppServer. A *server-side error* is an error that occurs on the AppServer machine, either before the AppServer runs a remote procedure or during procedure execution.

No matter what type of error Progress returns, the proxy makes the error message available to the client application. For Java applications, the proxy also makes the Progress error message number available.

For more information on how proxies return errors for your type of Open Client, see for:

- **Java Open Clients** — Chapter 5, "Programming Java Clients"
- **ActiveX Open Clients** — Chapter 7, "Programming ActiveX Controller Clients"

A different set of rules determines how Progress propagates each type of error to the client application. The following sections describe these rules and the types of errors that are returned.

4.8.1 Client-side Errors

The proxy returns all client-side errors to the client. An example of a client-side error is an input result set whose schema does not match that of the corresponding 4GL TEMP-TABLE. (For more information on result sets, see the"Passing TEMP-TABLE And TABLE-HANDLE Parameters" section.) The proxy returns any such error in a manner appropriate for your type of client.

Because the error occurs before the request is sent to the AppServer, the request is cancelled with no effect on the AppServer. In general, client-side errors have no effect on the AppServer to which the client is connected. However, the AppServer does attempt to detect the abnormal termination of a client connection and executed the configured disconnect sequence when it happens.

4.8.2 Server-side Errors Before Remote 4GL Execution

Progress returns all server-side errors to the client that occur before any remote procedure is executed. An example of this type of server-side error is where the client tries to execute a remote procedure that does not exist. Generally, both the client and AppServer can resume normal execution.

Progress also returns all network connection and authentication errors. For example, if an HTTP connection cannot be made, authentication cannot be established with the Web server, or the AppServer Internet Adapter (AIA) is not installed and configured properly.

4.8.3 Server-side Errors During Remote 4GL Execution

Progress selectively returns server-side errors that occur during execution of a remote procedure. There are three types of remote procedure conditions that an AppServer can return to the client:

- **ERROR condition** — A server-side ERROR condition is sent to the client only when a remote procedure or function (or an Activate procedure) executes a RETURN ERROR statement. All other server-side ERROR conditions (such as failure to connect to a database) are not raised beyond the scope of the 4GL procedure where they occur. Therefore, such ERROR conditions do not propagate to the client. Instead, the AppServer writes error messages for these conditions to the server log file.

- **STOP condition** — Any unhandled STOP condition raised on the AppServer is sent to the client. This type of server-side error can occur, for example, when the called remote procedure calls another procedure that cannot be found.

- **QUIT condition** —Any unhandled QUIT condition raised on the AppServer is sent to the client. The server-side 4GL application can use a QUIT statement to raise a QUIT condition because the client exceeded some resource, did something it should not do, or for any other reason it chooses.

5

Programming Java Clients

For the Java client, ProxyGen generates a proxy as a set of Java classes to access Progress AppServer functionality. ProxyGen generates the proxy classes into a package, if you specify one, for proxy generation. You can also access remote Progress SmartDataObjects from a Java client with or without a ProxyGen-generated proxy. For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."

This chapter describes how to access AppServer functionality (other than Progress SmartDataObjects) using ProxyGen-generated Java proxies and the Progress Open Client Runtime. It describes:

- Connecting To an AppServer
- Data Type Mapping
- Passing TEMP-TABLE Or TABLE-HANDLE Parameters
- Extending Proxy Objects
- Using Proxies In Multi-threaded Applications
- Accessing Proxy Run-time Properties
- Building Java Applets As Open Clients
- Handling Errors

NOTE: You can find HTML documentation on the classes and interfaces provided by the Open Client Runtime in *Progress-Install-Directory*/java/doc.

5.1 Connecting To an AppServer

To establish an AppServer connection in a Java client, you must instantiate an AppObject. ProxyGen generates the class constructor for each AppObject in three overloaded forms:

SYNTAX

```
public class-name
 ( String url,
   String userid,
   String password,
   String appserver-info
 );
```

```
public class-name
  ( String userid,
    String password,
    String appserver-info
  );
```

```
public class-name ();
```

The *url* parameter specifies the AppServer connection information. The *userid*, *password*, and *appserver-info* parameters specify any information required by the AppServer application at connection time. For more information on how to specify these parameters, see the information on connecting to an AppServer in Chapter 4, "Programming Concepts."

5.2 Data Type Mapping

INPUT 4GL parameters are mapped directly to Java data types and classes. OUTPUT and INPUT-OUTPUT parameters, with the exception of TEMP-TABLE and TABLE-HANDLE 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.

The Java proxy data type applies to both INPUT and OUTPUT parameters for all except TEMP-TABLE and TABLE-HANDLE parameters, where the data type differs for INPUT and OUTPUT parameters.

Table 5–1 shows the 4GL-to-Java data type mapping for parameters.

Table 5–1: Progress To Java Data Type Mapping For Parameters

Progress Data Type	Java Proxy Data Type
CHARACTER	`String`
COM-HANDLE	`com.progress.open4gl.COMHandle`
DATE	`java.util.GregorianCalendar`
DECIMAL	`Java.math.BigDecimal`
INTEGER	`int`
LOGICAL	`boolean`
MEMPTR	`com.progress.open4gl.Memptr`
RAW	`byte[]`
RECID	`long`
ROWID	`com.progress.open4gl.Rowid`
TABLE-HANDLE	`java.sql.ResultSet` (input) `com.progress.open4gl.ProResultSet` (output)
TEMP-TABLE	`java.sql.ResultSet` (input) `com.progress.open4gl.ProResultSet` (output)
WIDGET-HANDLE	`com.progress.open4gl.Handle`

The COM-HANDLE, WIDGET-HANDLE, RECID, or ROWID 4GL data types are not meaningful outside of the Progress environment. Their use is restricted to obtaining the value from a Progress procedure and being able to pass it back to another Progress procedure.

The mapping for the individual fields of a TEMP-TABLE differs from that of other parameters. For information on data type mapping for TEMP-TABLE or TABLE-HANDLE parameters, see Table 5–3.

NOTE: Open Clients do not support the Progress BUFFER parameter.

5.2.1 Progress Unknown Values

The Progress unknown value (?) is mapped to a Java object with a null value. However, the `int`, `boolean`, and `long` data types are intrinsic types, not Java classes. So, the null value does not belong to the set of values they support. If a proxy is generated to support Progress unknown values, parameters that would otherwise be intrinsic data types are instead 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 Progress unknown values for the 1st and 3rd parameter:

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

This is an example of a proxy method that allows all the parameter values to have unknown values:

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

5.2.2 INPUT-OUTPUT and OUTPUT Parameters

Java supports no direct mechanism to return a value from a method using parameters. However, any Java application can provide object parameters (*holder objects*) to contain the value that you want to return.

Holder Classes

For each of the Java data types that map to a Progress 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 shown in Example 5–1.

```
// Holder Definition
Public class Holder
{
  public Holder();
  public void setValue(Object value);
  public Object getValue();
  public boolean isNull();
}
```

Example 5–1: Output Holder Class

For an INPUT-OUTPUT parameter, you can set an input value to the 4GL unknown value (?) by calling `setValue(null)` on the holder object. For OUTPUT parameters, the application can find out whether an unknown output value is returned by calling the `isNull()` method or by testing for a `null` return value from the `getValue()` method on the holder object.

Table 5–2 shows the 4GL-to-Java mapping of input-output and output parameters to specific `Holder` classes.

Table 5–2: Progress/Java Data Type Map For Output Parameters

Progress Data Type	Java Proxy Data Type
CHARACTER	`com.progress.open4gl.StringHolder`
COM-HANDLE	`com.progress.open4gl.COMHandleHolder`
DATE	`com.progress.open4gl.DateHolder`
DECIMAL	`com.progress.open4gl.BigDecimalHolder`
INTEGER	`com.progress.open4gl.IntHolder`
LOGICAL	`com.progress.open4gl.BooleanHolder`
MEMPTR	`com.progress.open4gl.MemptrHolder`
RAW	`com.progress.open4gl.ByteArrayHolder`
RECID	`com.progress.open4gl.LongHolder`
ROWID	`com.progress.open4gl.RowidHolder`
TABLE-HANDLE	`com.progress.open4gl.ResultSetHolder`
TEMP-TABLE	`com.progress.open4gl.ResultSetHolder`
WIDGET-HANDLE	`com.progress.open4gl.HandleHolder`

Holder Class Definitions

The Holder class definitions for each data type follow the form shown in Example 5–2.

```
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 LongHolder extends Holder
{
  public LongHolder();
  public LongHolder(long value);
  public void setLongValue(long value);
  public long getLongValue();
}

/* Holder extensions for other data types follow similar forms */
```

Example 5–2: Holder Class Definitions

5.3 Passing TEMP-TABLE Or TABLE-HANDLE Parameters

Progress supports both TEMP-TABLE and TABLE-HANDLE parameters. Both parameter types map to the same Java class.

Java clients can access 4GL TEMP-TABLE or TABLE-HANDLE parameters using the following Java classes and interfaces provided with the Progress Open Client Runtime:

- **INPUT parameters** — Any class that implements the `java.sql.ResultSet` interface.
- **OUTPUT parameters** — Any class that implements the `com.progress.open4gl.ProResultSet` interface, which extends the `java.sql.ResultSet` interface.

For more information on the `java.sql.ResultSet` interface, see the relevant JavaSoft publications.

Progress provides the abstract class, `com.progress.open4gl.InputResultSet`, provides part of an implementation of `java.sql.ResultSet`. You can extend this class to implement your own input result sets. For more information, see the "Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT Parameter" section.

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 Progress fields, especially array fields, in a more Progress-like manner. For more information on how to access column and field values using the `ProResultSet` interface, see the "Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output" section.

The following sections provide information about:

- Default Data-type Mapping For TEMP-TABLE Fields
- Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT Parameter
- Providing Metadata For INPUT TABLE-HANDLE Parameters
- Passing a TEMP-TABLE Or TABLE-HANDLE As an OUTPUT Parameter
- Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT-OUTPUT Parameter
- Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output
- Getting Metadata For OUTPUT TEMP-TABLE and TABLE-HANDLE Parameters
- The Java Client Application Using AccountInfo

5.3.1 Default 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 in order to conform to the JDBC standard. Table 5–3 shows the mappings.

Table 5–3: Default Data-type Mapping For TEMP-TABLE Fields

Progress Data Type	Java Object Type
CHARACTER	`String`
COM-HANDLE	`Long`
DATE	`java.sql.Date` or `java.util.GregorianCalendar`
DECIMAL	`Java.math.BigDecimal`
INTEGER	`Integer`
LOGICAL	`Boolean`
RAW	`byte[]`
RECID	`Long`
ROWID	`byte[]`
WIDGET-HANDLE	`Long`

There are two different Java data types listed for DATE. For an input TEMP-TABLE or TABLE HANDLE, you can supply either of the object types shown. For details about how input TEMP-TABLE data is supplied to the proxy, see the "Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT Parameter."

For output, your application calls an instance of `java.sql.ResultSet` to get the data of each TEMP-TABLE field or TABLE-HANDLE. There is both a `getDate()` method that returns a `java.sql.Date` and a `getGregorianCalendar()` method that returns a `java.util.GregorianCalendar` on this result set. There is also a `getObject()` method that when called returns a `java.sql.Date`. For more information, see the "Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output" section.

`getObject()` can return either one of these object types. For output, there is both a that returns a `java.sql.Date` and a `getGregorianCalendar()` method that returns a `java.util.GregorianCalendar`. If `getObject()` is called, a `java.sql.Date` is returned.

When the `get`*`DataType`*`()` method that you use to get the value of a TEMP-TABLE field does not match the default Java data type of the field (Table 5–3), a valid conversion occurs as long as the `get`*`DataType`*`()` method corresponds to the Progress data type as shown in Table 5–4. 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 5–4: Data Type Conversions For TEMP-TABLE Fields *(1 of 2)*

Progress Data Type	Valid Java Data Type Conversions
CHARACTER	`java.lang.String getString()`

COM-HANDLE	`java.lang.String getString()`
	`int getInt()`
	`long getLong()`

DATE	`java.lang.String getString()`
	`java.sql.Date getDate()`
	`java.util.GregorianCalendar getGregorianCalendar()`

DECIMAL	`java.lang.String getString()`
	`int getInt()`
	`long getLong()`
	`double getDouble()`
	`java.math.BigDecimal getBigDecimal()`

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()`

Table 5–4: Data Type Conversions For TEMP-TABLE Fields *(2 of 2)*

Progress Data Type	Valid Java Data Type Conversions

RECID	`java.lang.String getString()`
	`int getInt()`
	`long getLong()`

ROWID	`byte[] getBytes()`

WIDGET-HANDLE	`java.lang.String getString()`
	`int getInt()`
	`long getLong()`

5.3.2 Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT Parameter

For an input 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 TEMP-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 Progress.

The two methods you must implement are:

SYNTAX

```
boolean next()
```

Advances the result set cursor to the next row. The cursor is always initially positioned before the first row of the result set, and 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.

NOTE: Make sure the result set 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.

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 4GL procedure.

If there is an array field in the Progress 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 only be one way to get the data and that one way assumes the flat column model native to the standard JDBC `ResultSet`. For more information on the relationship between the flat column model assumed by JDBC and the Progress field model (which includes arrays), see Chapter 4, "Programming Concepts."

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 read from a file. Therefore, the exact implementation depends entirely on the specific requirements of your application.

5.3.3 Providing Metadata For INPUT TABLE-HANDLE Parameters

For TEMP-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 call the `getMetaData()`method 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 under the following conditions**:
  - If the input ResultSet is a standard JDBC ResultSet obtained from a query, 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 Progress data types as shown in Table 5–5.

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

Table 5–5: Mapping Between SQL and Progress Data Types For Standard JDBC ResultSet

Progress 4GL Type	SQL Type
INTEGER	INTEGER
DECIMAL	DECIMAL
CHARACTER	LONGVARCHAR
LOGICAL	BIT
DATE	DATE
RAW	VARBINARY

If the ResultSet contains any other type, an Exception is throw.

– If the ResultSet contains a column type other than the types shown in Table 5–5 and the column can be converted to a compliant type, implement your own ResultSet class instead of passing the java.sql.ResultSet directly as the parameter. For example, you can extend com.progress.open4gl.InputResultSet with your own class.

When getObject() is called on your class, it retrieves data from the corresponding standard object and converts the data before returning it to the proxy. When the getMetaData() method is called, the implementation of your extended class must supply the appropriate metadata. To do this, the application must create and populate a ProResultSetMetaDataImpl object as described in the section under the **Use ProResultSetMetaDataImpl** bullet.

If the application requires the resulting server-side TEMP–TABLE to have an array field, this is another reason for you to implement a delegating InputResultSet class. This time, the implementation of both next() and getObject() call the standard instance's corresponding methods to do the work. However, when the getMetaData() method is called, the delegating instance must supply the required metadata, with the flattened schema view converted to the corresponding array field view. Again, to do this, the application must create and populate a ProResultSetMetaDataImpl object as described in the section under the **Use ProResultSetMetaDataImpl** bullet.

- **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, metadata can be obtained from that output set by calling getMetaData() on it. You can think of this as an INPUT–OUTPUT TABLE-HANDLE parameter even though it is output from one method call and input through another. The ResultSet object obtained from the output ResultSet can then be provided when getMetaData() is called for the input ResultSet.

- **Use ProResultSetMetaDataImpl under the following condition:**

  If the metadata for the input ResultSet must be supplied by code implemented in the client application, the getMetaData() method must be implemented to return an instance of ResultSetMetaData. The recommended way to do this is to use the com.progress.open4gl.ProResultSetMetaDataImpl class provided in the Open Client Toolkit. This is an implementation of the ProResultSetMetaData interface, the same interface that is used by the client when getting output data.

The only two methods of ProResultSetMetaDataImpl that the application must call are the constructor and the SetFieldMetaData() method as follows:

SYNTAX

```
ProResultSetMetaDataImpl (int numFields)
```

SYNTAX

```
SetFieldMetaData (int fieldNumber, String Name, int extent, int type)
```

The SetFieldMetaData() method requires the following:

- The *fieldNumber* parameter must be between 1 and the number of fields
- The *Name* parameter cannot be null
- The value of the *Name* parameter must be unique within the table
- The *extent* must be zero or greater
- The *type* parameter must be of the PRO_XXX types supplied by com.progress.open4gl.Parameter

The Open Client Toolkit JavaDoc in *Progress-Install-Directory*/java/doc describes the methods used by the client application for the ProResultSetMetaDataImpl class.

To pass a self-implemented result set for a TABLE–HANDLE parameter, the application must:

a. 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 also for getMetaData() which should return the com.progress.open4gl.ProResultSetMetaDataImpl instance when called.

b. Create an instance of the class you created in Step a.

c. Create an instance of com.progress.open4gl.ProResultSetMetaDataImpl.

d. Call ProResultSetMetaDataImpl.setFieldMetaData() for each field of the table

e. 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.

f. Make the proxy method call passing the `com.progress.open4gl.InputResultSet` instance as the parameter

NOTE: Step c and Step d can be implemented in `getMetaData()` which eliminates the need for Step e.

5.3.4 Passing a TEMP-TABLE Or TABLE-HANDLE As an OUTPUT Parameter

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 contents is set to a `com.progress.open4gl.ProResultSet` object. When the method returns, the Java client calls back through this object's methods to get the data. For more information on `com.progress.open4gl.ProResultSet`, see the "Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output" section.

If the parameter is a TABLE-HANDLE, the value returned can be unknown or undefined. As a result, calling `getObject()` on the holder returns null.

5.3.5 Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT-OUTPUT Parameter

For an INPUT-OUTPUT parameter, you must pass an instance of `com.progress.open4gl.ResultSetHolder` 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.

For more information on creating a `java.sql.ResultSet` for input, see the "Passing a TEMP-TABLE Or TABLE-HANDLE As an INPUT Parameter" section. For more information on `ProResultSet` objects, see the "Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output" section.

5.3.6 Accessing TEMP-TABLE Or TABLE-HANDLE Parameters As Output

For both output TEMP-TABLE and TABLE-HANDLE parameters and input-output TEMP-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 Progress array model, which presents a TEMP-TABLE as individual fields that can contain arrays. You can choose whichever model is most natural or convenient. For more information on the Progress field model, see Chapter 4, "Programming Concepts."

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 include:

SYNTAX

```
boolean next()
```

Advances the result set cursor to the next row.

SYNTAX

```
void close()
```

Closes the result set. For output result sets, the `close()` method may be called to close the result set before all the rows have been fetched.

NOTE: No other AppServer method can be called until all the rows have been fetched or until `close()` has been called.

SYNTAX

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

Returns the value of the column identified by the 1-based *columnNum*. The data type of the returned object is the default data type of the column. For example, the default data type for a 4GL CHARACTER field is `java.lang.String`. For more information, see the "Default Data-type Mapping For TEMP-TABLE Fields" section.

SYNTAX

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

Returns the value of the column identified by the *fieldName*. The data type of the returned object is the default data type of the field. For example, the default data type for a 4GL CHARACTER field is `java.lang.String`. For more information, see the "Default Data-type Mapping For TEMP-TABLE Fields" section.

SYNTAX

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

Returns the value of the column identified by *columnNum*. *DataType* is the name of the data type, for example, `String getString()`, `Integer getInteger()`, and so on.

SYNTAX

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

Returns the value of the column identified by the *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 4GL 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 Table 5–4.

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 result set based on the Progress array model (see the section on the array and flat models in Chapter 4, "Programming Concepts").

SYNTAX

```
Object getObject(int fieldNum, int arrayIndex)
```

Returns the value of the *arrrayIndex* element of the array field identified by *fieldNum*. The datatype of the object is the default datatype of the `arrayfield`. The *arrayIndex* and *fieldNum* are 1-based.

SYNTAX

```
Object getObject(String fieldName, int arrayIndex)
```

Returns the value of the `arrayIndex` element of the `arrayfield` identified by *fieldName*. The datatype of the object is the default datatype of the `arrayfield`. The *arrayIndex* and *fieldName* are 1-based.

SYNTAX

```
DataType getDataType(int fieldNum, int arrayIndex)
```

Name of datatype. For example, `String getString()`, `Integer getInteger()`, and so on. Returns the value of the *arrayIndex* element of the `arrayfield` identified by *fieldNum*. The *arrayIndex* and *fieldNum* are 1-based.

SYNTAX

```
DataType getDataType (String fieldName, int arrayIndex)
```

Name of the datatype. For example, `String getString()`, `Integer getInteger()`, and so on. The datatype of the object is the default datatype of the `arrayfield`. The *arrayIndex* and *fieldNum* are 1-based.

For example, here 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 *Progress-Install-Directory*/java/doc.

5.3.7 Getting Metadata For OUTPUT TEMP-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 a Progress 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:

```
// 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 and Progress TEMP-TABLE fields, using the array model.

Standard Methods In The ProResultSetMetaData Class

The `ProResultSetMetaData` class has the following public methods:

- These 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 result set:

SYNTAX

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

SYNTAX

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

SYNTAX

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

The `getColumnType()` method returns an `int` constant for the standard SQL types as shown in Table 5–6. The column under SQL Data Type in the table shows the type names as specified by the `java.sql.Types` class.

SYNTAX

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

The `getColumnTypeName()` method returns the corresponding Progress data type names shown in Table 5–6 under Progress Data Type.

Table 5–6: Progress-SQL ResultSet Data Type Mappings *(1 of 2)*

Progress Data Type	SQL Data Type
CHARACTER	LONGVARCHAR
COM-HANDLE	BIGINT
DATE	DATE
DECIMAL	DECIMAL

Table 5–6: Progress-SQL ResultSet Data Type Mappings *(2 of 2)*

Progress 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 of 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) and have prototypes that refer to `Column` or `column`.

- These methods allow you to see the data type of each result set column as Progress and Java types:

SYNTAX

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

Returns one of the following values accessible from `com.progress.open4gl.Parameter`:

```
PRO_CHARACTER = 1;
PRO_DATE = 2;
PRO_LOGICAL = 3;
PRO_INTEGER = 4;
PRO_DECIMAL = 5;
PRO_RECID = 7;
PRO_RAW = 8;
PRO_WIDGETHANDLE = 10;
PRO_ROWID = 13;
PRO_COMHANDLE = 14;
```

SYNTAX

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

`getColumnJavaTypeName()` returns the name of the Java class that each Progress data type maps to. For information on these mappings, see Table 5–3.

- These methods provide access to the schema of TEMP-TABLE parameters that contain array fields, viewed from a Progress point of view:

SYNTAX

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

SYNTAX

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

SYNTAX

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

SYNTAX

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

SYNTAX

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

SYNTAX

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

The Progress-oriented methods, which view the TEMP-TABLE using the Progress 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 Progress array fields, see the section comparing the flat model and array model for result sets in Chapter 4, "Programming Concepts."

Although you can access the data using either the SQL or Progress model, both provide equivalent functionality. Table 5–7 shows the correspondence between the `ProResultSetMetaData` methods.

Table 5–7: Comparing Some Array and Flat Model Methods

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

5.3.8 The Java Client Application Using AccountInfo

The code in Example 5–3 demonstrates how to create and use AppObjects and ProcObjects from a Java client. In this example, the AppObject was defined with ProxyGen as Account.

Java Client Using AccountInfo *(1 of 2)*

```
// 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 5-3 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();
```

Java Client Using AccountInfo *(2 of 2)*

```
// Prints all the payment records.
while (paymentsInfo.next())
{
// Gets the next record.

   int colNum = paymentsInfo.getMetaData().getColumnCount();

    // Prints the columns of the current record.
    for (int i = 1; i <= colNum; i++)
        System.out.println (paymentsInfo.getObject(i));
}

 // Releases the ProcObject.
 info._release();

 // Releases the AppObject. Since this is the last object to
 // share the connection this disconnects from the AppServer.
 account._release();
```

Example 5–3: Java Client Using Proxy Objects

PayeeList Class for Example 5–3 *(1 of 2)*

```
import java.sql.*;
import java.util.Vector;

/* An example of a simple implementation of payeeList INPUT result set. */
public class PayeeList extends com.progress.open4gl.InputResultSet
{
    private Vector rows;
    private int rowNum;
    private Vector currentRow;

    // Create a payee result set 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;
    }
```

PayeeList Class for Example 5–3 *(2 of 2)*

```
 // 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);
    }
}
```

Example 5–4: Java Input ResultSet Example

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`.

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.
```

Example 5–5: AppServer Application

5.4 Extending Proxy Objects

For the Java client, ProxyGen actually 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 that the Java client is not exposed to implementation details of the proxy. These delegating class 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` as well as the delegating classes `Account.java` and `Tax.java`. A skeleton of what the delegating class `Account` looks like can be found in Example 5–6.

Account Class (Delegating)

```
Public class Account
{
  protected AccountImpl m_accountImpl;

  public Account(connect-parameters)
  {
    m_accountImpl = new AccountImpl(connect-parameters);
  }
  public Add(add-parameters)
  {
    m_accountImpl.Add(add-parameters);
  }
  public Tax createAO_Tax()
  {
    return new Tax(m_accountImpl);
  }
  public Tax createPO_AccountInfo(AccountInfo-parameters)
  {
    return new AccountInfo(m_accountImpl, AccountInfo-parameters);
  }
  public _release()
  {
    m_accountImpl._release();
  }
  ...
}
```

Example 5–6: Delegating Classes

Note the `protected` member variable `m_accountImpl`. If a SubAppObject or a ProcObject is created, this variable must be passed to its constructor. Example 5–7 shows what the constructor in the `Tax` class looks like.

Tax Class Constructor

```
Public class Tax
{
  protected TaxImpl m_taxImpl;

  public Tax(AccountImpl accountImpl)
  {
    m_taxImpl = new TaxImpl(accountImpl);
  }

}
```

Example 5–7: SubAppObject/ProcObject Constructors

The member variables are `protected` rather than `private` to allow a Java client to extend these classes. Example 5–8 shows how a client might extend the `Account` class as well as the `Tax` class.

NOTE: To extend a SubAppObject or ProcObject, you must also extend the associated AppObject.

Account and Tax 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);
  }
}
```

Example 5–8: Class Extensions

5.5 Using Proxies In Multi-threaded Applications

If you access Open Client proxies in a multi-threaded application, this section provides information on the various states that AppServer connections and requests can assume during the running of an Open Client application. These states affect how your threads can interact.

5.5.1 Connection States

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

- **RUNNING** — The connection is in a state of RUNNING while an AppServer method is executing. No other AppServer methods can be executed when the connection is in the RUNNING state.
- **STREAMING** — The connection is in a state of STREAMING when there is at least one open output result set. AppServer methods cannot be executed when the connection is in the STREAMING state.

  NOTE: 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 in a state of IDLE when it is not in a RUNNING or STREAMING state.

5.5.2 Request States

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 can still be active if there is an open output result set (that is, the connection is in a STREAMING state).

The three request states include:

- **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.

5.5.3 Cancelling a Request

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

- Terminate a request that was initiated by any object in the proxy and that is currently running (normally during a long execution).
- Cancel any requests that might be queued for execution.
- Close any open result sets.

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

- **QUEUED** — Cancels the request and throws an Exception.
- **RUNNING** — Forwards a STOP request to the 4GL interpreter, which handles it according to normal 4GL STOP condition rules.
- **STREAMING** — Initiates the close of any currently open output result sets, but does not call the `close()` method. You must then execute the `close()` method on all affected output result sets to complete the closure. For large result sets, this is more efficient than using the `close()` method alone. For more information on the `close()` method, see the "Passing TEMP-TABLE Or TABLE-HANDLE Parameters" section.

NOTE: If there are no requests in the following states when you execute `_cancelAllRequests()`, the method has no effect.

5.5.4 Multiple-thread Access To ResultSets

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.

For more information, see Chapter 6, "Using SmartDataObjects From Java Clients."

5.6 Accessing Proxy Run-time Properties

The Open Client Runtime provides object `com.progress.open4gl.RunTimeProperties` that has several static methods that provide tracing, thread-control, Proxy server, and digital certificate management support for an Open Client application. The following sections provide information about:

- Methods For Tracing
- Methods For Thread Control
- Methods For Specifying a Proxy Server
- Methods For Managing Digital Certificates

5.6.1 Methods For Tracing

Tracing logs highlights of proxy execution, including data received by the client and data passed to the AppServer. The following methods provide tracing functionality to the Open Client application:

SYNTAX

```
RunTimeProperties.traceOn(String fileName, int level)
```

Sends all trace messages equal or greater than the value specified with *level* to the file specified with *fileName*. If *filename* is not specified, the output is Standard Error. The lower the value for *level*, the greater the information. If the *level* is not specified, the value equals 2.

SYNTAX

```
RunTimeProperties.traceOn(int level)
```

Sends all trace messages equal or greater than the value specified with *level* to Standard Error.

SYNTAX

```
RunTimeProperties.traceOn(String fileName)
```

Sends all trace messages of level 2 and greater to the file specified with *fileName*.

SYNTAX

```
RunTimeProperties.traceOn()
```

Sends all trace messages of level 2 and greater to Standard Error.

SYNTAX

```
RunTimeProperties.traceOff()
```

Turns off tracing.

SYNTAX

```
Boolean RunTimeProperties.isTracing()
```

Returns TRUE if tracing is turned on. Returns FALSE if tracing is turned off.

5.6.2 Methods For Thread Control

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 can 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.

These methods provide thread control to the Open Client Runtime:

SYNTAX

```
RunTimeProperties.setWaitIfBusy()
```

Waits until all currently executing AppServer requests are complete before executing this request.

SYNTAX

```
RunTimeProperties.setNoWaitIfBusy()
```

Do not wait for a currently executing AppServer request to finish. Instead throw an exception for any AppServer request that was made while the AppServer was executing another request.

SYNTAX

```
Boolean RunTimeProperties.getWaitIfBusy()
```

Checks the thread control setting to determine if `WaitIfBusy` state is set or cleared. Returns TRUE if `setWaitIfBusy()` was called and FALSE if `setNoWaitIfBusy()` was called.

5.6.3 Methods For Specifying a Proxy Server

You use the following methods if a Proxy Web server exists between the client and the Web server hosting AIA.

These methods provide Proxy Web server support to an Open Client application:

SYNTAX

```
RunTimeProperties.setProxyHost(String hostname)
```

Sets the Proxy Web server host to the value specified with *hostname*. This value can be a string with either the DNS name of the Proxy Web server or its dot-formatted IP address.

SYNTAX

```
RunTimeProperties.setProxyPort(int port)
```

Sets the Proxy server port to the value specified with *port*. This value can be an integer from 1 to 65536.

SYNTAX

```
RunTimeProperties.setProxyUserID(String proxyAccountName)
```

Sets the Proxy server user to the value specified with *AccountName*. This value can be string of up to 512 printable ASCIII characters.

SYNTAX

```
RunTimeProperties.setProxyPassword(String proxy AccountPassword)
```

Sets the Proxy server password to the value specified with *AccountPassword*. This value can be a string of up to 512 printable ASCII characters.

SYNTAX

```
String RunTimeProperties.getProxyHost()
```

Retrieves the current Proxy Web server hostname.

SYNTAX

```
int RunTimeProperties.getProxyPort()
```

Retrieves the current Proxy Web server port number.

SYNTAX

```
String RunTimeProperties.getProxyUserId()
```

Retrieves the current Proxy Web server user id.

5.6.4 Methods For Managing Digital Certificates

When using HTTPS to access the AppServer, you might be supplying digital certificates with your client application. These methods provide digital certificate management to the Open Client application:

SYNTAX

```
RunTimeProperties.setCertificateStore(String certStoreList)
```

Specifies the files that contain the Root Digital Certificates. If more than one directory, file, or both is specified, they must be separated by semicolons.

SYNTAX

```
String RunTimeProperties.getCertificateStore()
```

Retrieves the current certificate list set by `setCertificateStore()`.

SYNTAX

```
RunTimeProperties.setNoHostVerify(boolean verify)
```

Controls Domain Name checking. Specifying FALSE for `verify` enables Domain Name checking. Specifying TRUE disables Domain Name checking. The default value is TRUE (on).

SYNTAX

```
Boolean RunTimeProperties.getNoHostVerify()
```

Returns a boolean value that indicates the settings for Digital Certificate authentication options.

5.7 Building Java Applets As Open Clients

You can build a Java applet that runs as an Open Client. That is, the applet can make proxy method calls to remote AppServer procedures from the Web browser environment. However, you must include additional code to allow the Web browser to accept the Open Client applet for execution. For more information, see the Chapter 8, "Special Considerations For Java Applets."

5.8 Handling Errors

Java clients handle errors by catching Exceptions. To enable Java proxies to throw Exceptions, Progress establishes two `Exception` class hierarchies:

- **General 4GL Exceptions** — A hierarchy of `Exception` classes that extend the `Open4GLException` class. `Open4GLException`, in turn, extends the `com.progress.common.exception.ProException` class.
- **Output result set Exceptions** — A class `ProSQLException` that extends the standard `java.sql.SQLException` class.

In addition to the standard methods provided by `java.lang.exception` to get the message string, Progress provides the following method to obtain the error number:

SYNTAX

```
long getMessageID()
```

For information about additional methods provided by these Exception classes, see the Open Client Toolkit documentation in *Progress-Install-Directory*/java/doc.

The following sections describe each Exception class in detail.

General 4GL Exceptions

Table 5–8 shows the general 4GL Exception classes. The left column lists each Exception class and its position in the class hierarchy. The arrows illustrate this hierarchy. For example, RunTime4GLException is the super-class of several Exception classes such as RunTime4GLStopException.

Table 5–8: General 4GL Exceptions For Java Proxies *(1 of 2)*

Exception Class Hierarchy (Top → Bottom)	Description
Open4GLException	The root Exception for all Open Client errors. Used for miscellaneous client-side errors.
→ ConnectException	Error connecting to AppServer.
→ → HostUnknownException	Invalid NameServer host specified in the URL.
→ → InvalidNameServerPortException	Invalid NameServer port specified in the URL.
→ → MsgVersionInvalidException	Incorrect version for message received from NameServer.
→ → 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.
→ → NoSuchAppServiceException	Application Service specified in URL unknown to NameServer.
→ → NameServerMessageFormatException	Invalid format for message received from NameServer.

Table 5–8: General 4GL Exceptions For Java Proxies *(2 of 2)*

Exception Class Hierarchy (Top → Bottom)	Description
→ → BadUrlException	Invalid URL format in connection parameter.
→ → ConnectFailedException	Failed AppServer connection attempt. The message contains the specified reason for the failure.
→ RunTime4GLException	4GL run-time errors
→ → RunTime4GLErrorException	4GL ERROR condition
→ → RunTime4GLStopException	4GL STOP condition
→ → RunTime4GLQuitException	4GL QUIT condition
→ BusySessionException	AppServer busy servicing another request
→ OutputSetException	Output result set error, for example, trying to get a column value out of column order or trying to make an illegal type conversion
→ SystemErrorException	An unexpected error that indicates a Progress bug or an unusual environmental problem (such as running out of disk space)

Output Result Set Exceptions

Progress defines `ProSQLException` which is returned from all methods on an output TEMP-TABLE result set, including the SDOResultSet object:

```
ProSQLException class extends java.sql.SQLException
```

`ProSQLException` adds a method, `getProException()`, which returns an `OutputSetException`.

The SDOResultSet is an interface that maps to Progress SmartDataObjects running on the AppServer. For more information on SDOResultSet objects and how to handle Exceptions for them, see Chapter 6, "Using SmartDataObjects From Java Clients."

6

Using SmartDataObjects From Java Clients

Any Java application can access a remote Progress 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 general information on accessing AppServer functionality from a Java client, see Chapter 5, "Programming Java Clients."

The following sections provide information on:

- What Is a SmartDataObject (SDO)?
- Accessing Remote SmartDataObjects From Java
- Connecting To an AppServer Using an SDOAppObject
- Creating an SDOResultSet Object
- Working With SDOResultSet Objects
- Multi-threading
- Errors and Exceptions
- Choosing JDBC ResultSets Or SDOResultSets To Access Progress Data
- Accessing the Sample Application

6.1 What Is a SmartDataObject (SDO)?

A SmartDataObject is a Progress 4GL persistent procedure created using the SmartObject technology of the Progress AppBuilder. As such, it provides a number of SmartObject methods (implemented as 4GL internal procedures and functions) that allow a 4GL application to dynamically query and update Progress databases (and DataServers). It also implements the patented SmartObject messaging technology that allows it to communicate and exchange data with other types of Progress SmartObjects (also implemented in the 4GL). For more information on SmartDataObjects, see the *Progress AppBuilder Developer's Guide*, the *Progress ADM 2 Guide*, and the *Progress ADM 2 Reference*.

Using the Progress AppBuilder, a 4GL programmer can modify and extend the standard SmartDataObject methods to encapsulate any business logic needed to filter data into or out of a Progress-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.

6.2 Accessing Remote SmartDataObjects From Java

Figure 6–1 shows how a Java Open Client accesses a Progress 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.

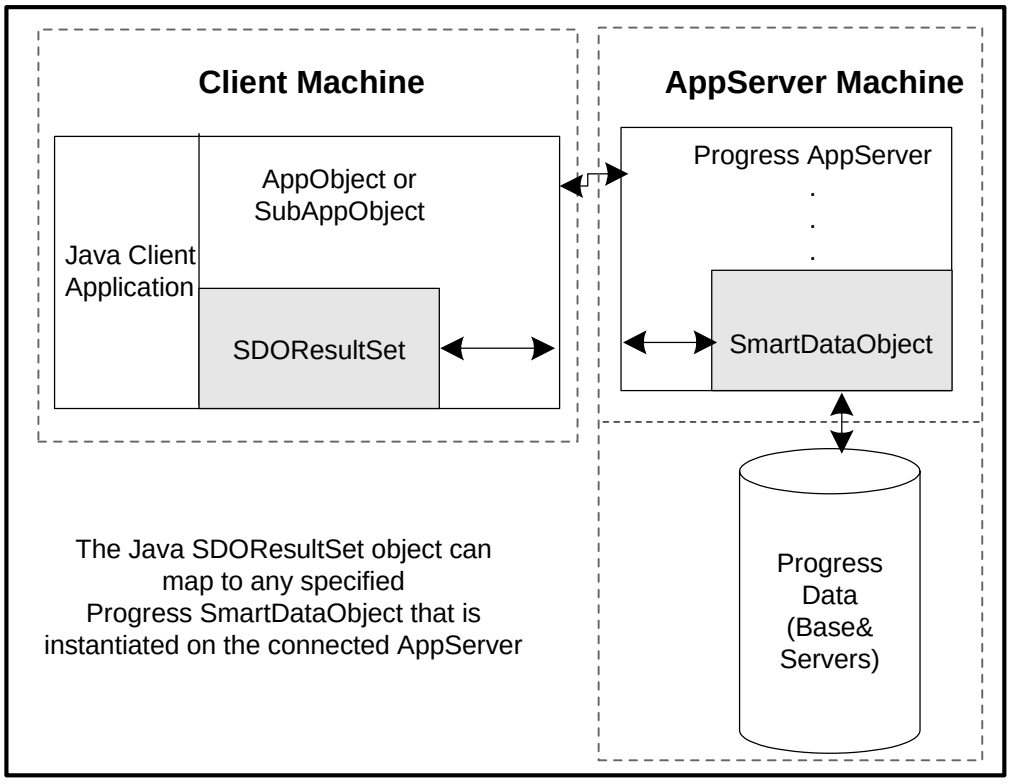

Figure 6–1: Java Open Client Access To a Remote SmartDataObject

6.2.1 Requirements For Accessing SmartDataObjects

To enable any Java client to access a remote Progress 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.

6.2.2 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".

Java Client Using an SDOAppObject

```
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 SDO 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 Progress 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.

6.2.3 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, come installed as part of the Open Client Runtime. For reference information on each class and interface, see the documentation in *`Progress-Install-Directory`*`\java\doc\*.html`. For information on ProxyGen, see Chapter 3, "Generating and Managing Proxies."

6.2.4 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** — An extended subset of the JDBC 2 (JDK 1.2) ResultSet interface that provides access to a SmartDataObject. Because all instances of a SmartDataObject present the same 4GL API, the one SDOResultSet class allows you to access any such instance. Note that SDOResultSet works with both JDK 1.1.x and JDK 1.2. Thus, much of the Java 1.2 functionality (standard JDBC ResultSet interface) is supported even with JDK 1.1.x.

  NOTE: SDOResultSet allows you to access only the default SmartDataObject API generated by the Progress AppBuilder. If you need to access custom programmer extensions to this API, you must access the SmartDataObject API directly. For more information, see Appendix A, "Accessing a SmartDataObject API Directly."

- **AppObject** or **SubAppObject** — An object that allows you to create an SDOResultSet (using `_createSDOResultSet()`) to access any remote SmartDataObject. You can access the SDOResultSet object from these proxy objects:

  – Any standard AppObject or SubAppObject that you generate in ProxyGen

  – A prebuilt `com.progress.open4gl.SDOAppObject` proxy that comes installed as part of the 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 Progress is a complete proxy that allows you to access any SmartDataObject without having to build a proxy in ProxyGen. It is essentially a complete AppObject built to access SDOResultSet objects only.

  Thus, if you only need to access SmartDataObjects on the AppServer and no other AppServer procedures, you can use this SDOAppObject out of the box. If you need to access other 4GL procedures on the AppServer in addition to the SmartDataObject, you can use any AppObject or SubAppObject generated with ProxyGen to access any SmartDataObject. For more information on AppObjects, see Chapter 4, "Programming Concepts."

- **SDOParameters** — An object that you can pass to the SDOResultSet constructor through the `_createSDOResultSet()` method. SDOParameters allows you to modify the default behavior of the SDOResultSet instance.

6.2.5 Developing and Deploying a SmartDataObject-aware Application

Follow these steps to develop and deploy Java applications that access SmartDataObjects:

1 ♦ Use the AppBuilder tool 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 Progress-compiled ADM2 (Application Development Model, Version 2) r-code files to a directory in the AppServer PROPATH. You can find these files in the *`Progress-Install-Directory`*`\gui\adm2` directory, wherever you have the Progress AppBuilder installed. To complete this step, you can copy these files to your AppServer *`Progress-Install-Directory`*`\tty\adm2` directory.

4 ♦ Deploy the Open4gl Client Runtime package on the client side. For more information, see Chapter 2, "Configuration and Deployment."

5 ♦ If your Java application needs only SmartDataObject access (it does not access other remote 4GL procedures), you are ready to write Java applications that access the SmartDataObject.

6 ♦ If your Java application needs to access 4GL procedures other than SmartDataObjects, use ProxyGen to create the proxies and deploy them on the client side (see Chapter 3, "Generating and Managing Proxies"). However, you do not have to map the SmartDataObject as a ProcObject. ProxyGen provides built-in access to SmartDataObjects in Java.

NOTE: If you add your own extensions (new 4GL methods) to a SmartDataObject, to access these extensions you must access the SmartDataObject as a separate ProcObject that you create in ProxyGen. This allows you to directly access the SmartDataObject as a Progress 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 Appendix A, "Accessing a SmartDataObject API Directly."

6.2.6 Writing a SmartDataObject-aware Java Client

Follow these steps 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.

4 ♦ Close the SDOResultSet using the `SDOResultSet.close()` method.

5 ♦ Release the SDOAppObject, AppObject, or SubAppObject using the *Appobject*.`_release()` method.

6.3 Connecting To an AppServer Using an SDOAppObject

The constructor for class SDOAppObject takes the same parameters as any AppObject constructor and connects to an AppServer in exactly the same way:

SYNTAX

```
public SDOAppObject
  ( String url,
    String userid,
    String password,
    String appserver-info
  );
```

```
public SDOAppObject
  ( String userid,
    String password,
    String appserver-info
  );
```

```
public SDOAppObject
  (
  );
```

The *url* parameter specifies the AppServer connection information. The *userid*, *password*, and *appserver-info* parameters specify any information required by the AppServer application at connection time. For more information on how to specify these parameters, see the information on connecting to an AppServer in Chapter 4, "Programming Concepts."

6.4 Creating an SDOResultSet Object

Once you have instantiated 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.

NOTE: 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.

You can call one of these overloaded versions of the `_createSDOResultSet()` method, which supports different combinations of the four factory method parameters, as follows:

SYNTAX

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

Uses the KEEP_ROWS scrolling mode and the default where and sort expressions.

SYNTAX

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

Uses the KEEP_ROWS scrolling mode.

SYNTAX

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

Uses whatever scrolling mode is specified by *parameters*. If *parameters* does not specify a scrolling mode, the default is used.

These are the possible parameters for the `_createSDOResultSet()` method:

- **String *sdoName***

  (Required) Specifies the filename of the SmartDataObject (`custSDO.w`, for example).

- **String *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 the *Progress ADM 2 Reference*.

- **String *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 the sort expression using one of the following syntax styles:

  SYNTAX

  ```
  "[ field | field-name ASCEND | field-name DESCEND
     [ , { field | field-name ASCEND | field-name DESCEND } ] ... ]"
  ```

  In this style, *field* is a valid database field referenced by the SmartDataObject and specifying no fields ("") is equivalent to null. Examples include:

  - "Name"
  - "Name, City"
  - "Name DESCEND, City, Zip"

  SYNTAX

  ```
  "[ field | field-name ASCEND | field-name DESCEND
     [ BY { field | field-name ASCEND | field-name DESCEND } ] ... ]"
  ```

  This style represents a valid 4GL sort clause (omitting the first BY keyword). Examples include:

  - "Name"
  - "Name BY City"
  - "Name DESCEND BY City BY Zip"

For more information on specifying this parameter, see the description of the setQuerySort method in the Beta *Progress ADM 2 Guide*.

- **SDOParameters** *parameters*

  Used for passing any additional initial parameters to the ResultSet object, such as the scrolling mode.

You only need to set values that differ from the defaults. The methods to set and get these values on the SDOParameters object follow:

SYNTAX

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

Sets the row ID of the first row to be fetched from the created and opened SDOResultSet. A null *rowId* is equivalent to not calling `setRowIdentity()` (see also, the `reOpenQuery(String` *rowId*) extended method in the "Miscellaneous Management Methods" section). It is usually 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` scrolling mode, see the "Understanding SDOResultSet Scrolling Modes" section.

The *rowId* value is equivalent to the value returned by the 4GL ROWID function and the `SDOResultSet.getRowIdentity()` method. For more information on ROWID, see the*Progress Programming Handbook*, or the appropriate *Progress DataServer Guide* for any DataServer accessed by the SmartDataObject. For more information on the `SDOResultSet.getRowIdentity()` method, see the "Miscellaneous Management Methods" section.

SYNTAX

```
String getRowIdentity()
```

Gets the row ID of the first row to be fetched from the created and opened SDOResultSet.

SYNTAX

```
void setFetchSize(int fetchsize)
```

Sets the number of rows the scrolling mechanism fetches each time it accesses the AppServer. The default is 200 rows.

SYNTAX

```
int getFetchSize()
```

Gets the number of rows the scrolling mechanism fetches each time it accesses the AppServer.

SYNTAX

```
void setStateless(boolean state)
```

Sets the Stateless mode for the SDOResultSet. Specify *state* as true for a Stateless SDOResultSet and false for a nonStateless SDOResultSet. For more information on the Stateless mode of SDOResultSet, see the "Understanding SDOResultSet Stateless Mode" section.

SYNTAX

```
boolean getStateless()
```

Gets the Stateless mode for the SDOResultSet. For more information on the Stateless mode of SDOResultSet, see the "Understanding SDOResultSet Stateless Mode" section.

SYNTAX

```
void setScrollingMode(SDOScrollingMode constant)
```

Sets the scrolling mode for the SDOResultSet specified as a class constant in com.progress.open4gl.SDOScrollingMode. 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 on these scrolling modes, see the "Understanding SDOResultSet Scrolling Modes" section.

SYNTAX

```
SDOScrollingMode getScrollingMode()
```

Gets the scrolling mode specified for the SDOResultSet, returned as a class constant in com.progress.open4gl.SDOScrollingMode. For more information on SDOResultSet scrolling modes, see the "Understanding SDOResultSet Scrolling Modes" section.

SYNTAX

```
void setPrefetchMaxRows(int maxRows)
```

Sets the maximum number of rows to be fetched when the scrolling mode is `SDOScrollingMode.PREFETCH`. 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 on using this method, see the "Understanding SDOResultSet Scrolling Modes" and "Understanding SDOResultSet Stateless Mode" sections.

SYNTAX

```
int getPrefetchMaxRows( )
```

Gets the value for the maximum number of rows to be fetched when the scrolling mode is `SDOScrollingMode.PREFETCH`. For more information on using this method, see the "Understanding SDOResultSet Scrolling Modes" and "Understanding SDOResultSet Stateless Mode" sections.

Understanding SDOResultSet Scrolling Modes

Different scrolling modes allow you to control response time, memory requirements, and the result set isolation level. The *isolation level* (in this context) is the visibility of modifications made to the result set 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. These are the values that you can pass for *constant*:

- **`SDOScrollingMode.PREFETCH`** — This is the default and only allowable mode for a `Stateless` SDOResultSet. In this mode, the whole result set 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 result set. This mode creates the highest level of isolation.

(For more information, see the information on visibility of updates in the "Updating SDOResultSet Objects" section.) The size of the result set is known from the beginning and does not change until the result set 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 result sets in combination with slow networks.

NOTE: If you are accessing a large result set, you might want to use the `setPrefetchMaxRows()` method to limit the number of rows that 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 are always in the `PREFETCH` scrolling mode. For more information, see the "Understanding SDOResultSet Stateless Mode" section.

- **`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 result set is closed or a row is refreshed or updated. The response time to get the first rows in `KEEP_ROWS` mode is shorter than in `PREFETCH` mode. `Stateless` SDOResultSets do not support this mode (see the "Understanding SDOResultSet Stateless Mode" section).

- **`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 result set. Therefore, this scrolling mode should be used when a very large result set 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 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 can also create a non`Stateless` SDOResultSet to access a stateless AppServer.

An SDOResultSet is in `Stateless` mode if `setStateless(true)` is called on the SDOParameters that the application passes to the `_createSDOResultSet()` class factory method.

The SDOResultSet `Stateless` mode has the following limitations:

- Only batch updates are allowed. For more information, see the "Using Batch Mode [Extension]" section.
- Stateless mode is only allowed with the `PREFETCH` scrolling mode. For more information, see the "Understanding SDOResultSet Scrolling Modes" section.

As with any SDOResultSet opened in `PREFETCH` mode, `Stateless` mode causes the SDOResultSet to return all of its rows to the client application at one time. This can pose performance problems for excessively large result sets. 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 to fetch the next set of rows for a different instance of the query. (For more information on the `reOpenQuery(String` *`rowId`*`)` method, see the "Miscellaneous Management Methods" section.) 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.

Note that 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 fetches the next nine (9) rows that have not yet been fetched, for a total of 10 returned for the query.

6.5 Working With SDOResultSet Objects

This section describes how to use the methods supported on SDOResultSet, including both the subset of standard JDBC 2 methods and SDOResultSet extensions to JDBC 2. For information on the full JDBC 2 standard, see the JDBC 2 specification and the JDK API description. This section also describes other objects and methods that affect SDOResultSet functionality and provides information about the following:

- Detaching SDOResultSet Objects [Extension]
- Navigating SDOResultSet Rows
- Getting SDOResultSet Column Values
- Updating SDOResultSet Objects
- Miscellaneous Management Methods
- Metadata Methods

In the following sections, **[Extension]** appears after the listed syntax for methods that are SDOResultSet extensions to JDBC 2. If an entire section describes methods that are all SDOResultSet extensions, **[Extension]** appears after the section title also.

To apply some of these extensions you must pass parameters to the SDOResultSet factory method (`_createSDOResult()`) that creates the SDOResultSet object.

6.5.1 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 result set and releasing the AppObject (disconnecting the AppServer), the Open Client can use and manipulate the result set 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 only exceptions include the following methods (described in other sections) that require an AppServer connection:

- **`reOpenQuery()`** — See the "Miscellaneous Management Methods" section.
- **`sendBatch()`** — See the information on `Batch` mode in the "Updating SDOResultSet Objects" section.
- **`sendBatchAndReOpen()`** — See the information on `Batch` mode in the "Updating SDOResultSet Objects" section.

Thus, before calling a method that requires an AppServer connection, you must attach the SDOResultSet to the same or a different AppObject that has a connection to the AppServer. The methods that provide this functionality include:

SYNTAX

```
void detachFromAppObj() [Extension]
```

Detaches from the AppObject that created the result set.

SYNTAX

```
void attachToAppObj(SDOFactory ProxyObject) [Extension]
```

Attaches to the AppObject, SubAppObject, or SDOAppObject specified by *`ProxyObject`*.

NOTE: ProxyGen-generated AppObjects and SubAppObjects, as well as the SDOAppObject proxy, all implement the SDOFactory interface.

SYNTAX

```
boolean isAttached() [Extension]
```

Returns `true` if the SDOResultSet is attached to an AppObject. Otherwise, it returns `false`.

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.

NOTE: 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 can also 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.

6.5.2 Navigating SDOResultSet Rows

These methods change or report the position of the cursor in the SDOResultSet:

SYNTAX

```
boolean previous()
```

Moves the cursor backwards.

SYNTAX

```
boolean next()
```

Moves the cursor forward.

SYNTAX

```
boolean first()
```

Positions the cursor on the first row.

SYNTAX

```
void beforeFirst()
```

Positions the cursor before the first row.

SYNTAX

```
boolean last()
```

Positions the cursor on the last row.

SYNTAX

```
void afterLast()
```

Positions the cursor after the last row.

SYNTAX

```
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.

SYNTAX

```
boolean relative(int n)
```

Moves the cursor *n* rows forward (backward if *n* is negative).

SYNTAX

```
boolean isAfterLast()
```

Is the cursor after the last row?

SYNTAX

```
boolean isBeforeFirst()
```

Is the cursor before the first row?

SYNTAX

```
boolean isFirst()
```

Is the cursor on the first row?

SYNTAX

```
boolean isLast()
```

Is the cursor on the last row?

SYNTAX

```
int getRow()
```

Gets the position (1-based) of the cursor.

6.5.3 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 Chapter 4, "Programming Concepts"). The column is specified by name (*columnName*) or number (*columnIndex*):

- **BigDecimal getBigDecimal(String columnName)**
- **byte[] getBytes(String columnName)**
- **String getString(String columnName)**
- **long getLong(String columnName)**
- **int getInt(String columnName)**
- **double getDouble(String columnName)**
- **boolean getBoolean(String columnName)**
- **java.sql.Date getDate(String columnName)**
- **Object getObject(String columnName)**
- **BigDecimal getBigDecimal(int columnIndex)**
- **byte[] getBytes(int columnIndex)**

- **`String getString(int columnIndex)`**
- **`long getLong(int columnIndex)`**
- **`int getInt(int columnIndex)`**
- **`double getDouble(int columnIndex)`**
- **`boolean getBoolean(int columnIndex)`**
- **`java.sql.Date getDate(int columnIndex)`**
- **`Object getObject(int columnIndex)`**

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 4GL and Java data types, see the sections on Progress/Java data type mapping for TEMP–TABLE fields and passing OUTPUT TEMP–TABLE parameters in Chapter 5, "Programming Java Clients."

6.5.4 Updating SDOResultSet Objects

To update an SDOResultSet, you must complete two general steps:

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 have updated columns (for example, using `next()`) but before you have committed 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 can also extend a transaction to include modifications to multiple rows by using an SDOResultSet in Batch mode or by remotely controlling an automatic transaction, if one is available on the AppServer. For more information on Batch mode, see the "Using Batch Mode [Extension]" section, and on automatic transactions, see the "Using Extended Transactions" section.

SmartDataObjects use optimistic concurrency control, which means that records are read without a lock. Locks are acquired only for the short duration of the transaction when modifications are actually 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 where modified by another user. If one of the columns 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 already locked, it also throws an `SDOModificationException`. For more information on `SDOModificationException`, see the "Errors and Exceptions" section.

Updating Column Values

These methods modify a column in an SDOResultSet according to the specified data type. All column access methods in SDOResultSet use the flat model (see Chapter 4, "Programming Concepts"). The column is specified by name (*columnName*) or number (*columnIndex*) and is set to the specified value (*value*):

- **void updateNull(String *columnName*)**
- **void updateBoolean(String *columnName*, boolean *value*)**
- **void updateInt(String *columnName*, int *value*)**
- **void updateLong(String *columnName*, long *value*)**
- **void updateDouble(String *columnName*, double *value*)**
- **void updateBigDecimal(String *columnName*, BigDecimal *value*)**
- **void updateString(String *columnName*, String *value*)**

- **void updateBytes(String *columnName*, byte *x*[])**
- **void updateDate(String *columnName*, java.sql.Date *value*)**
- **void updateObject(String *columnName*, Object *value*)**
- **void updateObject(int** *columnIndex*, **Object *value*)**
- **void updateNull(int *columnIndex*)**
- **void updateBoolean(int *columnIndex*, boolean *value*)**
- **void updateInt(int *columnIndex*, int** *value*)
- **void updateLong(int *columnIndex*, long *value*)**
- **void updateDouble(int *columnIndex*, double *value*)**
- **void updateBigDecimal(int *columnIndex*, BigDecimal *value*)**
- **void updateString(int *columnIndex*, String *value*)**
- **void updateBytes(int *columnIndex*, byte *x*[])**
- **void updateDate(int *columnIndex*, java.sql.Date *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 4GL and Java data types, see the sections on Progress/Java data type mapping for TEMP–TABLE fields and passing INPUT TEMP–TABLE parameters in Chapter 5, “Programming Java Clients.”

Inserting, Deleting, and Updating Rows

These methods manage operations on rows in the SDOResultSet:

SYNTAX

```
void moveToInsertRow()
```

Moves the cursor to a staging position for creating a new row.

SYNTAX

```
void moveToCurrentRow()
```

Moves the cursor from the insert row back to the current position.

SYNTAX

```
void insertRow()
```

Sends the newly created row to the AppServer.

SYNTAX

```
void deleteRow()
```

Deletes the row in the current position and sends the delete request to the AppServer.

SYNTAX

```
void updateRow()
```

Sends the updates for the current row to the AppServer.

SYNTAX

```
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()`.

SYNTAX

```
boolean rowInserted()
```

Returns `true` if the cursor is positioned on an inserted row position and the row is not yet sent to the AppServer.

SYNTAX

```
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.

NOTE: The only time the row is not also physically deleted on the AppServer is if you are using a batch update. For more information, see the "Using Batch Mode [Extension]" section.

SYNTAX

```
boolean rowUpdated()
```

Returns `true` if the cursor is positioned on a row that is modified but not yet sent to the AppServer.

Using Extended Transactions

As described earlier in this section, by default, each row modification (row update, insertion, and deletion method) executes as a single transaction in the SmartDataObject. You can extend a transaction to include multiple rows using `Batch` mode, where a `sendBatch()` method applies a set of row updates as a single transaction (see the "Using Batch Mode [Extension]" section). You can also 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 that underlies 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 the*Building Distributed Applications Using the Progress AppServer* manual.

NOTE: You should call `reOpenQuery()` after an explicit transaction roll-back in order to maintain a cache that is consistent with the database. The `reOpenQuery()` method can also be desirable after committing a transaction containing inserted rows so that these rows are visible within the result set. For more information on row visibility, see the "Visibility of Updates" section.

Using Batch Mode [Extension]

`Batch` mode allows an Open Client to group together several updates and send them to the AppServer in one transaction. This might be a better update strategy than generating a transaction for each update method, depending on your application requirements. Thus:

- 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 the *Building Distributed Applications Using the Progress AppServer* manual.
- 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 **only** apply modifications under `Batch` mode.

These are the SDOResultSet methods that support `Batch` mode:

SYNTAX

```
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.

SYNTAX

```
void sendBatch() [Extension]
```

Sends the current batch to the AppServer.

SYNTAX

```
void sendBatchAndReOpen() [Extension]
```

Sends the current batch to the AppServer, then reopens the result set. In `Stateless` mode, this method is more efficient than separately calling `sendBatch()` and `reOpenQuery()`. For more information, see the "Visibility of Updates" section.

SYNTAX

```
void cancelBatch() [Extension]
```

Undoes all the modifications of the current batch.

SYNTAX

```
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 result set is reopened.
- **Row Deletion** — Row deletions are visible.

Updates by external applications can typically 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 the data in the SDOResultSet. However, if you do not call `reOpenQuery()`, 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 4GL 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 4GL 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.

6.5.5 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:

SYNTAX

```
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.

SYNTAX

```
SDOResultSetMetaData getMetaData()
```

Returns the SDOResultSetMetaData object, which is an extension of the JDBC ResultSetMetaData object.

SYNTAX

```
void refreshRow()
```

Gets the latest version of the current row from the database.

SYNTAX

```
boolean wasNull()
```

Indicates if the last fetched value from a `get`*`Datatype`*`()` method equal to the 4GL 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 4GL 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 4GL unknown value (?) because valid integer values for the QUANTITY column must be positive.

SYNTAX

```
int findColumn(String columnName)
```

Returns the column index.

SYNTAX

```
int getFetchSize()
```

Returns the size (in number of rows) of the buffer the underlying implementation uses to fetch rows from the AppServer.

SmartDataObject Management Methods [Extension]

The following nonstandard SDOResultSet methods 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 4GL ROWID function and the `SDOResultSet.getRowIdentity()` method. For more information on ROWID, see the *Progress Programming Handbook* (or the *Progress DataServer Guide* for any DataServer accessed by the SmartDataObject).

SYNTAX

```
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.

SYNTAX

```
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()`. It is usually more efficient to call `reOpenQuery(String` *rowId*) than to call `reOpenQuery()`, then call `absolute(String` *rowId*) in order 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 the "Understanding SDOResultSet Scrolling Modes" section.

SYNTAX

```
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*).

SYNTAX

```
String getQuery() [Extension]
```

Returns the 4GL query that populated this SDOResultSet object.

SYNTAX

```
boolean absolute(String rowId) [Extension]
```

Repositions the cursor at the row specified by *rowId*.

SYNTAX

```
SDOInterface getSDOInterface() [Extension]
```

Provides direct access to the underlying 4GL methods of the SmartDataObject that are efficient for remote SmartDataObject access. The returned SDOInterface is a ProcObject that includes all of the 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 that you can delete only by calling `releaseSDOInterface()`. Calling `releaseSDOInterface()` on a non`Stateless` SDOResultSet has no effect. 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 the *Progress ADM 2 Guide* and the *Progress ADM 2 Reference*.

NOTE: A typical application does not require access to the SDOInterface directly. 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. Thus, use the SDOInterface directly only after carefully analyzing the alternatives.

SYNTAX

```
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()`.

6.5.6 Metadata Methods

The metadata information 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. Note that 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 Metadata Methods

The supported standard JDBC 2 methods include:

SYNTAX

```
int getColumnCount()
```

The number of columns in the SDOResultSet.

SYNTAX

```
String getColumnName(int columnIndex)
```

The name of the column.

SYNTAX

```
int getColumnType(int columnIndex)
```

The SQL type of the column.

SYNTAX

```
int isNullable(int columnIndex)
```

0 if this column is mandatory (does not allow the 4GL unknown value (?)); 1 if it is not mandatory (allows the 4GL unknown value (?)).

SYNTAX

```
int getColumnDisplaySize(int columnIndex)
```

The maximum display size width of this column in characters.

SYNTAX

```
String getColumnLabel(int columnIndex)
```

The 4GL dictionary label for this column.

SYNTAX

```
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.

SYNTAX

```
boolean isReadOnly(int columnIndex)
```

True if the column is an SDOResultSet read-only column.

SYNTAX

```
boolean isWritable(int columnIndex)
```

False if the column is an SDOResultSet read-only column.

SYNTAX

```
String getColumnTypeName(int columnIndex)
```

The 4GL data type name of the column.

Nonstandard Metadata Methods [Extension]

SDOResultSetMetaData extensions to the standard interface include the following features:

- Methods that expose metadata information that exists in the schema of the Progress database but does not have an equivalent standard (for example, the `getColumnInitialValue()` method).
- Methods that expose 4GL-to-Java mapping information (for example, the `getColumnJavaTypeName()` method).

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 list of supported nonstandard SDOResultSetMetaData methods follows:

SYNTAX

```
String getColumnValExp(int column) [Extension]
```

The 4GL dictionary validation expression.

SYNTAX

```
String getColumnValMsg(int column) [Extension]
```

The 4GL dictionary validation message.

SYNTAX

```
String getColumnFormat(int column) [Extension]
```

The 4GL dictionary display format.

SYNTAX

```
String getColumnInitalValue(int column) [Extension]
```

The 4GL dictionary initial value as a string.

SYNTAX

```
int getColumnProType(int column) [Extension]
```

The 4GL data type number. The return value corresponds to one of the class constants defined in `com.progress.open4gl.Parameter`. For more information, see Chapter 5, "Programming Java Clients."

SYNTAX

```
String getColumnJavaTypeName(int column) [Extension]
```

The Java type name.

6.6 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.

6.7 Errors and Exceptions

Two types of errors can occur working with SDOResultSet, depending on the category of method that is called:

1. Errors that are caused by a bug in the application (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.

2. 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 that allows you to handle the error programmatically.

This section describes the SDOResultSet Exception hierarchy and typical errors by method category.

6.7.1 SDOResultSet Exceptions

All SDOResultSet and SDOResultSetMetaData methods throw `com.progress.open4gl.ProSQLException`, which is the root of more specific Exceptions. Table 6–1 shows the full Exception hierarchy:

Table 6–1: Output Result Set Exceptions For SDOResultSet

Exception Class Hierarchy (Top → Bottom)	Description
`Java.sql.SQLException`	The root Exception for all JDBC 2 Exceptions
→ `com.progress.open4gl.ProSQLException`	The 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

The following Exceptions from this hierarchy allow you to handle normal program conditions:

- **`DeletedRowAccessException`**

  An attempt to access a deleted row generates this Exception. To prevent this Exception, 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 of the error information provided by `SDOModificationException` using the following methods:

  SYNTAX

  ```
  String getFields()
  ```

  Returns the list of fields (if any) related to this error message.

SYNTAX

```
String getTable()
```

Returns the name of the table (if any) related to this error message.

SYNTAX

```
String getText()
```

Returns the text of the current error message.

SYNTAX

```
void nextMessage()
```

Positions the cursor on the next error message.

For more information on the SDOModificationException class and its methods, see the `SDOModificationException.html` file in *`Progress-Install-Directory`*`\java\doc\`.

6.7.2 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
- **Metadata Methods** —The typical errors in this category occur when trying to get metadata information about a column that does not exist.

6.8 Choosing JDBC ResultSets Or SDOResultSets To Access Progress Data

Progress Software supports a JDBC interface directly to the Progress database. Some applications might benefit from using the direct JDBC interface to Progress rather than SDOResultSets, and others might benefit more from using SDOResultSets.

JDBC access to the Progress database should be used in the following cases:

- There is no 4GL business logic (such as 4GL triggers) that must be used.
- Two-tier access (client/server) is sufficient.
- The client side application requires full JDBC support.

SmartDataObject result sets should be used in the following cases:

- 4GL 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.

6.9 Accessing the Sample Application

Progress comes installed with a sample SmartDataObject application, including a small Java Open Client application and the generated SmartDataObject ready to deploy. For more information, see the `readme.htm` file in the following samples directory:

```
Progress-Install-Directory\src\samples\open4gl\SDO
```

7

Programming ActiveX Controller Clients

For the ActiveX client, ProxyGen generates a proxy as a set of ActiveX Automation objects to access Progress AppServer functionality. The proxy follows COM conventions for method names, data type mapping, and error handling. ProxyGen also generates a Type Library that shows the methods available for the AppObject, each related SubAppObject, and each related ProcObject that defines the proxy.

The default language used in this chapter to illustrate ActiveX client programming is Visual Basic (VB), with additional references to C++. However, any ActiveX Controller can act as an ActiveX client to the Progress AppServer.

The sections in this chapter describe:

- Accessing Proxies As Automation Objects
- Connecting To an AppServer
- Data Type Mapping
- Passing TEMP-TABLE and TABLE-HANDLE Parameters
- Accessing Proxy Run-time Properties
- Handling Errors
- Payment Account Example

7.1 Accessing Proxies As Automation Objects

You can access an ActiveX proxy object using either early or late binding.

This section contains information about:

- Using Early Binding
- Using Late Binding
- Managing Object Versions

7.1.1 Using Early Binding

For early binding, the objects must be known at compile time. In VB, for example, you must add the Type Library generated by ProxyGen (and embedded in the `.dll` file) to the VB project references list. To do this, choose Project→ References from the menu bar in Microsoft Visual Basic. The name for the library is either the COM Object Description that you can specify in ProxyGen, or *`AppObject-name`*Lib if you don't specify the COM Object Description.

Once your objects are known to the compiler, you can then write code using an early binding reference. For example, if you have a proxy object, `Account`, you might specify the following reference to create the object in VB:

```
Dim myObj as AccountLib.CAccount   'Library name -> AccountLib
Set myObj  = new CAccount          'Object name -> CAccount
```

Note the naming conventions shown in Table 7–1, where *AppObject-name* is the name of an AppObject specified in ProxyGen.

Table 7–1: Proxy Component Names Generated By ProxyGen

Proxy Component	Generated Name
Type Library	*`AppObject-name`*Lib
AppObject	C*`AppObject-name`*
SubAppObject	CAO_*`AppObject-name`*
ProcObject	CPO_*`AppObject-name`*

For more information on, proxy naming conventions, see Chapter 3, "Generating and Managing Proxies."

7.1.2 Using Late Binding

For late binding, you use the Prog ID specified in ProxyGen to identify the object, which (by default) is the combination of the package and name of the AppObject. For example, for the same `Account` object:

```
Dim myObj as Object
Set myObj = CreateObject ("proxy.Account")  ‘registered Prog ID
```

7.1.3 Managing Object Versions

If you choose to generate a new GUID for an AppObject in ProxyGen (select the **Assign New Object IDs** check box in the Generate/Preferences dialog boxes), ProxyGen generates a new CLSID for the proxy object. However, ProxyGen registers the AppObject with the same ProgID as the previous version unless you also do one of the following:

- Change the AppObject name (Name field of the AppObject tab in the ProxyGen main window).
- Change the package name (Package field in the Generate/Preferences dialog boxes) so *`Package.AppObject-name`* is a new, unique value.
- Change the Prog ID directly (Prog ID field in the Generate/Preferences dialog boxes)

7.2 Connecting To an AppServer

Once you have created the proxy AppObject Automation object, you can establish an AppServer connection for it by executing the `OC_Connect( )` method. This method has the following syntax:

SYNTAX

```
Sub OC_Connect
( [ url As String ] ,
  [ userId As String ] ,
  [ Password As String] ,
  [ appServerInfo As String]
)
```

The *url* parameter specifies the AppServer connection information. The *userId*, *Password*, and *appServerInfo* parameters specify any information required by the AppServer application at connection time. For more information on how to specify these parameters, see the information on connecting to an AppServer in Chapter 4, "Programming Concepts."

7.3 Data Type Mapping

Table 7–2 lists the mapping between Progress and VB data types for method parameters.

NOTE: If you are writing a C++ client, it sees all of the data types in this table as a `Variant` data type.

Table 7–2: ActiveX Data Types In Progress and VB *(1 of 2)*

Progress Data Type	Visual Basic Data Type
CHARACTER	`String`
COM-HANDLE	`Long`
DATE[1]	`Double`
DECIMAL	`Double`
INTEGER	`Long`
LOGICAL	`Boolean`

Table 7–2: ActiveX Data Types In Progress and VB *(2 of 2)*

Progress Data Type	Visual Basic Data Type
MEMPTR	`Variant` (byte array)
RAW	`Variant` (byte array)
RECID	`Long`
ROWID	`Variant` (byte array)
TABLE-HANDLE	`ProTempTable`
TEMP-TABLE	`ProTempTable`
WIDGET-HANDLE	`Long`

[1] Dates earlier than July 1, 1600 do not pass accurately between ActiveX open clients and the Progress AppServer. The date gains or loses several days when passed in either direction.

NOTE: Open Clients do not support the Progress BUFFER data type.

7.3.1 Progress Data Type Requirements

Neither COM-HANDLE, WIDGET-HANDLE, RECID, or ROWID 4GL data types are valid outside of the Progress environment. Their use is restricted to obtaining the value from a Progress procedure and being able to pass it back to another Progress procedure.

`ProTempTable` is an Automation object that is part of the Progress Open Client Runtime, packaged in `o4glActiveX.dll`. You must register this object on the ActiveX client system. You can do this by running Microsoft's `RegSvr32.exe` on `o4glActiveX.dll` after you copy the Open Client Runtime to your client machine. For more information, see Chapter 2, "Configuration and Deployment."

NOTE: For early binding in VB (see the "Using Early Binding" section), the name of this object is `ProO4glActXiveXLib.CProTempTable` and Progress.TempTable.2 is the Prog ID.

If any Progress parameter can be the unknown value (?), the ActiveX data type must be set to `Variant`, rather than the type shown above, so that its value can be null (`Null` in VB, `VT_NULL` in C++). For C++ programmers, input/output or output parameters must have the `VT_BYREF` flag set in the `Variant`.

7.3.2 Data Type Conversions

In general, an Automation Controller might attempt any possible and reasonable conversion of parameter data passed to a method call on an Automation object. For example, if a method takes an integer parameter and the VB code passes a string with the value of "12", the Automation Controller (VB) can do one of the following:

- Return a data type mismatch error
- Convert the string value "12" to the integer value 12 before passing it to the Automation object as the parameter value.

Some Automation objects do not do automatic conversions. Therefore to avoid errors, use of the data types specified in Table 7–2 is recommended.

NOTE: There can be other unexpected behavior exhibited by an Automation controller. In some versions of VB, for example, if a `Variant`, converted to a specific data type (such as `CStr(Var)`), is passed as an output parameter, VB treats the parameter like a `String` constant and does not alert you that the result is unsuitable as an output parameter. Thus, it runs the method, but does not store the output value in the `Variant`.

7.4 Passing TEMP-TABLE and TABLE-HANDLE Parameters

Progress supports both TEMP-TABLE and TABLE-HANDLE parameters. Both parameter types map to a `ProTempTable` Automation object. The client uses `ProTempTable` objects to hold the source for input TEMP-TABLE parameters and to provide methods that it can use to access data from output TEMP-TABLE parameters. `ProTempTable` has a Prog ID of `Progress.ProTempTable`. When you reference this object in Visual Basic for early binding, the object name is `Proo4glActiveLib.CProTempTable`.

The following sub-sections references methods on the `CAccount` ProcObject described in a previous chapter (see Chapter 4, "Programming Concepts"). These include `SetDirectDeposit()` and `GetDirectDeposit()`. Each method takes a TEMP-TABLE parameter, one for input and the other for output.

This section provides information about:

- Passing a TEMP-TABLE OR TABLE-HANDLE As an Input Parameter
- Providing Metadata For INPUT TABLE-HANDLE Parameters
- Passing a TEMP-TABLE Or TABLE-HANDLE As an Output Parameter

7.4.1 Passing a TEMP-TABLE OR TABLE-HANDLE As an Input Parameter

To pass an input parameter, you must create an object whose methods the proxy can call to obtain the data. In a Visual Basic program, you can provide one of three types of objects:

- A DAO Recordset object.
- An ADO Recordset object.
- An Automation Object that is written in VB or another language and that explicitly implements a Progress-specified interface-`IClientTempTable`.

Whatever mechanism you use, the proxy determines what kind of object you are passing and calls the appropriate methods to access it.

NOTE: This section also applies to the input side of input/output TEMP-TABLE parameters.

The following sections provide information about:

- Choosing an Object Type to Pass
- Input TEMP-TABLE Parameter Examples For ActiveX

Choosing an Object Type to Pass

If you have access to ADO technology, this is the most flexible mechanism, because you can obtain an ADO Recordset in two ways:

- Execute a query on a local database to initialize the Recordset.
- Build an ADO Recordset in memory.

If you do not have access to ADO technology, and the input data resides in a local database, using a DAO Recordset is your best alternative. Otherwise, you must use the `IClientTempTable` interface.

The source object (the DAO, ADO, or `IClientTempTable` object) is not passed directly as a parameter. Instead, the client application creates a `ProTempTable` object and sets its `DataSource` property to one of these source objects.

The implementation of an Automation object that supports the `IClientTempTable` interface can obtain the data in whatever way is necessary. For example, it might read the data from an ActiveX Grid control, obtain it from an array variable or load it from of a file. You can find an example of how to implement this interface installed with the Open Client Runtime and located in the directory: *`Progress-Install-Directory`*`\src\samples\open4gl\IClientTempTable.`

The IClientTempTable interface has the following methods:

SYNTAX

```
MoveNext (Boolean moreRecs)
```

Changes the current record to the next row of the Recordset. The current record is initialized to the record before the first record of the set. Therefore, the first time MoveNext() is called it moves to the first record. *moreRecs* is an output parameter that is set to FALSE when the end of the Recordset is reached.

NOTE: Make sure the result set 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 TEMP-TABLE. Only rows after the current cursor position are passed to the AppServer.

SYNTAX

```
GetValue(short fldIndex, Variant value)
```

Gets the *value* for the column specified with *fldIndex*,a 1-based index. This must map to the order in which the fields are defined in the 4GL TEMP-TABLE. For input, a 4GL array field is viewed as a flattened set of columns (see Chapter 4, "Programming Concepts"). This is generally convenient in VB because the likely data sources for input Recordsets (such as, grids, arrays, or Access databases) do not themselves support array fields.

Input TEMP-TABLE Parameter Examples For ActiveX

Example 7–1 shows a sample of VB code that takes a TEMP-TABLE input parameter as an ADO Recordset.

```
Dim objAcct As AccountLib.CAccount
Dim ProTT As ProTempTableLib.CProTempTable
Dim rs As ADOR.Recordset

Set objAcct = New CAccount
Call objAcct.OC_Connect ("AppServer://myhost:2054", "guest",
 "guest", "")

' Create an ADO Recordset with three column and two rows
Set rs = CreateObject("ADOR.Recordset")
Call rs.fields.Append("AcctNum", adInteger)
Call rs.fields.Append("Amount", adDouble)
Call rs.fields.Append("StartDate", adDate)

Call rs.Open(, , adOpenDynamic)

Call rs.AddNew("AcctNum", 1)
Call rs.Update("Amount", 1000.11)
Call rs.Update("StartDate", 10 / 26 / 53)

Call rs.AddNew("AcctNum", 2)
Call rs.Update("Amount", 2000.22)
Call rs.Update("StartDate", 1 / 1 / 2000)

SetProt = new CProtempTable
ProTT.DataSource = rs
‘ The proxy calls back through the ADO to get the input data
Call objAcct.SetDirectDeposit (ProTT)
```

Example 7–1: ActiveX TEMP-TABLE Input Parameter Using ADO

Example 7–2 shows a sample of VB code that takes a TEMP-TABLE input parameter as a DAO Recordset.

```
Dim objAcct As AccountLib.CAccount
Dim ProTT As ProTempTableLib.CProTempTable
Dim MyDb As Database
Dim RecSet As Recordset

Set objAcct = New CAccount
Call objAcct.OC_Connect ("AppServer://myhost:2054", "guest",
 "guest", "")

Set MyDb = OpenDatabase("temp.mdb")
Set RecSet = MyDb.OpenRecordset("DirectDeposit")
Set VBTT = New VBTempTable

ProTT.DataSource = RecSet
‘ The proxy calls back through the DAO RecSet to get the input data
Call objAcct.SetDirectDeposit (ProTT)
```

Example 7–2: ActiveX TEMP-TABLE Input Parameter Using DAO

Example 7–3 shows a sample of VB code that takes a TEMP-TABLE input parameter as an `IClientTempTable` Automation object. In this example, `vbtt.VBTempTable` is an object that implements `IClientTempTable`.

NOTE: You must implement your own `IClientTempTable` object similar to the example `VBTempTable` class installed with the Open Client Runtime (see "Choosing an Object Type to Pass" earlier in this section).

```
Dim objAcct As AccountLib.CAccount
Dim VBTT As vbtt.VBTempTable ’An object that implements IClientTempTable
Dim ProTT As ProTempTableLib.CProTempTable

Set objAcct = New CAccount
Call objAcct.OC_Connect ("AppServer://myhost:2054", "guest",
 "guest", "")

Set ProTT = New CProTempTable

ProTT.DataSource = VBTT
‘ The proxy calls back through VBTT to get the input data
Call objAcct.SetDirectDeposit (ProTT)
```

Example 7–3: ActiveX TEMP-TABLE Input Parameter Using IClientTempTable

7.4.2 Providing Metadata For INPUT TABLE-HANDLE Parameters

For TEMP-TABLE parameters, the proxy calls the `MoveNext()` and `GetValue()` methods on the `ProTempTable` to get the data that needs to be passed to the AppServer methods. The metadata is already build into the proxy. For a TABLE-HANDLE parameter, the proxy must first get the metadata from the `DataSource` object in the `ProTempTable`. Then the proxy calls the `MoveNext()` and `GetValue()` methods on the `ProTempTable` to get the data that needs to be passed to the AppServer methods. You can provide the metadata in one of the following ways:

- Using a DAO or ADO `DataSource`:

  If the data source is a standard DAO or ADO object, the proxy can obtain the metadata automatically. When the proxy needs the schema information, it asks the DAO or ADO object for it. The DAO or ADO data types obtained from these objects map to Progress data types as shown in Table 7–3 and Table 7–4. Any data type not specified is unsupported.

Table 7–3: Mapping Between Progress and DAO Data Types

Progress 4GL Type	DAO Type
CHARACTER	`dbChar, dbMemo, dbText`
INTEGER	`dbByte, dbInteger, dbLong`
DECIMAL	`dbDouble, dbCurrency, dbSingle`
LOGICAL	`dbBoolean`
DATE	`dbDate`
RAW	`dbBinary, dbLongBinary, dbVarBinary`

Table 7–4: Mapping Between Progress and ADO Data Types

Progress 4GL Type	ADO Type
CHARACTER	adBSTR, adChar, adLongVarChar, adLongVarWChar, adVarChar, adVarWChar, adWChar
INTEGER	adInteger, adSmallInt, adTinyInt, adUnsignedInt, adUnsignedSmallInt, adUnsignedTinyInt, adError
DECIMAL	adDouble, adCurrency, adSingle
LOGICAL	adBoolean
DATE	adDate, adDBDate
RAW	adBinary, adLongVarBinary, adVarBinary

This methodology does not support array fields. If the application requires that the resulting server-side TEMP–TABLE have an array field, the application code must provide a mapping between the flattened set of DAO/ADO fields and the array fields of the TEMP–TABLE. You can do this by calling a `ProTempTable` method as follows:

SYNTAX

```
SpecifyArrayField(startFld as Integer, len as Integer)
```

Identifies the array where *`startFld`* is the 1-based index of the field in the DAO/ADO object (based on the flattened field model) and *`len`* is the length of the array.

You must call this method after the `DataSource` is specified and before the proxy method call. You must call the method once for each array field that exists in the TEMP–TABLE. For example, if the TEMP–TABLE has the following schema, you would call the method twice:

```
SalesRep, Name, Region, Month-Quota[12], Telephone[3]
Call SpecifyArrayField(4, 12)
Call SpecifyArrayField(5,3)
```

- Using the IClientTempTableMetaData interface

  There is an interface called IClientTempTableMetaData in ProO4glActiveXLib. This would be used in conjunction with IClientTempTable. In other words, if you are passing a TABLE-HANDLE parameter where the DataSource is an IClientTempTable object, that same object must also implement the IClientTempTableMetaData interface. Static TEMP–TABLE parameters can still use an object that only implements the IClientTempTable interface.

  This new interface contains four properties that can return the metadata for the input TABLE-HANDLE:

```
FieldCount as Integer
Name (fldIndex as Integer) as String
Type (fldIndex as Integer) as Integer
Extent (fldIndex as Integer) as Integer
```

  The *FldIndex* is the 1-based index of the field as defined by the array model.

  The proxy requests the FieldCount, then the Name, Type and Extent for each field before it sends its request to the AppServer. As with TEMP–TABLE data, the metadata information can be obtained in whatever way the application requires.

- Using the CFields/CField objects obtained from a previous output TEMP–TABLE parameter

  If the input result set is an updated version of a result set that was first obtained from the AppServer through an OUTPUT parameter, the schema obtained on output can be passed in as the schema for the input table. This schema is available from the CFields and CField objects associated with CProTempTable. 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. This is accomplished by calling another new ProTempTable method:

  SYNTAX

```
UseFieldsForMetaData(Object fields)
```

  fields

  A CFields object that was obtained from a CProTempTable object. If a null object is passed, this setting is cleared, and some other source of metadata must be supplied.

This method must be called after the DataSource is specified and before the proxy method call is made. Also, you must make this call even if the CProTempTable object is the same instance that was used for the original output parameter and from which the CFields object is obtained.

7.4.3 Passing a TEMP-TABLE Or TABLE-HANDLE As an Output Parameter

In order to get back TEMP-TABLE data from an AppServer, you must create a ProTempTable object and pass it as a parameter, without needing to set the DataSource property. On return, you can call methods on the ProTempTable object directly to retrieve the data.

If the parameter is a TABLE-HANDLE, the value returned can be unknown or undefined. As a result, calling Move Next() or Fields() on CProTempTable results in an error and the application must be able to handle this situation.

The interface that ProTempTable implements is similar to the ADO and DAO interfaces. The interface has methods to get both the data and the metadata of the TEMP-TABLE. For this process, the following objects are used:

- CProTempTable Object
- CFields Object
- CField Object

CProTempTable Object

The CProTempTable object provides the following methods and properties for getting output data:

SYNTAX

```
Close()
```

Closes the Recordset (used for output parameters only). Invoke this method if you want to close the Recordset before reading all of the rows.

NOTE: Until you read all rows or invoke this method for the output Recordset, you can call no other Open Client proxy method. For more information, see the sections on streaming in Chapter 4, "Programming Concepts."

SYNTAX

```
Variant GetRow()
```

Returns a `Variant` that contains a `Variant Array` with one `Variant` for each field of the current row. Only one row can be returned at a time. If the TEMP-TABLE contains an array field, it is flattened such that each element is returned as another `Variant` in the array. You can only call this method once for each row. Unlike the similar DAO or ADO method, after the data is returned, the Recordset cursor is not automatically moved to the next row, and you must call `MoveNext()` to move to the next row.

SYNTAX

```
Boolean MoveNext()
```

Changes the current record to the next row of the TEMP-TABLE. If `MoveNext()` has not been called before, it assumes we are at the beginning of the set and `MoveNext()` moves to the first record. It returns false when there are no more rows.

NOTE: If the parameter is a TABLE-HANDLE, the value returned can be unknown or undefined. In this case, calling `MoveNext()` returns an error.

SYNTAX

```
CFields Fields
```

A collection object that holds a set of `CField` objects, where each `CField` object provides access to the data and metadata of a field in the TEMP-TABLE. This property is read-only.

SYNTAX

```
Object DataSource
```

For an input TEMP-TABLE parameter, a property that you can set to an ADO, DAO, or `IClientTempTable` object from which a proxy can obtain the data. For an output TEMP-TABLE parameter, this property is ignored.

CFields Object

The `CFields` object provides the following methods and properties:

SYNTAX

```
CField Item(Variant index)
```

Returns a `CField` object for the specified field. The `index` parameter can either be a number or a string. If it is a number, it represents the 1-based field index which must correctly map to the order in which the fields are defined in the 4GL TEMP-TABLE. Otherwise, it must be the field's name.

This method uses the Progress array model to access result set columns. For more information on the array model and flat model, see Chapter 4, "Programming Concepts."

If you specify the name for an array field, specify the simple name without any array element reference. For example, specify `Months`, not `Months[1]`.

This is the default method for the object.

SYNTAX

```
short Count
```

The number of fields in the TEMP-TABLE (including array fields). Each array field counts as one because it is based on the Progress array model.

CField Object

The `CField` object provides the following methods and properties:

SYNTAX

```
Variant Value( [ integer index ] )
```

The value of the field returned as a `Variant`. If the field is a scalar value, `index` can be 0, 1 or left out. If the field is an array, `index` is a 1-based index into the array.

Depending on your version of Visual Basic, if you access `Value()` indirectly without first setting a `CField` object, you must provide a value for *`index`*. Otherwise, as in this example (where `proTT` is a `CProTempTable` object), VB returns a run-time error indicating that there are an invalid number of parameters:

```
x = proTT.Fields.Item(1).Value
```

However, the following example executes without error in VB:

```
x = proTT.Fields.Item(1).Value(0)
```

You can also access `Value()` directly on a `CField` object with or without an index in VB, again without error:

```
set fld = proTT.Fields.Item(1)
x = fld.Value
```

You can invoke this method only once on the same `CField` object, and you must obtain each field value in order. For more information, see the section on output result sets in Chapter 4, "Programming Concepts." This is the default method for the object.

NOTE: The limitation allowing you to call `Value()` only once per field might cause errors when debugging your VB application. For example, if you hold your mouse cursor over the call to the `Value()` method in order to display the value, in tooltip fashion, this causes an error if the code has already been executed. In a similar manner, an error results if you enter `field.Value()` in the watch window twice.

SYNTAX

```
String Name
```

The name of the field.

SYNTAX

```
short Type
```

Data type of the field. A set of constants accessible from VB are defined as part of `ProTempTableLib` called `ProDataTypeConstants`. Table 7–5 lists the constants and their values.

Table 7–5: ActiveX Data Types For TEMP-TABLE Output Parameters

ProDataTypeConstants	Values
ProChar	1
ProDate	2
ProLogical	3
ProInteger	4
ProDecimal	5
ProRecid	7
ProRaw	8
ProWidgetHandle	10
ProRowid	13
ProComHandle	14

SYNTAX

```
integer Extent
```

The field's extent. This returns 0 for non-array fields.

Output TEMP-TABLE Parameter Example for ActiveX

Example 7–4 shows how a VB client can retrieve the first two records of direct deposit information for a particular account.

```
Dim objAcct As AccountLib.CAccount
Dim ProTT As ProTempTableLib.CProTempTable
Dim flds As ProTempTableLib.CFields
Dim fld(3) As ProTempTableLib.CField
Dim DirDeposit(2, 3) ‘ This is Variant type by default

Set objAcct = New CAccount
Call objAcct.OC_Connect ("AppServer://myhost:2054", "guest", "guest", "")

Set ProTT = New CProTempTable

Call objAcct.GetDirectDeposit(ProTT)

‘ Get an array of Field objects
flds = ProTT.Fields
For j = 1 to 3 ‘ For each of 3 columns
    fld(j) = flds.Item(j)
Next j

For i = 1 to 2  ‘ For each of 2 rows
    ProTT.MoveNext()
    For j = 1 to 3 ‘ For each of 3 columns
        DirDeposit(i, j) = fld(j).Value
    Next j
Next i
ProTT.Close()
```

Example 7–4: ActiveX TEMP-TABLE Output Parameter Using ProTempTable

7.5 Accessing Proxy Run-time Properties

The Open Client Runtime comes installed with an Automation object that has a Prog ID of `Progress.RunTimeProperties`. It is packaged in `o4glActiveX.dll`. You must register the object as an ActiveX object on the client system. You can do this by running Microsoft's `RegSvr32.exe` on `o4glActiveX.dll` after you copy the Open Client Runtime to your client machine. For more information, see Chapter 2, "Configuration and Deployment."

NOTE: For early binding in VB (see the "Using Early Binding" section), the object name is `ProO4glActiveXLib.CProRunTimeProps` and `Progress.RunTimeProperties.2` is the Prog ID.

The Automation object has several methods that provide tracing, thread control, Proxy server, and digital certificate management support to the client. The following sections provide information about:

- Methods For Tracing
- Methods For Thread Control
- Properties For Specifying a Proxy Server
- Properties For Managing Digital Certificates

7.5.1 Methods For Tracing

Tracing logs highlights of proxy execution, including data received by the client and data passed to the AppServer. The following methods provide tracing functionality to the Open Client application:

SYNTAX

```
traceOn(String fileName, int level)
```

Sends all trace messages equal or greater than the value specified with *level* to the file specified with *fileName*. If *filename* is not specified, the output is Standard Error. The lower the value for *level*, the greater the information. If the *level* is not specified, the value equals 2.

SYNTAX

```
traceOnDefaultLevel(String filename)
```

Sends trace information to the file specified with *filename*. If no file is specified, the output is Standard Error.

SYNTAX

```
traceoff()
```

Turns off tracing.

SYNTAX

```
Boolean isTracing()
```

Returns TRUE if tracing is turned on. Returns FALSE if tracing is turned off.

NOTE: For ActiveX clients, tracing information is always written to a file.

7.5.2 Methods For Thread Control

Thread control specifies how multi-threaded requests to the AppServer are handled. These methods are only useful in multi-threaded clients, such as C++. The following methods provide thread control to the Open Client application:

SYNTAX

```
setWaitIfBusy()
```

Waits until all currently executing AppServer requests are complete before executing this request.

SYNTAX

```
setNoWaitIfBusy()
```

Do not wait for a currently executing AppServer request to finish. Instead throw an exception for any AppServer request that is made while the AppServer is executing another request.

SYNTAX

```
Boolean getWaitIfBusy()
```

Checks the thread control setting to determine if `WaitIfBusy` state is set or cleared. Returns TRUE if `setWaitIfBusy()`is called and FALSE if `setNoWaitIfBusy()` is called.

7.5.3 Properties For Specifying a Proxy Server

You use the following properties if a Proxy Web server exists between the client and the Web server hosting AIA.

These read/write properties provide Proxy Web server support to an Open Client application:

SYNTAX

```
ProxyHost As String
```

The Proxy Web server host. This value can be a string with either the DNS name of the Proxy Web server or its dot-formatted IP address.

SYNTAX

```
ProxyPort As Integer
```

The Proxy server port. This value can be an integer from 1 to 65536.

SYNTAX

```
ProxyUserID As String
```

The user Id needed to authenticate the Proxy server. This value can be string of up to 512 printable ASCIII characters.

SYNTAX

```
ProxyPassword As String
```

The password needed to authenticate the Proxy server. This value can be a string of up to 512 printable ASCII characters.

7.5.4 Properties For Managing Digital Certificates

When using HTTPS to access the AppServer, you might be supplying digital certificates with your client application. These read/write properties provide digital certificate management to the Open Client application:

SYNTAX

```
CertificateStore As String
```

Specifies the files that contain the Root Digital Certificates. If more than one directory, file, or both is specified, they must be separated by semicolons.

SYNTAX

```
NoHostVerify As Boolean
```

Controls Domain Name checking. FALSE enables Domain Name checking. TRUE disables Domain Name checking. The default value is TRUE (on).

7.6 Handling Errors

Proxies return errors to an ActiveX client as an HRESULT code and an error message string. The HRESULT identifies the type of error. Each value can be used in conjunction with more than one error message.

The default behavior for VB is to put up an error message based on this information. It also provides a way for you to suppress the error (using `On Error`) and access the error code and message instead. There is no programmatic way to obtain the Progress error message number associated with the message. However, the error number is included in the message string.

If your client is written in C++, you can obtain the error message from the `EXCEPTINFO` structure in the standard manner.

Table 7–6 lists the facility code for each HRESULT.

Table 7–6: HRESULT Values For ActiveX Proxies

HRESULT Value[1]	Description
0x80040200 (-2147220992 decimal)	Used for miscellaneous client-side errors
0x80040201 (-2147220991 decimal)	Error connecting to AppServer
0x80040202 (-2147220990 decimal)	4GL ERROR condition
0x80040203 (-2147220989 decimal)	4GL STOP condition
0x80040204 (-2147220988 decimal)	4GL QUIT condition
0x80040205 (-2147220987 decimal)	AppServer is busy servicing another request
0x80040206 (-2147220986 decimal)	Output result set error, for example, trying to get a column value out of column order or trying to make an illegal type conversion
0x80040207 (-2147220985 decimal)	An unexpected system error that indicates a Progress bug or an unusual environmental problem (such as running out of disk space)

[1] If you display a trapped error number in Visual Basic, it appears in decimal.

7.7 Payment Account Example

This section includes an example of an ActiveX Open Client application written in Visual Basic.

7.7.1 4GL Procedures

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`.

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.
```

Example 7–5: AppServer Application

7.7.2 The ActiveX Client Application Using AccountInfo

The code in Example 7–6 demonstrates how to create and use AppObjects and ProcObjects from a Visual Basic client.

ActiveX Client Using AccountInfo *(1 of 2)*

```
' Create and connect to the AppObject.
Dim acct As AccountLib.CAccount
Dim info As AccountLib.CAcctInfo

Set acct = New CAccount
Call acct.OC_Connect("AppServer://ithaca:3090", "guest", "guest", "")

' Creates a Persistent Procedure instance for account 777
Set info = acct.CreatePO_AcctInfo(777)

' Gets information about payments done after a specific date to
' payees who belong to the payeeList set.
' This example does not show the details of what the payeeList is.
Dim fromDate As Double
Dim payeeList As VBTT.VBTempTable  'NOTE: Implementation not shown.
Dim payees As ProTempTableLib.CProTempTable
Dim payments As ProTempTableLib.CProTempTable
Dim paymentsNum As Long

fromDate = #1/10/98#
Set payeeList = New VBTempTable
Set payees = New CProTempTable
payees.DataSource = payeeList
Set payments = New CProTempTable
' Make the call
Call info.getPaymentsInfo(fromDate, payees, paymentsNum, payments)

MsgBox ("The number of records is: " + Str(paymentsNum))

' Prints all the payment records.
Dim moreRecords As Boolean
Dim columnData As Variant
Dim numCols as Integer

numCols = payments.Fields.Count
moreRecords = payments.MoveNext

While moreRecords
    ColumnData = payments.GetRow
    For i = 0 To numCols - 1
        MsgBox columnData(i)
    Next i
    moreRecords = payments.MoveNext
Wend
```

ActiveX Client Using AccountInfo *(2 of 2)*

```
' Deletes the persistent procedure.
info.OC_Release

' Release the AppObject.  Since this is the last object to share
' the connection this disconnects from the AppServer.
acct.OC_Release
```

Example 7–6: ActiveX Client Using AppServer Objects

8

Special Considerations For Java Applets

The Open Client can support Java applets, by using an HTML page to distribute the appropriate files. However, there are a number of factors you must consider, including:

- The Open Client applet obtains the HTTP Basic authentication and Proxy Web server information from the end user and then passes the information to the Open Client. However, you cannot rely on the browser to provide the user interaction. Your applet must perform this task. You might also have to access to the Proxy Web server configuration information from the browse. In this case, the applet also needs to obtain the Proxy Web server information from the end user.

- An Open Client applets can run on both a company intranet and the Internet. As a result, you need to sign your Java code with a Digital Certificate and your applet must get permission to perform a network connection.

The following sections provide information about:

- Asking For Permission In an Applet
- Signing an Open Client Applet
- Adding Your Applet To an HTML Page

8.1 Asking For Permission In an Applet

After you create and digitally sign your applet distribution files, you must write code that requests permission to establish network connections to servers other than the Web server from which your applet was down loaded. This is necessary because often the Progress servers you are connecting to are on systems other than the ones from which the applet was loaded.

Although the procedure for asking for permission is different for Microsoft Internet Explorer and Netscape browsers, it is possible to support both in the same applet due to Java's ability to dynamically load classes.

The following sections provides information about:

- Asking For Permission In Microsoft Internet Explorer
- Asking For Permission In Netscape Version 4
- Supporting Both Browsers In a Single Applet

8.1.1 Asking For Permission In Microsoft Internet Explorer

Asking for permission in Internet Explorer requires that you add specific code to your applet**.** To add the code, you must:

1 ♦ Down load the security classes for Microsoft by putting the security classes on your CLASSPATH and adding the following import statement to your applet code:

```
import com.ms.security.*;
```

NOTE: The Microsoft security classes are not available as a standalone package. You can get these security classes by downloading the latest Microsoft Java SDK from www.microsoft.com.

2 ♦ Insert the following code into your applet just before creating your Java proxy AppObject:

```
// Assert Net IO privileges
try
{
      if (Class.forName("com.ms.security.PolicyEngine") != null)
          PolicyEngine.assertPermission(PermissionID.NETIO);
}catch(ClassNotFoundException e)
{
      System.out.println("IE Security Manager not found");
}
```

This code searches for the class `com.ms.security.PolicyEngine`. When found, the code calls the static method `assertPermission` asking for NETIO permission. This method verifies that the user selected yes in the security dialog box when the applet was downloaded. If yes was selected, the user can connect to the AppServer.

NOTE: With the Microsoft, a user is prompted only once when the code is first downloaded.

If you use HTTPS, you might also need to obtain file IO privileges from the Microsoft VM security system in order to read the files containing the Root Digital Certificates. If this is the case, add the following line to your code after the line of code that request NETIO privileges:

```
PolicyEngine.assertPermission(PermissionID.FILEIO);
```

Permissions are only valid for a particular method call and that is why you should insert this code right before you construct your Java proxy AppObject. For instance, if you put this code in its own method, you would no longer have the permission when your method returns. For more information on Microsoft's trust based security model, see the Microsoft SDK for Java documentation.

8.1.2 Asking For Permission In Netscape Version 4

Asking for permission in Navigator requires compiling your applet to use permissions. To do this, you must:

1 ♦ Download Netscape’s Object Signing Tools and the Capabilities API Classes from the Netscape Web Site.

2 ♦ Put the Capabilities API classes in your CLASSPATH and add the following import statement in your applet

```
import netscape.security.PrivilegeManager;
```

3 ♦ Insert the following code into your applet just before creating your Java proxy AppObject:

```
try
{
        if (Class.forName("netscape.security.PrivilegeManager") != null)
        {
            try
            {
               PrivilegeManager.enablePrivilege("UniversalConnect");
            }
            catch(Exception ex)
            {
               System.out.println("Netscape threw exception: " +
               ex.toString());
            }
        }
}
catch(ClassNotFoundException e)
{
         System.out.println("Netscape Security Manager not found");
```

When you use the Netscape security model, the user is prompted for the specific permission each time you call the `enablePrivilege` method. If a user gives your applets the requested permission, the Java proxy is instantiated. For more information on the Netscape permission based security model, see the Netscape Web site.

8.1.3 Supporting Both Browsers In a Single Applet

The following code works for both Microsoft Internet Explorer and Navigator in a single applet.

NOTE: To test this code, make sure that both security manager classes are in your CLASSPATH.

```
import com.ms.security.*;
import netscape.security.PrivilegeManager;
try
{
// Assert Net IO privileges
     System.out.println("Checking for Internet Explorer Security Manager");
      if (Class.forName("com.ms.security.PolicyEngine") != null)
     PolicyEngine.assertPermission(PermissionID.NETIO);
}
catch(ClassNotFoundException e)
{
       System.out.println("IE Security Manager not found");
}
try
{
        System.out.println("Checking for Netscape Security Manager");
        if (Class.forName("netscape.security.PrivilegeManager") != null)
        {
           try
           {
                   PrivilegeManager.enablePrivilege("UniversalConnect");
           }
            catch(Exception ex)
            {
                   System.out.println("Netscape threw exception: " +
                   ex.toString());
            }
       }
}
catch(ClassNotFoundException e)
{
        System.out.println("Netscape Security Manager not found");
}
```

8.1.4 Asking For Permission In Netscape Version 6

Asking for permissions in Netscape version 6 or when using a Java plug-in in your Microsoft Internet Explorer requires that you change the Java 2 security policy. The Java 2 security policy file is located in the JRE's `lib\security` subdirectory. It might be combined with a per user policy file located in the user's home directory.

The JRE's file is named `java.policy` and the version in a user's home directory is named `.java.policy`. Changing the JRE policy file grants permissions to every user on the system, while changing the policy file in the user's home directory affects only that user. Progress Software recommends changing the user's home directory on individual machines. For more information about Java 2 security policy files, refer to the Java 2 Security guides.

The Open Client applet requires additional network connection privileges to support HTTP and HTTPS. The Java permission is named `java.net.SocketPermission`, and it takes these arguments:

- The fully qualified DNS address and port (usually port 80) of the Web server for the AppServer Internet Adapter (AIA)
- The network actions `connect,resolve`

The following shows the full entry:

```
permission java.net.SocketPermission host:80, connect,resolve
```

When using HTTPS in your Open Client applet, you might also need an additional Java permission to read the Root Digital Certificate files in order to validate the identity of the Web server for the AIA. If this is the case, you must add file read permissions that provide full read ability. The following shows the entry:

```
permission java.io.FilePermission *, read;
```

The Java permissions should be placed in the Java 2 policy file Grant blocks. One permission grants access to only your Open Client applet code and the other permission grants access to only the Open Client Runtime.

The following shows some sample code for these permission grants:

```
grant codebase http://applicationhost.acme.com/aia/jars-{
permission java.net.SocketPermission
https://aia.acme.com/aia/Aia1:443,connet,resolve;
permission java.io.FilePermission *,read;
};
grant codebase http://applicationhost.acme.com/aia/applets/-{
permission java.net.SocketPermission *, read;
};
```

In this sample:

- The `o4glrthsl.jar` runtime and Root Digital Certificates are located in the applicationhost Web server's `aia/jar` path.
- The Open Client applet is located in the applicationhost Web server's `aia/applets` path.
- The AIA is located in the AIA Web server's `aia/Aia1` path.

8.2 Signing an Open Client Applet

When preparing your Java applet for download with a Digital Certificate, you must sign your code. To sign the code, you perform the following tasks:

1 ♦ Generate your Java proxy. For more information, see Chapter 3, "Generating and Managing Proxies."

2 ♦ Code your applet. For more information, see Chapter 5, "Programming Java Clients."

3 ♦ Create a `.cab` file for Internet Explorer browsers and a `.jar` file for Netscape browsers for the Java applet code and proxy you created that includes a digital signature to download the applet. For instructions about completing these tasks, see the *Microsoft SDK for Java* and the *Netscape Object Signing Tools* documentation.

In order to sign your code, you need a code signing certificate. While both tools allow you to create a test digital certificate, you need a real digital certificate to distribute your application. If you distribute your application over the Internet, you usually get a digital certificate from a public PKI provider such as VeriSign. If you distribute your application on a company intranet, you can set up a Digital Certificate Server and create your own certificates.

The steps for setting up and operating your own Certificate Server are not covered in this document. However, you can get information about this tasks from the Microsoft or Netscape Web sites.

NOTE: Although both Microsoft and Netscape use a standard digital certificate format, the disk storage formats used in the respective signing tools are not interoperable. As a result, you must get separate digital certificates if you want to support both Microsoft and Netscape browsers.

4 ♦ Progress provides the Open Client Runtime packages in digitally signed `.cab` and `.jar` files so that you can distribute these directly without additional work.

These `.cab` and `.jar` files are located in the `Java` directory under your Progress installation directory and have file names that begin with `o4glrt`.

For more information about which of these files to choose for your Java applet see the "Selecting an Open Client Runtime Package" section in Chapter 2, "Configuration and Deployment."

When you embed an Open Client Runtime package in a Web page, a verification dialog box automatically opens when a user accesses the page. The user has the option of accepting or not accepting the installation of the Open Client Runtime. If the user does not accept, the Open Client is not installed and your Open Client applet cannot be run.

8.3 Adding Your Applet To an HTML Page

To add your applet to a Web page, you must add the correct HTML tags to allow a browser to download and run your applet. Although how you do this differs between Microsoft Internet Explorer and Navigator, it is possible to have a single Web page that supports both browsers.

The following sections provide information about:

- Microsoft Internet Explorer
- Navigator
- Supporting Both Internet Explorer and Navigator With the Same Page

8.3.1 Microsoft Internet Explorer

Internet Explorer allows you to download and install Java libraries on the user's computer. Therefore, the Open Client Runtime can be downloaded and installed once. Internet Explorer also supports versioning on these Java libraries. This allows you to update your users automatically when the Open Client Runtime is updated.

To install the Open Client Runtime from a Web page, you add an OBJECT tag to your Web page as shown in the following code:

```
<!-- This checks for
installation of the Open4GL runtime code, and will install it if it isn't
there or is an older version --> <OBJECT
CLASSID="CLSID:E79DA910-15D0-11d2-B591-00C04FD4A860"
        CODEBASE="/cabs/o4glrt.cab#Version=9,1,1115,0">
```

When you do this, you must:

Modify the CODEBASE attribute to the location of the Open Client Runtime packages on the Web server from which the `.cab` file was distributed. When the Internet Explorer encounters this tag, it looks in the user's Windows registry for an entry with the specified class ID and version. Then, it looks to see if the `o4glrt.cab` file exists. If any of the above conditions are not met, the `.cab` file is downloaded.

NOTE: The Microsoft security classes are not available as a standalone package. You can get these security classes by downloading the latest Microsoft Java SDK from `www.microsoft.com`.

Each of the Open Client Runtime `.cab` files has its own unique class ID. This allows your application to change the types of network protocols it uses dynamically the next time the Web page is run. The version number is supplied by Progress in the `o4glrt.cab` file's `open4glrt??.inf` file and changes from release to release.

Once the `.cab` file is downloaded, a dialog box opens advising the user that this code needs to be trusted, and that it is signed by Progress Software Corporation. If the user selects OK, the `.zip` file is installed in their `Windows/Java/TrustedLib` directory, and all required registry keys are added or updated.

The above tag also dynamically adds the .zip file to the TrustedClassPath. This entry is volatile, and if the user restarts Internet Explorer, the .zip file is no longer included in its CLASSPATH. Therefore, any time you have a Web page that uses the Open Client Runtime, you should include that OBJECT tag. Once the code is installed, there is no further action required by the user based on that OBJECT tag, even if they restart Internet Explorer. Since the run time is installed on the TrustedClassPath, only trusted code can access it. Since you need to be trusted to open a connection to a Progress server, you are required to package your applet and proxies into a signed .cab file. Once you have done this, you can add an applet tag like the following example:

```
<applet code=Open4GLDSO.class width=0 height=0 id=datactl MAYSCRIPT=true>
        <PARAM NAME=cabbase VALUE=/cabs/o4gldso.cab> </applet>
```

The code attribute specifies your applet's class name that must be contained in the .cab file specified with the cabbase PARAM. The value of the cabbase PARAM should point to the .cab file's path that contains the applet and proxy on the distribution Web server.

Internet Explorer looks for the cabbase, and if it finds it, downloads the specified .cab file. Since this is a signed .cab file, a dialog box opens and displays information from your code signing certificate. Your Java code is saved in Internet Explorer's volatile cache. So, if the user restarts the Internet Explorer, your .cab file is downloaded again.

8.3.2 Navigator

When adding your applet to a Web page for Navigator, you must consider the following:

- The earlier Navigator browsers did not support downloading and installing Java libraries. Therefore, every time a user accesses the Web page for the first time in a session, the applet, proxy, and the Open Client Runtime packages are downloaded.
- Navigator versions prior to 4.05 did not allow an applet to support multiple .jar files. If you need to support a version of Navigator prior to 4.05, you must repackage everything into a single .jar file and sign it.

For Navigator version 4.05 or later, you can use the following applet tag to embed your applet in a Web page:

```
<applet code="AppletTest.class"
codebase="/cabs" name="AppletTest" width="320" height="240"
archive="apptest.jar,o4glrt.jar">
</applet>
```

In the above example, the applet code and proxies are in the signed .jar file apptest.jar and the Open Client Runtime package is in the .jar file o4glrt.jar. When Navigator encounters this tag, it downloads both .jar files. When your applet requests UniversalConnection privileges, a dialog box opens and prompts your user to grant the privilege.

8.3.3 Supporting Both Internet Explorer and Navigator With the Same Page

It is possible to support both Internet Explorer and Navigator with the same Web page. To do this, you would include code similar to the following:

```
<OBJECT CLASSID="CLSID:E79DA910-15D0-11d2-B591-00C04FD4A860"
        CODEBASE="/cabs/o4glrt.cab#Version=9,0,1066,0"> <applet
code="AppletTest.class"
codebase="/cabs" name="AppletTest" width="320" height="240"
archive="apptest.jar,o4glrt.jar">
<PARAM NAME=cabbase VALUE=/cabs/apptest.cab> </applet>
```

When using the code in this example, you should be aware of the following issues that arise for the difference browsers:

- Navigator ignores the OBJECT tag and the cabbase PARAM because it does not support them.
- Internet Explorer supports the OBJECT tag and the cabbase PARAM and looks for classes in the .cab file specified in the cabbase PARAM of the OBJECT tag before using the .jar files specified in the applet tag's archive attribute.
- Internet Explorer does not support Netscape's code signing technology, as a result, any code found in .jar files is untrusted and limits the servers to which the browser can connect.
- Navigator does not support .cab files, and as a result, ignores the cabbase PARAM. Therefore, you need to specify both .cab and .jar files.

For more information about supporting both Internet Explorer and Navigator with the same Web page, see the Microsoft Knowledge Base article Q179652.

A

Accessing a SmartDataObject API Directly

You can directly access the complete 4GL API (including custom extensions) for any remote Progress 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 SmartDataObjects from an ActiveX Open Client application or if you need to access custom extensions to the default SmartDataObject API that have been provided by a 4GL programmer. If you want to access only the default API generated by the Progress AppBuilder and access it only from a Java Open Client application, use the Java SDOResultSet interface provided by the Open Client Toolkit, instead. For more information on SDOResultSet, see Chapter 6, “Using SmartDataObjects From Java Clients.” The following section describes how to access the SmartDataObject API directly.

A.1 Providing Access To the SmartDataObject API In a Proxy

Before writing an Open Client application that uses the SmartDataObject API, review all of 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 the *Progress ADM 2 Guide*, the *Progress ADM 2 Reference*, and the AppBuilder online help.

A.1.1 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 of 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 must first make the r-code for the corresponding SmartDataObject available to ProxyGen. You can 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 *Building Distributed Applications Using the Progress AppServer*.

You must also ensure that all parameters, internal procedures, and user-defined functions of the ProcObject allow Progress unknown values (?). To do this 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 Chapter 3, "Generating and Managing Proxies."

B

Certificate Management Utility

If your Open Client application uses HTTPS (SSL), you must provide digital certificates with the application. Progress ships a small set of Root Digital Certificates with the Open Client Toolkit.

Most likely, there will be times when you need to manage digital certificates. For example:

- You might want to use a digital certificate from a CA not included with the set from Progress.
- A digital certificate expires, and you need to replace it with a newer version.
- You might want to create your own set of certificates using just those certificates you know your application will use.

The certificate files included with the Open Client Toolkit are stored in `.zip` and `.jar` files called Certificate Store files. For more information about digital certificates and the `.zip` and `.jar` files, see Chapter 2, “Configuration and Deployment.”

Each Certificate Store file holds a number of individual Root Digital Certificates and contains one digital certificate list (`.dcl`file) that lists all the certificate files stored in the file. The individual certificates come in one of the following formats:

- **DER** — These file types have extensions of `.cer` and `.crt`. There is no difference between `.cer` and `.crt` files. One file stores one binary certificate.
- **PEM** —These file types have extensions of `.pem`, `.txt` and `.0`. There is no difference between `.pem`, `.txt` and `.0` files. One file stores one or more certificates.

To help you manage digital certificates, Progress provides the Certificate Management Tool (`procertm`). The `procertm` tool runs on Windows platforms only and lets you import, export, and remove certificates to and from `.jar` and `.zip` files. You can also use this tool to convert digital certificates file formats.

. The following sections provide information about:

- Managing Certificate Store Files
- Converting Digital Certificates

B.1 Managing Certificate Store Files

The `procertm` utility is run from a command line and has the following syntax:

SYNTAX

```
procertm [options] cert_store
```

`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.

`cert`

Path to the Digital Certificate you want to import, export, or remove.

When importing, the path is relative path to the *working directory*. When exporting or removing Digital Certificates from the `cert_store` file, the path is the full Digital Certificate path as specified in the `cert_store` file. Subdirectories should be specified with the forward slash "/".

You can use "*" and "?" wildcards in the file-name and file-extension for the `cert` value.

The options are:

- **-v** — Prints verbose information about the progress of the digital certificates import and export. When used with the -l (Lists) option, 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 of the import, export and remove operations have been completed.
- **-i cert** — Imports certificate file(s) matching cert to the cert_store file from the working directory. The cert_store file is created as required. This option can be specified multiple times.
- **-e cert** — Exports the certificate file(s) matching cert from the cert_store file to the working directory. Any sub-directories are created if required. This option can be specified multiple times.
- **-r cert**— Removes the certificate file(s) matching cert from the cert_store file. You can specify this option multiple times.
- **-d —** Sets the working directory path where certificates are imported from or exported to. The default working directory is the current working directory.

NOTE: The procertm utility supports a limited use of wildcard characters for the cert value in the import, export, and remove options.

You can specify any number of options in any order. 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 the cert_store file. If a certificate is not found, a warning message displays.
2. Exports any certificates specified with the -e option from the 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 the cert_store file. 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.

B.2 Converting Digital Certificates

You can use `procertm` to convert Digital Certificates between the `.DER` and `.PEM` file formats. To convert files from one file format to the other, use the following command line syntax:

SYNTAX

```
procertm -c in_cert out_cert
```

`in_cert`

The Digital Certificate whose file format you want to convert.

`out_cert`

The file format to which you want to convert the digital certificate.

The file utility performs the conversion based on the file-extension type. For example, if `in_cert` has a file extension type of `.crt` and `out_cert` has a file extension type of `.pem`, `in_cert` is converted from the `.der` to `.pem` format and written to the file `out_cert`.

Index

Symbols

? (4GL unknown value). *See* Unknown value (?)

A

absolute( ) method, SDOResultSet 6–19, 6–30

Accessing arrays in output result sets 4–27

Accessing output result sets 4–26

Accessing the AppServer 1–5

ActiveX client programming 7–1
  4GL data type mapping
    See also Data type mapping
    basic 4GL types 7–4
  4GL unknown values 7–5
  connecting to an AppServer 7–4
  data conversions 7–6
  error handling 7–23
  example application 7–25
  ProTempTable object 7–5
  proxies as Automation objects 7–2
  TEMP-TABLE parameters 7–6

ActiveX proxy
  generating 3–16
  generation rules 3–23
  handling errors 7–23
  requirements
    generation 2–8, 6–19
    setting MIDL compiler 2–8
    using in applications 2–12
  validation rules 3–22

Activity log file 3–16

Adding procedure files 3–9

ADO Recordset example 7–8

AIA
  connecting to an AppServer 4–5

AIA path
  invoking a JSE 4–6
  virtual path 4–6

Applets 5–37

AppObject
  components 4–13
  defining in ProxyGen 3–7
  definition 1–5
  examples 4–14, 4–16
    ActiveX 4–16
    Java 4–16
  specifying 4GL procedures and functions 3–8

AppServer
  1–2

connecting directly 4–4
connecting over an intranet 4–3
connecting over the Internet 4–5
connecting through a NameServer 4–3, 4–4
default connection values 4–3
See also Progress AppServer

AppServer connection
with HTTP protocol 4–5
with HTTPS protocol 4–5

appservice-name option
connecting clients 4–6

Array model 4–27
Java TEMP-TABLE parameters 5–16, 5–17, 5–19, 5–22

attachToAppObj ( ) method, SDOResultSet 6–17

Audience 1–xiii

Automation object
data type conversions 7–6

Automation objects
proxy object early binding 7–2
proxy object late binding 7–3
version management 7–3

B

Basic packages
Open Client Runtime 2–2

beforeFirst( ) method, SDOResultSet 6–19

Bold typeface
as typographical convention 1–xv

boolean wasNull() method 6–29

Building
ActiveX client applications 2–12
Java client applications 2–11

Building applications
overview 1–3

C

Call-back sequence for TEMP-TABLE parameters 4–23

_cancelAllRequests() method. *See* Cancelling all requests

cancelBatch( ) method, SDOResultSet 6–27

Cancelling all requests 4–12

cancelRowUpdates( ) method, SDOResultSet 6–24

CField object 7–16

CFields object 7–16

Class Factory method
definition 1–5, 1–7
using in an Open Client 4–11

Client-side errors
client-proxy interactions 4–29
definition 4–29

Close( ) method
CProTempTable object 7–14

close( ) method
ProResultSet interface 5–16

Closing output result sets 4–26

Common methods 4–12

Configuration requirements 2–1

Connecting clients
using appservice-name option 4–6
using HTTPS protocol 4–5
using password option 4–5
using the HTTP protocol 4–5

Connecting directly
to an AppServer 4–4

Connecting to an AppServer
ActiveX 7–4
general programming 4–2

Java 5–2
over an intranet 4–3
using AIA 4–5

Connection methods 4–9
ActiveX 7–4
Java 5–2

Connection parameters
using password option 4–5

Connection states 5–31

Count property
CFields object 7–16

CProTempTable object 7–14

Creating
AppObjects 3–7
ProcObjects 3–8
SubAppObjects 3–7

Customizing access to 4GL procedures 3–13

D

Data type mapping
ActiveX 7–4
4GL unknown values 7–5
TEMP-TABLE OUTPUT parameters 7–17
conversions 7–6
Java 5–2
4GL unknown values 5–4
output parameters 5–4
Progress-SQL ResultSet 5–20
TEMP-TABLE field defaults 5–8
Progress-specific types 7–5

DataSource property
CProTempTable object 7–15

Default values
connecting to an AppServer 4–3

Defining a proxy 3–6

Delegating classes 5–28

deleteRow( ) method, SDOResultSet 6–24

Deleting procedure files 3–9

Deploying
ActiveX client applications 2–14
Java client applications 2–13

Deployment
requirements
ActiveX Controller 2–14
Java application 2–13

detachFromAppObj ( ) method, SDOResultSet 6–17

Distribution packages
Open Client Runtime 2–2

Distribution packages for Open Client 2–3

/DPrefFile execution option 3–28

/DProxyGen.UseMidlFromC++ execution option 3–28

/DWork.Dir execution option 3–27

E

ERROR condition handling 4–30

Error handling 4–29
ActiveX proxies 7–23
Java proxies
5–37
See also Exceptions
ProxyGen validation and generation 3–16
rules 3–21

Error messages
displaying descriptions 1–xxiii

Example application
ActiveX 7–25

Example procedures 1–xx

Exceptions
general 4GL

class hierarchy 5–38
definition 5–37
output result set
definition 5–37
Progress extensions 5–39

F

4GL internal procedures 3–12

4GL procedures 1–6, 3–8, 3–13

4GL unknown value. *See* Unknown value (?)

4GL user-defined functions 3–12

Fields property
CProTempTable object 7–15

findColumn(String columnName) method 6–29

first( ) method, SDOResultSet 6–18

Flat model 4–27
Java TEMP-TABLE parameters 5–16, 5–17, 5–22

G

Generating
ActiveX proxies 2–8
Java proxies
GUI ProxyGen 2–6, 2–7
Open Client proxies 2–6

Generating proxies 3–14

Generation rules 3–23

geQuery( ) method, SDOResultSet 6–30

geRow( ) method, SDOResultSet 6–20

getColumnCount( ) method, SDOResultSetMetaData 6–32

getColumnDisplaySize( ) method, SDOResultSetMetaData 6–33

getColumnJavaTypeName( ) method
ProResultSetMetaData interface 5–21

getColumnLabel( ) method, SDOResultSetMetaData 6–33

getColumnName( ) method, SDOResultSetMetaData 6–32

getColumnProType( ) method
ProResultSetMetaData interface 5–21

getColumnType( ) method, SDOResultSetMetaData 6–33

getColumnTypeName( ) method, SDOResultSetMetaData 6–34

_getConnectionId() method. *See* Getting the connection ID

getDataType( ) method
ProResultSet interface 5–17

getFetchSize( ) method, SDOParameters 6–11

getFetchSize() method 6–29

getFieldCount( ) method
ProResultSetMetaData interface 5–22

getFieldExtent( ) method
ProResultSetMetaData interface 5–22

getFieldJavaTypeName( ) method
ProResultSetMetaData interface 5–22

getFieldName( ) method
ProResultSetMetaData interface 5–22

getFieldProType( ) method
ProResultSetMetaData interface 5–22

getFields( ) method, SDOModificationException 6–36

getFieldTypeName( ) method
ProResultSetMetaData interface 5–22

getObject ( ) method
ProResultSet interface 5–18

getObject( ) method
InputResultSet class 5–11
ProResultSet interface 5–16, 5–17

getPrefetchMaxRows( ) method, SDOParameters 6–12

_getProcReturnString() method. *See* Getting the procedure return value

GetRow( ) method
CProTempTable object 7–15

getRowIdentity( ) method, SDOParameters 6–10

getRowIdentity( ) method, SDOResultSet 6–30

getScrollingMode( ) method, SDOParameters 6–12

getSDOInterface( ) method, SDOResultSet 6–31

getStateless( ) method, SDOParameters 6–11

getTable( ) method, SDOModificationException 6–37

getTableName( ) method, SDOResultSetMetaData 6–33

getText( ) method, SDOModificationException 6–37

Getting started with ProxyGen 3–2

Getting the connection ID 4–12

Getting the procedure return value 4–12

GetValue( ) method
IClientTempTable interface 7–8

H

Help
Progress messages 1–xxiii

Hierarchy of proxy objects 1–7

Holder classes 5–4
super class definition 5–4
type-specific
definitions 5–6
mappings 5–5

HRESULT values 7–23

HTTP
Open Client Runtime packages 2–2

HTTP protocol
connecting an AppServer 4–5
using password option 4–5
using with clients 4–5

HTTPS 4–5

HTTPS lite
Open Client Runtime packages 2–2

HTTPS protocol
connecting an AppServer 4–5

HTTPS standard
Open Client Runtime packages 2–2

I

IClientTempTable example 7–10

IClientTempTable interface 7–7

IDLE state
connection 5–31

Implementation classes 5–28

inBatch( ) method, SDOResultSet 6–27

Input result sets 4–25

INPUT-OUTPUT parameters

Java clients 5–4

InputResultSet class
definition 5–7
using 5–10

insertRow( ) method, SDOResultSet 6–24

Internal procedures
mapping in ProxyGen 3–12

Internet
connecting an AppServer 4–5

isAfterLast( ) method, SDOResultSet 6–19

isAttached( ) method, SDOResultSet 6–17

isBeforeFirst( ) method, SDOResultSet 6–19

isFirst( ) method, SDOResultSet 6–20

isLast( ) method, SDOResultSet 6–20

isNullable( ) method, SDOResultSetMetaData 6–33

isReadOnly( ) method, SDOResultSetMetaData 6–33

_isStreaming() method. *See* TEMP-TABLE parameters, OUTPUT

isWritable( ) method, SDOResultSetMetaData 6–33

Italic typeface
as typographical convention 1–xv

Item( ) method
CFields object 7–16

J

Java applications, building with SmartDataObjects 6–7

Java client programming 5–1
4GL data type mapping 5–2
See also Data type mapping
applets 5–37
connecting to an AppServer 5–2
Exception handling 5–37
extending proxy objects 5–28
4GL unknown values 5–4
multi-threaded applications 5–31
TEMP-TABLE parameters 5–7

Java proxy
generating 3–16
generation rules 3–23
handling errors 5–37
requirements
generation on Windows 2–6, 2–7
using in applications 2–11

JSE
virtual path for AIA 4–6
virtual paths 4–6

K

Keystrokes 1–xv

L

last( ) method, SDOResultSet 6–19

Log file for ProxyGen 3–16

M

Manual
syntax notation 1–xvi

Manual, organization of 1–xiv

Messages
displaying descriptions 1–xxiii

Metadata for Java output ResultSets 5–19

Methods
Class Factory 4–11
Common 4–12

Connection 4–9
customizing for ProcObjects 3–12
customizing in ProxyGen 3–11
defining in ProxyGen 3–10
passing parameters 4–22
Remote 4GL 4–9

MIDL compiler
setting 2–8
setting environment 2–9

Monospaced typeface
as typographical convention 1–xv

MoveNext( ) method
CProTempTable object 7–15
IClientTempTable interface 7–8

moveToCurrentRow( ) method,
SDOResultSet 6–24

moveToInsertRow( ) method,
SDOResultSet 6–23

Multi-threaded applications 5–31
_cancelAllRequests( ) method 5–32
access to ResultSets 5–32
connection states 5–31
request states 5–31

N

Name property
CField object 7–17

NameServer
connecting to an AppServer 4–3
connecting to an AppServer. 4–4

Naming conventions
automatic conversions 3–20
proxy object 3–18
restrictions and side effects 3–21

next ( ) method
SDOResultSet 6–18

next( ) method
InputResultSet class 5–11
ProResultSet interface 5–16
SDOResultSet 6–18

nextMessage( ) method,
SDOModificationException 6–37

Non-persistent procedures
adding and deleting in ProxyGen 3–9
definition 1–6

O

Object model 1–5

OC_CancelAllRequests() method. *See*
Cancelling all requests

OC_GetConnectionId() method. *See*
Getting the connection ID

OC_GetProcReturnString() method. *See*
Getting the procedure return value

OC_IsStreaming() method. *See*
TEMP-TABLE parameters, OUTPUT

OC_Release() method. *See* Releasing proxy
objects

Open Client
applications
building and running 1–3
requirements 2–11
AppServer connections 4–2
architecture 1–2
deployment requirements
ActiveX 2–14
Java 2–13
Java applets 5–37
object model 1–5
programming 4–2
supported applications 2–3
supported protocols 2–3
types 1–3

Open Client application
specifying an AppServer connection 4–2

Open Client proxies

generating 2–6

Open Client Runtime
definition 1–3
distribution packages 2–2

Open Client Runtime package
supported security 2–4, 2–5

Open Client Runtime packages
basic packages 2–2
HTTP 2–2
HTTPS lite 2–2
HTTPS standard 2–2
list of packages 2–3

Open4GLException class 5–38

OUTPUT parameters
Java clients 5–4

Output result sets 4–25

P

Parameter passing in Open Clients 4–22

Persistent procedures
adding and deleting in ProxyGen 3–9
definition 1–6

ppObject
SmartDataObject access in Java 6–5

Preferences
ProxyGen 3–15
specifying a file to save 3–28

PRO_* data type constants 5–21

Procedure files
adding and deleting 3–9
methods 3–11

Procedures
examples of 1–xx

ProcObject
components 4–19
customizing 3–11
defining in ProxyGen 3–8
examples 4–20, 4–21
Java 4–20

Programming Open Clients
See also ActiveX client programming, Java client programming
accessing a SmartDataObject API A–1
AppServer connections 4–2
error handling 4–29
error handling. *See* Error handling
method types 4–8
overview 4–2
passing parameters 4–22
proxy run-time properties 4–28
using proxy objects 4–13

Progress AppServer
accessing with Open Clients 1–5
connections from Open Clients 4–2
managing Open Client proxies 3–24
Open Client architecture 1–2

Progress SmartDataObjects
direct API access A–1

Progress unknown value. *See* Unknown value (?)

Project
file
definition 3–5
saving 3–14
proxy directory 3–15
work directory 3–14

PROPATH
AppObjects and SubAppObjects 3–8

ProResultSet interface
definition 5–7
extension methods 5–17
getObject ( ) method 5–18
standard methods 5–16
using 5–16

ProResultSetMetaData interface 5–19
extension methods 5–21, 5–22
standard ResultSetMetaData methods 5–20

ProSQLException class 5–39

ProTempTable object 7–5

Proxy
  ActiveX HRESULT values 7–23
  adding and deleting procedures 3–9
  Automation object access 7–2
    early binding 7–2
    late binding 7–3
  definition 1–7
  directory 3–15
  generation and validation 3–14
    rules 3–21
  Java Exceptions 5–37
  management for AppServer 3–24
  methods 4–8
  naming conventions 3–18
  objects 1–5
    extending in Java 5–28
    hierarchy 1–7
    Java multi-threaded applications 5–31
    releasing 1–7
    types 1–8
    using 4–13
  parameter passing 4–22
  run-time properties 4–28

Proxy Generator (ProxyGen) 1–3
  activity log 3–16
  creating a ProcObject 3–8
  creating a SubAppObject 3–7
  creating an AppObject 3–7
  execution options
    Batch on UNIX 3–28
    Batch on Windows Platforms 3–28
    GUI 3–26
  generating and validating proxies 3–14
  main window 3–4
  overview 3–2
  preferences 3–15
    specifying file to save 3–28
  proxies
    defining 3–6
  saving the project file 3–14
  starting
    batch 3–6
    GUI 3–3
  system requirements 2–6

ProxyGen
  definition 1–3

.pxg. *See* Project

Q

QUEUED state
  request 5–31

QUIT condition handling 4–30

R

R-code
  definition 1–5

refreshRow( ) method, SDOResultSet 6–28

relative( ) method, SDOResultSet 6–19

_release() method. *See* Releasing proxy objects

releaseSDOInterface() method 6–32

Releasing objects 1–7

Releasing proxy objects 4–13
  can close output result sets 4–26

Remote 4GL methods 4–9

reOpenQuery( ) method, SDOResultSet 6–14, 6–17, 6–30

reOpenQuery() method 6–30

Request states 5–31

Requirements 2–1
  See also Configuration requirements

Result set parameters 4–22
  See also TEMP-TABLE parameters
  accessing arrays 4–27
  accessing schemas 4–27
  input 4–25
  output 4–25

Root digital certificate
validating Web servers 4–7

rowDeleted( ) method, SDOResultSet 6–25

rowInserted( ) method, SDOResultSet 6–24

rowUpdated( ) method, SDOResultSet 6–25

Rules
proxy generation 3–23
proxy validation 3–22

Running applications
overview 1–3

RUNNING state
connection 5–31
request 5–31

Run-time properties 4–28
ActiveX client access 7–20
Java client access 5–33

S

Scrolling modes, SDOResultSet 6–12

SDOAppObject
connecting to AppServer 6–7
definition 6–5

SDOModificationException class 6–36

SDOParameters
definition 6–5
passing 6–10

SDOResultSet 6–28, 6–29, 6–30, 6–32
batch mode 6–26
comparing with JDBC 6–38
creating 6–8
definition 6–5
detaching 6–17
typical scenario 6–18
getting column values 6–20
meta data 6–32
nonstandard methods 6–34
standard JDBC 2 methods 6–32
methhods and operations 6–16
miscellaneous management methods 6–28
multi-threading 6–35
navigating rows 6–18
scrolling modes 6–12
SmartDataObject management methods 6–29
stateless mode 6–13
updating 6–21
columns 6–22
rows 6–23
visibility 6–27

SDOResultSetMetaData interface 6–32

SDORresultSetMetaData getMetaData() method 6–28

SDOScrollingMode
FORWARD_ONLY 6–13
KEEP_ROWS 6–13
PREFETCH 6–12

sendBatch 6–26

sendBatch( ) method, SDOResultSet 6–17

sendBatchAndReOpen( ) method, SDOResultSet 6–17

sendBatchAndReopen( ) method, SDOResultSet 6–26

Server-side errors
client-server interactions 4–30
definition 4–29

setFetchSize( ) method, SDOParameters 6–10

setPrefetchMaxRows( ) method
SDOParameters 6–12
SDOResultSet 6–14

setRowIdentity( ) mehtod, SDOParameters 6–10

setScrollingMode( ) method, SDOParameters 6–11

setStateless( ) method, SDOParameters

6–11

SmartDataObject
building Java applications 6–7
definition 6–2
direct API access A–1
Java access 6–2
coding 6–7
example 6–4
methods supported 6–29
objects required 6–5
requirements 6–3
sample application 6–38
tools and documentation 6–4
mapping from ProcObjects A–2
techniques for accessing 1–9

Specify an AppServer connection
Open Client application 4–2

startBatch( ) method, SDOResultSet 6–26

Starting
Batch ProxyGen 3–6
GUI ProxyGen 3–3

Stateless mode,SDOResultSet 6–13

STOP condition handling 4–30

Streaming
connection state 5–31
definition 4–26
request state 5–31

STREAMING state
connection 5–31
request 5–31

SubAppObject
components 4–17
defining in ProxyGen 3–7
definition 1–5
examples 4–17, 4–19
ActiveX 4–19, 4–21
Java 4–17
specifying 4GL procedures and functions 3–8

Supported applications
Open Client 2–3

Supported protocols
Open Client 2–3

Supported security
Open Client packages 2–4, 2–5

Syntax notation 1–xvi

System requirements
building applications 2–11
deployment 2–13
ProxyGen 2–6

T

TEMP-TABLE parameters 4–22
accessing array fields 4–27
ActiveX clients 7–6
ADO Recordsets 7–7
choosing object type for input 7–7
DAO Recordsets 7–7
example 7–19
IClientTempTable interface 7–7
INPUT 4–23
ActiveX 7–7
Java 5–10
INPUT-OUTPUT
Java 5–15
Java clients 5–7
accessing on output 5–16
default field data type mappings 5–8
metadata on output 5–19
Progress type constants 5–21
supported data type conversions 5–9
OUTPUT
ActiveX 7–14
checking if open 4–12
closing 4–26
Java 5–15

Thread control 4–28
ActiveX clients 7–20
Java clients 5–33

Tracing 4–28
ActiveX clients 7–20
Java clients 5–33

Transactions, SDOResultSet 6–22
  concurrency control 6–22
  extended 6–25

Type property
  CField object 7–17

Types of Open Clients 1–3

Types of proxy objects 1–8

Typographical conventions 1–xv

U

Unknown value (?)
  ActiveX client programming 7–5
  example for managing 3–25
  Java client programming 5–4
  proxy validation rules 3–22
  setting for procedure file methods and ProcObjects 3–11, 3–12
  setting for proxy object 3–8

updateRow( ) method, SDOResultSet 6–24

Updating SDOResultSet objects 6–21
  batch mode 6–26
  column values 6–22
  rows 6–23
  visibility 6–27

use-id option
  using with HTTP protocol 4–5

-usemidlfromc++ execution option 3–28

User-defined functions
  mapping in ProxyGen 3–12

user-id option
  using with connection parameter 4–5

username option
  connecting clients 4–5

V

Validating
  Web server digital certificates 4–7

Validating proxies 3–14

Validation rules 3–22

Virtual path
  AIA path 4–6

Virtual paths
  for JSE 4–6

void close() method 6–28

W

Web server digital certificates
  validating 4–7

Web servers
  root digital certificates 4–7

Work directory
  project 3–14