OpenEdge® Development: .NET Open Clients
# Table of Contents

**Preface**

Purpose..............................................................................................................................................11

Audience............................................................................................................................................11

Organization.....................................................................................................................................12

Using ABL documentation..................................................................................................................12

References to ABL compiler and run-time features.................................................................13

References to ABL data types........................................................................................................13

Typographical conventions.............................................................................................................13

Examples of syntax descriptions.............................................................................................14

Long syntax descriptions split across lines............................................................................16

Complex syntax descriptions with both required and optional elements...........................16

Example procedures......................................................................................................................17

OpenEdge messages......................................................................................................................18

Obtaining more information about OpenEdge messages.....................................................18

**Chapter 1: Configuring and Deploying .NET Open Client Applications**

Selecting a .NET Open Client Runtime package............................................................................21

Preparing to generate proxies for a .NET client using ProxyGen or Batch ProxyGen...................23

Building a .NET Open Client application that uses a .NET proxy.........................................24

Deploying a .NET Open Client application using a .NET proxy..............................................25

Sample .NET client applications...............................................................................................25

**Chapter 2: Proxy Objects and Methods**

Proxy objects...................................................................................................................................28

Namespace for proxy objects....................................................................................................28

Proxy methods...............................................................................................................................29

Connection methods................................................................................................................29

Remote ABL methods..................................................................................................................29

Class factory methods................................................................................................................30

Common properties......................................................................................................................31

Common methods..........................................................................................................................33

Running methods on session-free AppObjects.........................................................................34

Session-free threading model.....................................................................................................34

Sharing of the connection pool between session-free AppObjects.........................................34

Sample proxy..................................................................................................................................35

Sample AppObject......................................................................................................................35

Sample SubAppObject................................................................................................................36
Chapter 3: Connecting to an AppServer

Connection class
Establishing the connection
Handling connection exceptions
Returning a user defined string to the client from the AppServer connection procedure
Supported AppServer modes
Releasing a session-free Connection object

Chapter 4: Passing Parameters

ABL data type mappings
Date and time conversions
Arrays as parameters
Passing arguments by reference
Unknown value (?) as a parameter
ABL temp-table to ADO.NET DataTable mappings
Mapping temp-table components
Mapping temp-table component attributes
Resolving differences between ABL and .NET date and time implementations
Mapping the XML-NODE-TYPE attribute to ADO.NET
ABL ProDataSet to ADO.NET DataSet mappings
Mapping ProDataSet components
Mapping ProDataSet component attributes
Passing TABLE and TABLE-HANDLE parameters
Static and dynamic temp-tables
Data type mapping for temp-table fields
Unknown value (?)
Temp-table array fields
Indexes for temp-tables
Passing a TABLE or TABLE-HANDLE as an INPUT parameter
Providing ABL extensions for INPUT TABLE-HANDLE parameters
Passing a TABLE or TABLE-HANDLE as an INPUT-OUTPUT parameter
Passing a TABLE or TABLE-HANDLE as an OUTPUT parameter
Temp-table examples
Passing DATASET and DATASET-HANDLE parameters
Static and dynamic ProDataSets
Data-type mapping for ProDataSets
Passing a DATASET or DATASET-HANDLE as an INPUT parameter
Providing ABL extensions for INPUT DATASET-HANDLE parameters
Passing a DATASET or DATASET-HANDLE as an INPUT-OUTPUT parameter
Passing a DATASET or DATASET-HANDLE as an OUTPUT parameter
Updating a DataSet
Chapter 5: Accessing Proxy Properties

Application configuration file .................................................................94
RunTimeProperties and Connection classes.................................................94
  RunTimeProperties class...........................................................................94
  Connection class.....................................................................................95
Accessing properties with methods.............................................................95
  General property accessor methods for the RunTimeProperties class........95
  General property accessor methods for Connection objects....................96
Available properties..................................................................................97
Accessing properties using purposed accessor methods............................99
Accessing properties directly.....................................................................100
  Session properties................................................................................100
  Thread control property........................................................................102
Proxy server specification properties..........................................................102
Secure Sockets Layer management properties............................................103
Tracing property (RunTimeProperties only)..................................................103
Relationship between RunTimeProperties and Connection objects...............105
Modifying an instantiated Connection object's properties.........................106
Alphabetical listing of properties.................................................................108
PROGRESS.Session.AppServerKeepalive.....................................................108
PROGRESS.Session.CompressionEnabled...................................................109
PROGRESS.Session.CompressionLevel......................................................109
PROGRESS.Session.CompressionThreshold...............................................110
PROGRESS.Session.ConnectionLifetime....................................................110
PROGRESS.Session.ConnectionTimeout.....................................................110
PROGRESS.Session.EnableTracing............................................................111
PROGRESS.Session.IdleConnectionTimeout...............................................111
PROGRESS.Session.InitialConnections......................................................112
PROGRESS.Session.LogFileName..............................................................112
PROGRESS.Session.LoggingLevel.............................................................112
PROGRESS.Session.MaxConnections.........................................................113
PROGRESS.Session.MinConnections........................................................113
Chapter 6: Handling Errors.................................................................123
   General ABL exceptions...............................................................123
      Exception class hierarchy.......................................................124
      Exception descriptions..........................................................124

Chapter 7: Using OpenEdge .NET Proxy Objects in Microsoft Visual Studio .NET.................................................................127
   Types of .NET open clients..........................................................127
      Windows forms (GUI)..............................................................128
      Web forms (ASP.NET).............................................................128
      .NET Web services (ASP Web services)....................................128
   Getting started building a .NET Open Client application using Microsoft Visual Studio .NET..................128
   Using the samples.......................................................................129

Chapter 8: Using the Open Client .NET OpenAPI to Directly Access the AppServer..........................................................133
   Overview of .NET OpenAPI classes..........................................133
   Connecting to the AppServer.....................................................134
   Running procedures and user-defined functions.......................136
      Running a non-persistent procedure on an OpenAppObject......136
      Running a persistent procedure (OpenProcObject) on an OpenAppObject...........................................137
      Running a single-run or singleton procedure (OpenProcObject) on an OpenAppObject.........................137
Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages

**Purpose**

The Open Client Toolkit (a component of OpenEdge® Studio) exposes AppServer™ functionality to Open Clients (non-ABL clients). As discussed in *OpenEdge Development: Open Client Introduction and Programming*, the ProxyGen tool allows you to generate proxy objects for .NET (and other) Open Clients. These proxy objects encapsulate remote 4GL procedures and functions supported on an AppServer. Your .NET Open Client application can then access these AppServer procedures and functions through methods of the generated proxy objects.

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

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

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

Organization

Configuring and Deploying .NET Open Client Applications on page 21
Describes configuration prerequisites specific to .NET Open Client development and deployment.

Proxy Objects and Methods on page 27
Presents the objects and methods generated by ProxyGen for a .NET Open Client.

Connecting to an AppServer on page 39
Describes how to connect to an AppServer, by instantiating an AppObject and, optionally, a connection object.

Passing Parameters on page 45
Discusses mappings for basic data types, static and dynamic temp-tables, and static and dynamic ProDataSets (ABL ProDataSets), when data is passed between a .NET application and OpenEdge.

Accessing Proxy Properties on page 93
Describes the different ways you can access the proxy properties that govern behavior across an entire application.

Handling Errors on page 123
Describes error handling in .NET Open Client applications.

Using OpenEdge .NET Proxy Objects in Microsoft Visual Studio .NET on page 127
Lists the types of .NET open clients and gives an example of getting started building a .NET Open Client application using Microsoft Visual Studio .NET.

Using the Open Client .NET OpenAPI to Directly Access the AppServer on page 133
Describes an API for accessing application services on the AppServer from a .NET client without the need to generate Open Client proxies using ProxyGen.

Using ABL documentation

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

For the latest documentation updates see the OpenEdge Product Documentation Overview page on Progress Communities:
References to ABL compiler and run-time features

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

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

References to ABL data types

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

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

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

Typographical conventions

This documentation uses the following typographical and syntax conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td>Convention</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a <em>simultaneous</em> key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a <em>sequential</em> key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>

**Syntax:**

<table>
<thead>
<tr>
<th>Fixed width</th>
<th>A fixed-width font is used in syntax, code examples, system output, and file names.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-width italics</td>
<td>Fixed-width italics indicate variables in syntax.</td>
</tr>
<tr>
<td>Fixed-width bold</td>
<td>Fixed-width bold italic indicates variables in syntax with special emphasis.</td>
</tr>
<tr>
<td>UPPERCASE fixed width</td>
<td>ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.</td>
</tr>
<tr>
<td>Period (.) or colon (:)</td>
<td>All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>[]</td>
<td>Small brackets are part of ABL.</td>
</tr>
<tr>
<td>{ }</td>
<td>Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.</td>
</tr>
<tr>
<td>{}</td>
<td>Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate repetition: you can choose one or more of the preceding items.</td>
</tr>
</tbody>
</table>

**Examples of syntax descriptions**

In this example, **ACCUM** is a keyword, and **aggregate** and **expression** are variables:
Syntax

**ACCUM aggregate expression**

*FOR* is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

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

**Syntax**

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

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

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

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

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

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

**Syntax**

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

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:
Syntax

\[
\text{MAXIMUM ( expression, expression } [ \text{, expression } ] \ldots)
\]

In this example, you must specify MESSAGE and at least one expression or SKIP \([ ( n ) ]\), and any number of additional expression or SKIP \([ ( n ) ]\) is allowed:

Syntax

\[
\text{MESSAGE \{ expression | SKIP [ ( n ) ] \} \ldots}
\]

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

Syntax

\[
\{ \text{include-file}
\quad [ \text{argument | &argument-name = "argument-value"} ] \ldots \}
\]

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

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

Syntax

\[
\text{WITH [ ACCUM max-length ] [ expression DOWN ]}
\quad [ \text{CENTERED } [ n \text{ COLUMNS } ] [ \text{SIDE-LABELS } ]
\quad [ \text{STREAM-IO } ]
\]

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more field entries or one record. Options available with field or record are grouped with braces and brackets:
Syntax

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

Example procedures

OpenEdge documentation may provide example code that illustrates syntax and concepts. You can access many of the example files, and details for installing them, from the following locations:

- A self-extracting Documentation and Samples file available on the OpenEdge download page of the Progress Software Download Center
- The OpenEdge Product Documentation Overview page on Progress Communities:


Once installed, you can locate the example files in the following paths under the OpenEdge Documentation and Samples installation directory:

<table>
<thead>
<tr>
<th>This directory . . .</th>
<th>Contains examples for the following documents . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>src\prodoc\dotnetobjects</td>
<td>OpenEdge Development: GUI for .NET Programming</td>
</tr>
<tr>
<td>src\prodoc\dynamics</td>
<td>The Progress Dynamics documentation</td>
</tr>
<tr>
<td>src\prodoc\getstartoop</td>
<td>OpenEdge Development: Object-oriented Programming</td>
</tr>
<tr>
<td>src\prodoc\handbook</td>
<td>OpenEdge Getting Started: ABL Essentials</td>
</tr>
<tr>
<td>src\prodoc\interfaces</td>
<td>OpenEdge Development: Programming Interfaces</td>
</tr>
<tr>
<td>src\prodoc\json</td>
<td>OpenEdge Development: Working with JSON</td>
</tr>
<tr>
<td>src\prodoc\langref</td>
<td>OpenEdge Development: ABL Reference</td>
</tr>
<tr>
<td>src\prodoc\prodatasets</td>
<td>OpenEdge Development: ProDataSets</td>
</tr>
<tr>
<td>src\prodoc\tranman</td>
<td>OpenEdge Development: Translation Manager</td>
</tr>
<tr>
<td>src\prodoc\visualdesigner</td>
<td>OpenEdge Getting Started: Introducing Progress Developer Studio for OpenEdge Visual Designer</td>
</tr>
<tr>
<td>src\prodoc\xml</td>
<td>OpenEdge Development: Working with XML</td>
</tr>
<tr>
<td>src\samples\open4gl\java</td>
<td>OpenEdge Development: Java Open Client</td>
</tr>
</tbody>
</table>
OpenEdge messages

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

• **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

• **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before 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 OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge 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 after execution messages.

• Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

• Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.

• Terminates the current session.

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

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

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

• Choose **Help > Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.

• Choose **Help > Messages** and then type the message number to display a description of a specific OpenEdge message.

• In the Procedure Editor, press the **HELP** key or F1.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the `pro` command to obtain a message description by message number:
1. Start the Procedure Editor:

```
OpenEdge-install-dir/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.
Configuring and Deploying .NET Open Client Applications

This chapter describes configuration prerequisites specific to .NET Open Client development and deployment.

For details, see the following topics:

• Selecting a .NET Open Client Runtime package
• Preparing to generate proxies for a .NET client using ProxyGen or Batch ProxyGen
• Building a .NET Open Client application that uses a .NET proxy
• Deploying a .NET Open Client application using a .NET proxy
• Sample .NET client applications

Selecting a .NET Open Client Runtime package

.NET Open Clients use the .NET Open Client Runtime. This supports the AppServer, AppServerDC, AppServerS, and AppServerDCS and HTTP and HTTPS connection protocols on an intranet or internet. The assemblies that make up the .NET Open Client Runtime are:

• Progress.o4glrt.dll
• Progress.ssl.dll
• Progress.Messages.dll
Based on the run-time configuration you want to support, you have the option of selecting one of the following Open Client Runtime distribution packages:

- **Strong-named and digitally-signed runtime package** — Supports tightly secure .NET applications. If you use this set of Open Client Runtime assemblies, you must use these exact same assemblies for generating proxies both for building your .NET client application and for deployment. For OpenEdge product updates, you must regenerate your proxies using the new (updated) assemblies, rebuild your application using the updated proxies, and then deploy the updated proxy and application along with the new OpenEdge assemblies.

  This configuration gives you the ability to strong-name the proxy and the .NET application. You can also digitally sign the proxy and .NET application. This configuration also allows you to load the OpenEdge .NET assemblies into the Global Assembly Cache.

  This set of assemblies is located in:

  ```
  OpenEdge-install-directory\dotnet\deploy\strongnamed-signed
  ```

- **Digitally-signed only runtime package** — Supports .NET applications where security is not a high concern. You can use these assemblies to generate proxies and build and deploy your application. For OpenEdge product updates, you do not need to regenerate your proxies or rebuild your application. You can simply give your users the updated OpenEdge assemblies.

  This configuration does not give you the ability to strong-name the proxy and the .NET application. As a result, this configuration prevents you from loading the OpenEdge .NET assemblies, the proxy, and the .NET application into the Global Assembly Cache. However, you can digitally sign the proxy and .NET application.

  This set of assemblies is located in:

  ```
  OpenEdge-install-directory\dotnet\deploy\signed
  ```

- **Strong-named only runtime package** — Supports a less common type of assembly required by some applications. The assemblies are strong-named but are not digitally signed. A strong-named assembly will allow the Windows system to determine the assembly's data-integrity (i.e., has it been altered since it was compiled and linked). But the strong-name cannot be used as a proof of authenticity because it lacks the digital signature that can be used to identify the author of the assembly.

  If you use this set of Open Client Runtime assemblies, you must use these exact same assemblies for generating proxies both for building your .NET client application and for deployment. For OpenEdge product updates, you must regenerate your proxies using the new (updated) assemblies, rebuild your application using the updated proxies, and then redeploy the updated proxy and application along with the new OpenEdge assemblies.

  This configuration gives you the ability to strong-name both the proxy and the .NET application. You can also have the OpenEdge .NET assemblies loaded into the .NET Global Assembly Cache. However, you cannot digitally sign the proxy or the .NET application.

  This set of assemblies is located in:

  ```
  OpenEdge-install-directory\dotnet\deploy\strongnamed
  ```
In ProxyGen, in the .NET client settings of the Generate Proxy dialog box, you can specify what distribution package to use. The selected assemblies are copied to the ProxyGen output directory (which is the location of the generated proxies).

**Note:** Only the message resources for the current locale are copied to the ProxyGen output directory. To support additional languages, you must copy the files yourself as described in Supporting localized messages for .NET client interfaces on page 23.

The classes in the .NET Open Client Runtime are written in C#, but can be accessed by any .NET language.

**Supporting localized messages for .NET client interfaces**

When ProxyGen builds the .NET proxy, it copies the required .NET runtime assemblies to the proxy generation output directory. For the message assembly, only the resources for the language matching the current local language are copied to the output directory. If you need to support a different language, you must manually copy some additional files to the output directory.

To support an alternative language or more than one language, you must copy the language resources from one of the following locations depending on your generation option:

- **Strong-named and digitally-signed runtime assemblies** — Copy the resource for the languages you want to support from the following directory:
  
  \`OpenEdge-install-directory\dotnet\deploy\strongnamed-signed\`

- **Digitally-signed only runtime assemblies** — Copy the resource for the languages you want to support from the following directory:
  
  \`OpenEdge-install-directory\dotnet\deploy\signed\`

- **Strong-named only runtime assemblies** — Copy the resource for the languages you want to support from the following directory:
  
  \`OpenEdge-install-directory\dotnet\deploy\strongnamed\`

For example, if you want to support a French deployment for a signed-only environment, copy the fr directory from \`OpenEdge-install-directory\dotnet\deploy\signed\` to your proxy generation output directory.

**Preparing to generate proxies for a .NET client using ProxyGen or Batch ProxyGen**

To configure your environment to generate a .NET proxy:

Before generating a proxy, you must configure your environment.
1. Install Progress® OpenEdge® Studio, OpenEdge Development Server, Progress Developer Studio for OpenEdge, or 4GL Development in Windows where you plan to run ProxyGen.

**Note:** If you install on networked storage on your local intranet, you must adjust your .NET security settings to fully trust your local intranet storage device. (Microsoft's default is to not fully trust these storage devices.) For more information, see Microsoft's documentation on security settings.

2. Install Microsoft Visual Studio .NET or Microsoft .NET SDK Version 3.0 or later. Follow the instructions on the Microsoft Web site to complete the installation.

3. Make sure the ABL r-code for the proxy is accessible to your system, and the directory structure for the r-code is the same as the Propath relative path on the AppServer.

4. Choose one:
   - Run ProxyGen from the icon.
   - Run Batch ProxyGen by executing the `bproxygen` command with an existing project file.

**Note:** For more information, see the chapter on generating proxies and Web service definitions in OpenEdge Development: Open Client Introduction and Programming.

---

**Building a .NET Open Client application that uses a .NET proxy**

Several steps are involved in building an Open Client application that uses a .NET proxy.

To build a client application that uses a .NET proxy:

1. Install Microsoft Visual Studio .NET.

2. To build the client application on a machine other than the proxy machine, you must:
   a) Copy the proxy assembly (.dll file) to a directory on the system.
   b) Copy the OpenEdge .NET Open Client Runtime assemblies (`Progress.o4glrt.dll`, `Progress.ssl.dll`, and `Progress.Messages.dll`) and the message resource files from the ProxyGen output directory to a directory on the system or, if the assemblies are strong-named, you can install them into the .NET Global Assembly Cache. For information on strong-named assemblies, see Selecting a .NET Open Client Runtime package on page 21.

3. Create your client application as a project using Microsoft Visual Studio .NET and ensure that the compiler version specified in the proxy is the same compiler version used to develop the client application.

4. Add to your client application project a reference to the proxy assembly (.dll file).

5. If you selected the Delay Sign option in the .NET Client Details group of the ProxyGen Generate Proxies dialog box, you must complete the signing process for the proxy using the private/public key pair. For details, see the Microsoft Visual Studio .NET Documentation on the Strong Name Tool (`sn.exe`).

6. To your client application project, add a reference to the main Open Client Runtime assembly, `Progress.o4glrt.dll`. See Step 2 for the location.

7. Write, compile, and execute the client application. (This is the focus of the remainder of this book.)
Deploying a .NET Open Client application using a .NET proxy

To deploy a client application that uses a .NET proxy, you must first perform certain steps on the system where you plan to run the application.

To deploy a client application that uses a .NET proxy:

1. Install the Microsoft .NET Redistributable Framework Version 1.1 or later.
2. Copy the required files for the client application and proxy:
   a) Copy the client application.
   b) Copy the proxy assembly (.dll file) to the directory required by the client application.
   c) Copy the OpenEdge .NET Open Client Runtime assemblies (Progress.o4glrt.dll, Progress.ssl.dll, and Progress.Messages.dll) and the message resource files from the ProxyGen output directory to the directory required by the client application or, if the assemblies are strong-named, you can install them into the .NET Global Assembly Cache.
   d) Run your .NET Open Client application as designed.

Note: Alternately, you can follow Microsoft's .NET deployment model. For information, see the Microsoft Visual Studio help.

Sample .NET client applications

Sample .NET client applications are located in the self-extracting Documentation and Samples file available on the OpenEdge download page of the Progress Software Download Center. For more information on accessing this directory, see the section on example procedures in the . This samples directory contains separate subdirectories for different sets of samples and support files. Each sample subdirectory contains a readme.txt file with instructions on building and running the sample.

The samples in the subdirectory are written in C# and VB.NET. This book references some of these samples, but also contains many other code samples and examples. The examples throughout this book are shown in C# unless noted otherwise.
Using the Open Client architecture, .NET Open Clients access AppServer™ business logic through proxy code. The application developer identifies business logic to define the Open Client interface, using ProxyGen. Once the interface is defined, the application developer generates the .NET client proxy, also using ProxyGen. The client programmer writes a .NET client application in any .NET language. The client application accesses AppServer functionality through methods on the generated .NET proxy objects.

The generated .NET proxy code uses the .NET Open Client Runtime to communicate with the AppServer. The .NET Open Client Runtime converts parameters and return values between ABL (Advanced Business Language) and .NET data types as needed. The .NET proxies can be generated only on Microsoft operating systems.

For details on the Open Client architecture, see *OpenEdge Development: Open Client Introduction and Programming*.

The .NET proxies generated by ProxyGen contain one or more classes, packaged as a .NET assembly. This chapter describes the .NET proxy classes and methods generated by ProxyGen, as detailed in the following sections:

* Proxy objects on page 28
* Proxy methods on page 29
* Running methods on session-free AppObjects on page 34

For details, see the following topics:

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

For a .NET Open Client proxy, ProxyGen generates one class for each Open Client object, each unique static temp-table, and each unique static ProDataSet, as shown in the following table:

<table>
<thead>
<tr>
<th>For each . . .</th>
<th>ProxyGen generates . . .</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppObject</td>
<td>A .NET class</td>
<td>public class Account</td>
</tr>
<tr>
<td>SubAppObject</td>
<td>A .NET class</td>
<td>public class Tax</td>
</tr>
<tr>
<td>ProcObject</td>
<td>A .NET class whose name matches the name of the persistent, single-run, or singleton procedure</td>
<td>public class AccountInfo</td>
</tr>
<tr>
<td>Static temp-table</td>
<td>A strongly typed ADO.NET DataTable class</td>
<td>public class CustomerTableDataTable : Progress.Open4GL.ProDataTable</td>
</tr>
<tr>
<td>Static ProDataSet (ABL DataSet)</td>
<td>A strongly typed ADO.NET DataSet class</td>
<td>public class PurchaseOrderDataSet : Progress.Open4GL.ProDataSet</td>
</tr>
</tbody>
</table>

ProxyGen creates one assembly, *AppObject.dll*, which contains the above Open Client classes and strongly typed ADO.NET DataTable and ADO.NET DataSet classes (for example, *Account.dll*).

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

All these files are placed in the output directory specified in the general settings of the ProxyGen *Generate Proxies* dialog box. In addition, the assemblies for the selected Open Client Runtime package are copied to the output directory.

Namespace for proxy objects

All .NET Open Client proxy objects can be defined within a namespace specified using the *General* field of the *Namespaces* group in the .NET *Client Details* of the *Generate Proxies* dialog box. A namespace is an organizational unit for classes or types. All Open Client proxy objects are defined in the specified namespace. If no namespace is specified, the objects are defined in the global namespace. Progress Software Corporation recommends that you specify a namespace, to help ensure the uniqueness of your proxy objects.

For example, if the namespace specified in ProxyGen is *Acme*, and an AppObject named *Account* is defined, ProxyGen generates the following .NET class:

```csharp
using Acme;

public class Account
```

Chapter 2: Proxy Objects and Methods

OpenEdge Development: .NET Open Clients
The namespace for strongly typed proxy classes for ADO.NET DataTables and DataSets is, by default, defined by appending `StrongTypesNS` to the specified `General` namespace. For example, if a static temp-table named `OrderTT` is defined in a procedure added to the `Account` AppObject, ProxyGen generates the following .NET class for the temp-table:

```csharp
Acme.StrongTypesNS.OrderTT
```

Note that the `StrongTypesNS` namespace is prepended with the `Acme` namespace specified using the `General` field, resulting in the namespace `Acme.StrongTypesNS`.

You can also define your own namespace for strongly typed DataTable and DataSet classes using the `DataSet` field of the `Namespaces` group in .NET Client Details. By deselecting the `Use Default` option, you can enter your own value in `DataSet` that ProxyGen appends to the specified `General` namespace.

### Proxy methods

The proxies you generate include several types of methods and properties:

- **Connection methods** on page 29
- **Remote ABL methods** on page 29
- **Class factory methods** on page 30
- **Common properties** on page 31
- **Common methods** on page 33

### Connection methods

The constructor of the AppObject is used to establish a connection to an AppServer. For details, see Connecting to an AppServer on page 39.

### Remote ABL methods

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

A sample non-persistent procedure follows:

```abl
AddCustomer.p:
DEFINE INPUT PARAMETER name AS CHARACTER.
DEFINE INPUT PARAMETER phone AS CHARACTER.
DEFINE INPUT PARAMETER email AS CHARACTER.
DEFINE OUTPUT PARAMETER CustomerNumber AS INTEGER.
...
```
ProxyGen generates the following .NET proxy method:

```csharp
public void AddCustomer(string name, string phone, string email,
                      out int CustomerNumber)
```

### Passing parameters

ProxyGen maps ABL data types to equivalent data types in .NET for ABL INPUT, OUTPUT, and INPUT-OUTPUT parameters. For details, see Passing Parameters on page 45.

### Handling return values

When using ProxyGen, you optionally can specify whether the ABL RETURN-VALUE should be added to the proxy method, for each non-persistent procedure and internal procedure. (User-defined functions always return a value.) If you specify this, the method returns `string`; otherwise, the method returns `void`. If specified for the non-persistent previous procedure, ProxyGen generates the following .NET proxy method:

```csharp
public string AddCustomer(string name, string phone, string email,
                      out int CustomerNumber)
```

Also, if you did not specify that the ABL RETURN-VALUE should be returned, the client can access the current value of the ABL RETURN-VALUE function by calling the `_GetProcReturnString` common method on the Open Client object. See Common methods on page 33 for additional information.

### Class factory methods

ProxyGen generates two class factory methods.

#### SubAppObject

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

```csharp
public SubAppObject
CreateAO_SubAppObject( )
```

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

```csharp
public Tax CreateAO_Tax( )
```
**ProcObject**

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

```csharp
public ProcObject CreatePO_ProcObject()
```

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

```csharp
public AccountInfo CreatePO_AccountInfo()
```

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

### Common properties

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

#### Connection ID

The `ConnectionId` property is most often used to identify entries in AppServer log files. This is a read-only property.

For session-managed applications, the `ConnectionId` property returns a string containing a unique identifier for this connection.

For session-free applications, the `ConnectionId` property returns a string containing a unique identifier for the connection most recently used by the current thread. For example:

```csharp
public string ConnectionId
```

The `ConnectionId` property returns null if the Application Service is not connected.

#### Procedure return string

This is a read-only property.

For session-managed applications, the following property retrieves the return value of the method most recently run on any thread.
For session-free applications, the following property retrieves the return value of the method most recently run on the current thread:

```java
public String ProcReturnString
```

This property can also contain a return string from the AppServer connection procedure immediately after connecting, if the connection procedure returns a string.

The return value cannot be retrieved by another thread.

For more information on handling return values, see *OpenEdge Development: Open Client Introduction and Programming*.

**Procedure type**

The `ProcedureType` property identifies the type of procedure that a `ProcObject` encapsulates as persistent, single-run, or singleton. The property holds a .NET enum type with a value of `Persistent`, `SingleRun`, or `Singleton`:

**Syntax**

```java
public ProcedureType ProcedureType
```

**Request ID**

This is a read-only property.

For session-managed applications, the `RequestId` property returns a unique string identifying the request most recently run on any thread.

For session-free applications, the `RequestId` property returns a unique string identifying the request most recently run on the current thread. For example:

```java
public String RequestId
```

The `RequestId` cannot be retrieved by another thread.

The `RequestId` property returns null if the Application Service is not connected.

**SSL subject name**

This is a read-only property.
The SSLSubjectName property provides the SSL server's subject name that is obtained from its validated digital certificate. For example:

```java
public String SSLSubjectName
```

## Common methods

The Open Client interface provides common methods which provide information about the current state of .NET proxy AppObjects, SubAppObjects, and ProcObjects, with respect to their connection to the application service. This section lists these methods and describes the information they provide.

### Cancel all requests

The following methods raise a STOP condition in the context of each outstanding request on the AppServer:

```java
public void CancelAllRequests()
public void _CancelAllRequests()
```

**Note:** The _CancelAllRequests() method is not CLS-compliant; it is supported only for backward compatibility. Progress Software recommends that you use CancelAllRequests(), instead.

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

The CancelAllRequests() method is defined to throw a Progress.Open4GL.Exceptions.Open4GLEexception.

A client can call the CancelAllRequests() method to do the following:

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

### Dispose

The following method makes the object unavailable for further use:

```java
public void Dispose()
```

When you execute the Dispose() method on the last available proxy object that shared a particular connection, the client disconnects from the AppServer. In addition, if you execute this method on a ProcObject, the associated persistent procedure on the AppServer is deleted. The connection to the AppServer is automatically released for single-run and singleton procedures after their internal procedures or user-defined functions are completed, and in the case of a single-run procedure, the procedure itself is also automatically deleted from the AppServer. However, the Dispose() method does clean up client-side resources for single-run and singleton procedures.
Running methods on session-free AppObjects

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

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

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

Session-free threading model

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

- When the application constrains the size of the connection pool, requests can become blocked if connections are unavailable. This is accomplished by the Open Client Runtime blocking the thread on which the request is run.
- Certain request-specific methods must be executed on the same thread on which the request was run. See Common methods on page 33 for more details.

Sharing of the connection pool between session-free AppObjects

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

The Connection object’s reference to the connection pool will be maintained until the ReleaseConnection() method is called on the Connection object. This reference might affect the life cycle of the connection pool. That is, the Connection object may be used to sustain the existence of the connection pool beyond the lifetime of the AppObject (and its associated SubAppObjects and ProcObjects). It is your responsibility to call ReleaseConnection() on the Connection object to remove the reference to the connection pool.

Note that a single Connection object can be used to instantiate instances of different AppObject classes, provided that the Connection object refers to an AppServer that provides the necessary application services. This is possible since a single AppServer can serve multiple application services. However, no validation of this is provided.
The Connection object does not maintain a reference to a session-managed AppObject, regardless of how it is constructed. If a session-managed AppObject is constructed using a Connection object that contains a reference to a connection pool, an exception is thrown. In other words, Connection objects cannot be shared between session-free and session-managed AppObjects.

Sample proxy

This section presents examples of .NET proxy objects and their methods. All .NET proxies are generated using C#.

Sample AppObject

The methods in this section comprise the definition of a sample .NET AppObject, Account, as shown:

- The AppObject constructors that connect to an Application Service:

  ```csharp
  public Account(Connection connectObj)
  public Account(string urlString, string userId, string password,  
  string appServerInfo)
  public Account(string userId, string password, string appServerInfo)
  public Account( )
  ```

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

  ```csharp
  public Tax CreateAO_Tax( )
  ```

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

  ```csharp
  public AccountInfo CreatePO_AccountInfo(int accountNum)
  ```

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

```csharp
public AccountInfo CreatePO_AccountInfo( )
```

See Sample ProcObject on page 36 for an example of how the methods for the internal procedures and user-defined functions of might look if AccountInfo.p were called as a single-run or singleton procedure.
Remote ABL methods to run the non-persistent procedures Add (defined by Add.p) and Remove (defined by Remove.p) on the AppServer:

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

Common properties in all .NET proxy objects, excluding the ProcedureType property:

```java
public string ConnectionId
public string ProcReturnString
public string RequestId
public string SSLSubjectName
```

Common methods in all .NET proxy objects:

```java
public void CancelAllRequests()
public void Dispose()
```

Sample SubAppObject

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

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

```java
public TaxInfo CreatePO_TaxInfo(int accountNum)
```

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

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

Common methods in all .NET proxy objects.

Sample ProcObject

The following methods comprise the definition of a sample .NET ProcObject, AccountInfo:
Remote ABL methods to run the internal procedures `getPaymentsInfo`, `setDirectDeposit`, and `getDirectDeposit` (defined in `AccountInfo.p`) on the AppServer:

```java
public void getPaymentsInfo(System.DateTime fromDate,
    System.Data.DataTable payeeList,
    int paymentsNum,
    System.Data.DataTable paymentsInfo)

public void setDirectDeposit(System.Data.DataTable ddData)

public void getDirectDeposit(System.Data.DataTable ddData)
```

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

```java
public void getPaymentsInfo(int accountNum,
    System.DateTime fromDate,
    System.Data.DataTable payeeList,
    int paymentsNum,
    System.Data.DataTable paymentsInfo)

public void setDirectDeposit(int accountNum,
    System.Data.DataTable ddData)

public void getDirectDeposit(int accountNum,
    System.Data.DataTable ddData)
```

- The `ProcedureType` property
- Common methods in all .NET proxy objects.
Connecting to an AppServer

This chapter details the mechanisms and steps for connecting to an AppServer from your application.

For details, see the following topics:

• Connection class
• Establishing the connection

Connection class

OpenEdge provides a connection object, Progress.Open4GL.Proxy.Connection. This connection object provides a means to store AppServer connection information, which can be passed to the AppObject constructor when connecting to the AppServer.

The connection object has the two constructors shown in the syntax boxes that follow.

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

Syntax

```csharp
public Connection(string url, string userid, string password,
                   string appserverInfo)
```
Note: For an HTTPS connection using the AppServer Internet Adapter (AIA) with the Web server running on the .NET Open Client machine, you must specify the Internet host name (not "localhost") in the URL exactly as it appears in the CN field of the Web server certificate.

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

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

**Syntax**

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

```java
public Connection(string userid, string password, string appserverInfo)
```

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

Before you establish a connection, you can set properties for the `Connection` object. See **Accessing Proxy Properties** on page 93 for more information on setting and updating `Connection` object properties.

### Establishing the connection

To establish a connection to an AppServer, you must instantiate an `AppObject`. You can create an `AppObject` using one of the four available constructors.

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

```java
public AppObject(Progress.Open4GL.Proxy.Connection connectObj)
```

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

Syntax

```csharp
public AppObject(string url, string userid, string password,
string appserver-info)
```

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

**Note:** For an HTTPS connection using the AppServer Internet Adapter (AIA) with the Web server running on the .NET Open Client machine, you must specify the Internet host name (not "localhost") in the `url` parameter exactly as it appears in the CN field of the Web server certificate.

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

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

Syntax

```csharp
public AppObject(string userid, string password, string appserver-info)
```

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

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

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

Syntax

```csharp
public AppObject()
```

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

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

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

**Handling connection exceptions**

The creation of a connection to an AppServer should be wrapped with exception-handling code to avoid memory leaks and unnecessary allocation of client and server resources. Also, information received from a caught exception can be valuable in tracing and fixing connection errors.
The following is an example of handling connection exceptions:

```csharp
try
{
    //Create Customer AppObject to connect
    Customer appObj = new Customer(ConnectObj);
}
catch (Progress.Open4GL.Exceptions.ConnectException ex)
{
    MessageBox.Show(ex.ToString());
}finally
{
    if (appObj != null)
    {
        appObj.Dispose();
    }
}
```

This example executes the connect in the `try` block. If there is an exception thrown by the process, the `catch` block displays the exception message to the user.

The `catch` block is specifically catching the `ConnectException`. For more information on .NET Open Client exceptions, see Handling Errors on page 123.

The `finally` block ensures that the `Dispose()` method is called on the `AppObject`. By putting the `Dispose()` method in the `finally` block, you ensure that the connection is closed whether an exception occurs or the connect is successful.

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

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

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

If the connection is successful, and the connection procedure has a `RETURN` statement, you can access the string using the existing `AppObject` public property `ProcReturnString`. If the connect procedure does not return a value, then the `ProcReturnString` property will be null.
The following sample .NET Open Client code illustrates this functionality:

```csharp
try
{
    //Create Customer AppObject to connect
    Customer appObj = new Customer(ConnectObj);
    MessageBox.Show(appObj.ProcReturnString);
}
catch (Progress.Open4GL.Exceptions.Open4GLException ex)
{
    // if there is a application defined return string, then display it
    if (ex.ProcReturnString != string.Empty)
    {
        MessageBox.Show(ex.ProcReturnString);
    }
    else
    {
        // display Progress defined error message
        MessageBox.Show(ex.ToString());
        // or display new generic message
        MessageBox.Show("Connection failed");
    }
}
finally
{
    if (appObj != null)
    {
        appObj.Dispose();
    }
}
```

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

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

### Supported AppServer modes

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

To access a session-free AppServer, you must set the proxy property `PROGRESS.Session.SessionModel` to 1. For more information, see *Accessing Proxy Properties* on page 93.

### Releasing a session-free Connection object

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

```java
public void ReleaseConnection()
```

**Note:** This method is not necessary for and has no effect in session-managed applications.
Passing Parameters

This chapter describes the data type mappings for parameter passing as well as the process for passing and mapping temp-tables and ProDataSets.

For details, see the following topics:

- ABL data type mappings
- ABL temp-table to ADO.NET DataTable mappings
- ABL ProDataSet to ADO.NET DataSet mappings
- Passing TABLE and TABLE-HANDLE parameters
- Passing DATASET and DATASET-HANDLE parameters
- ProDataTable class methods
- ProDataSet class methods

ABL data type mappings

.NET data types are defined as part of the common type system within .NET. The common type system supports two general categories of types, value types and reference types. Many ABL data types are mapped to built-in value types provided by .NET. Reference types are classes, either those provided by .NET or user-defined classes. Some ABL data types are mapped to .NET-provided classes, and a few are mapped to ABL-provided classes.
An ABL parameter is mapped directly to its corresponding .NET data type or OpenEdge-provided class, as shown in the following table. These mappings also apply to user-defined function return types, although LONGCHAR and MEMPTR types cannot be returned across the Open Client interface.
<table>
<thead>
<tr>
<th>ABL</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>System.String</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>Progress.Open4GL.COMHandle</td>
</tr>
<tr>
<td>DATASET</td>
<td>Strongly typed DataSet</td>
</tr>
<tr>
<td></td>
<td>See Passing DATASET and DATASET-HANDLE parameters on page 73.</td>
</tr>
<tr>
<td>DATASET-HANDLE</td>
<td>System.Data.DataSet</td>
</tr>
<tr>
<td></td>
<td>See Passing DATASET and DATASET-HANDLE parameters on page 73.</td>
</tr>
<tr>
<td>DATE</td>
<td>System.DateTime</td>
</tr>
<tr>
<td>DATETIME</td>
<td>System.DateTime</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>System.DateTime</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>System.Decimal</td>
</tr>
<tr>
<td>HANDLE</td>
<td>Progress.Open4GL.Handle</td>
</tr>
<tr>
<td>INT64</td>
<td>System.Int64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>System.Int32</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>System.Boolean</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>System.String</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>Progress.Open4GL.Memptr</td>
</tr>
<tr>
<td>RAW</td>
<td>System.Byte[ ]</td>
</tr>
<tr>
<td>RECID</td>
<td>System.Int64</td>
</tr>
</tbody>
</table>
The COM-HANDLE, HANDLE, RECID, ROWID, and WIDGET-HANDLE data types are not meaningful outside the ABL environment. Their use is restricted to obtaining the value from an ABL procedure and passing it back to another ABL procedure.

The mapping for the individual fields of a temp-table or temp-tables within a ProDataSet differs from that of other parameters. For information on data type mapping for fields within these parameters, see Data type mapping for temp-table fields on page 67.

.NET Open Clients do not support the ABL BUFFER parameter.

### Date and time conversions

The following table details the conversions for date and time between the AppServer and the .NET Open Client.

#### Table 2: .NET Open Client DATE, DATETIME, and DATETIME-TZ conversions

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td><strong>Input:</strong> Extract Date portion&lt;br&gt;<strong>Output:</strong> Time is set to midnight</td>
</tr>
<tr>
<td>DATETIME</td>
<td><strong>Input and Output:</strong> Date and Time preserved</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td><strong>Input:</strong> Use time zone of the .NET Client&lt;br&gt;<strong>Output:</strong> Convert to .NET Client time zone</td>
</tr>
</tbody>
</table>

### Arrays as parameters

ABL allows arrays (called EXTENTS in ABL) to be passed as run-time parameters, using the same data type mapping shown in the following table:
Table 3: ABL to .NET data-type mapping for array parameters

<table>
<thead>
<tr>
<th>ABL EXTENT data type</th>
<th>Proxy code (C#) return type for values</th>
<th>Proxy code (C#) return type for unknown values</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>System.Int32[]</td>
<td>IntHolder[]</td>
</tr>
<tr>
<td>INT64</td>
<td>System.Int64[]</td>
<td>LongHolder[]</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>System.Decimal[]</td>
<td>DecimalHolder[]</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>System.String[]</td>
<td></td>
</tr>
<tr>
<td>LOGICAL</td>
<td>System.Boolean[]</td>
<td>BooleanHolder[]</td>
</tr>
<tr>
<td>DATE</td>
<td>System.DateTime[]</td>
<td>DateHolder[]</td>
</tr>
<tr>
<td>DATETIME</td>
<td>System.DateTime[]</td>
<td>DateHolder[]</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>System.DateTime[]</td>
<td>DateHolder[]</td>
</tr>
<tr>
<td>RECID</td>
<td>System.Int64[]</td>
<td></td>
</tr>
<tr>
<td>RAW</td>
<td>System.Byte[][]</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>Progress.Open4GL.Rowid[]</td>
<td></td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>Progress.Open4GL.COMHandle[]</td>
<td></td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>Progress.Open4GL.WIDGET-HANDLE[]</td>
<td></td>
</tr>
</tbody>
</table>

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

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

In .NET, parameters can be passed by value or by reference. Most .NET languages provide a way to declare a parameter’s mode. For example, in C#, parameters are passed by value by default, and you can use ref and out to specify passing the parameter by reference. In Visual Basic .NET, use ByVal and ByRef to identify the parameter mode.

INPUT-OUTPUT parameters

An INPUT-OUTPUT parameter is mapped to its .NET data type, with the parameter mode set based on the .NET language. For example, in C# you would use the ref parameter modifier. The ref modifier requires that the variable be initialized before being passed to a method.

OUTPUT parameters

An OUTPUT parameter is mapped to its .NET data type, with the parameter mode set based on the .NET language. For example, in C# you would use the out parameter modifier. The out modifier does not require that the variable be initialized before being passed to a method.

Note: If the variable already contains a value, it is replaced for INPUT-OUTPUT and OUTPUT parameters.

For more information about parameter passing in .NET, see the documentation for your .NET programming language.

Unknown value (?) as a parameter

The Unknown value (?) is mapped to a null value for a .NET reference type (class); however, the .NET value types referenced by a .NET proxy (System.Boolean, System.Decimal, System.Int32, System.Int64, and System.DateTime) always have a default value, and the null value does not belong to the set of values they support. For each of these value types, OpenEdge provides two options for representing a null value:

• A .NET nullable value type, which is a structure that can be set to a null value as well as to one of the values of a corresponding .NET value type.

• An OpenEdge holder class, which is defined in the Progress.Open4GL namespace, and can be set to a null value as well as to one of the values of a corresponding .NET value type. This option is provided for backward compatibility in ProxyGen projects created using previous OpenEdge releases that do not support .NET nullable types.

You can select the option you want to represent the Unknown value (?) for parameters and user-defined function return values in ProxyGen using the Unknown Support options in the .NET Client Details of the Generate Proxies dialog box.

Note: For ProxyGen projects created in OpenEdge releases without support for nullable types (releases prior to OpenEdge 11.0), the project defaults to using holder classes to represent the Unknown value (?). For ProxyGen projects created in OpenEdge releases that support nullable types, the project defaults to using nullable types to represent the Unknown value (?).
Using .NET nullable value types

In ProxyGen, if you specify that a supported value type parameter or return value can have the Unknown value (?), and you have chosen the option to use nullable value types, the method is defined with the corresponding nullable type; otherwise the parameter is defined using the .NET value type.

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

```csharp
foo(int i, string s, bool b)
```

The following is an example of a C# proxy method that allows all the parameter values to have Unknown value (?), using nullable value types where necessary:

```csharp
foo(System.Int32? i, string s, System.Boolean? b)
```

.NET nullable types are represented using the following syntax:

**Syntax**

```
ValueType?
```

Where `ValueType` is a built-in .NET value type that has a corresponding nullable type, and `?` identifies the value type as nullable.

A nullable type is actually a generic structure type and the `ValueType?` syntax is shorthand for the generic structure `System.Nullable<valueType>`. These forms are interchangeable. However, the .NET Open Client uses the `ValueType?` form to represent nullable types in a proxy.

The following table shows the mapping between ABL primitive types and the corresponding .NET nullable value types.
### Table 4: ABL primitive type to .NET nullable value type mappings

<table>
<thead>
<tr>
<th>This ABL primitive type...</th>
<th>Maps to this .NET nullable type...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>System.Datetime?</td>
</tr>
<tr>
<td>DATETIME</td>
<td></td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>System.Decimal?</td>
</tr>
<tr>
<td>INT64</td>
<td>System.Int64?</td>
</tr>
<tr>
<td>INTEGER</td>
<td>System.Int32?</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>System.Boolean?</td>
</tr>
<tr>
<td>RECID</td>
<td>System.Int64?</td>
</tr>
</tbody>
</table>

To set the value of an object with a nullable value type (nullable type object), you can directly assign it a value as if you were setting an object with the corresponding value type, and you can also assign it directly to `null`. When reading the value of a nullable type object, you need to know if the object is `null` (has no value) in order to avoid throwing an exception when reading its value, which is undefined when set to `null`. Also, to read the value of a nullable type object, you must access a property of the object that contains the value. Therefore, the `System.Nullable<ValueType>` generic structure supports the following read-only properties for nullable type objects:

- **HasValue** — A property of type `System.Boolean` that is `true` if the object has a value and is `false` otherwise (the object is set to `null`)

- **Value** — A property of type `ValueType` that returns the object's value if `HasValue` is `true`, and throws a `System.InvalidOperationException` exception if `HasValue` is `false`

You can also directly test a nullable type object for `null` using the `==` and `!=` operators, and directly access the value of a non-`null` nullable type object.
For example, within a C# method, you can use nullable type objects as follows:

```csharp
System.Int32? intNullable; // Nullable value type object
System.Int32 intValue; // Corresponding value type object

intNullable = 10; // Set a nullable object to a value
intValue = intNullable.Value; // Get the value of the nullable object
intNullable = null; // Set the nullable object to null
...

if (intNullable.HasValue) // Test if nullable object is null before reading
    intValue = intNullable.Value; // The object has a value; read its property
else
    intValue = 0; // The object is set to null; assigning 0
...

if (intNullable != null) // Directly test if nullable object is null
    intValue = intNullable; // The object has a value; read directly
else
    intValue = 0; // The object is set to null; assigning 0
```

For more information on the characteristics of nullable value types, including additional methods and C# operators that they support, see the Microsoft .NET documentation on using nullable types in C#.

**Setting an INPUT or INPUT-OUTPUT parameter to the Unknown value (?)**

For **INPUT** or **INPUT-OUTPUT** parameters, you can set an input value to the **Unknown value (?)** by directly setting the nullable type object to null, as shown in the previous example.

**Testing an INPUT-OUTPUT or OUTPUT parameter for the Unknown value (?)**

For **OUTPUT** and **INPUT-OUTPUT** parameters, you can find out whether an output **Unknown value (?)** is returned by testing if the output object's `HasValue` property is `false` or if the object equals `null`, shown in the previous example.

**Using OpenEdge holder classes**

In ProxyGen, if you specify that a supported value type parameter or return value can have the **Unknown value (?)**, and you have chosen to use holder classes, the method is defined with the corresponding holder class; otherwise the parameter is defined using the .NET value type.

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

```csharp
foo(int i, string s, bool b)
```

The following is an example of a C# proxy method that allows all the parameter values to have **Unknown value (?)**, using holder classes where necessary:

```csharp
foo(IntHolder i, string s, BooleanHolder b)
```
For each ABL data type that maps to a .NET value type, there is a `Holder` class. These `Holder` classes belong to the `Progress.Open4GL` namespace and extend the `Progress.Open4GL.Holder` class, as shown in the following table.

Table 5: Holder class definitions

<table>
<thead>
<tr>
<th>Class</th>
<th>Constructors</th>
<th>Property</th>
</tr>
</thead>
</table>
| BooleanHolder    | BooleanHolder( )
                  | BooleanHolder(bool value)                          | System.Boolean
                  | BooleanValue;                                     |
| DateHolder       | DateHolder( )
                  | DateHolder(System.DateTime value)                  | System.DateTime
                  | DateValue;                                        |
| DecimalHolder    | DecimalHolder(decimal value)
                  | DecimalHolder(double value)                        | System.Decimal
                  | DecimalValue;                                     |
| IntHolder        | IntHolder( )
                  | IntHolder(int value)                               | System.Int32
                  | intValue;                                         |
| LongHolder       | LongHolder( )
                  | LongHolder(long value)                             | System.Int64
                  | LongValue;                                        |

The property specified in the above table allows you to set and get a value with the .NET value type for which it is defined. These classes all contain two additional properties that allow you to set and test for null:

- **Value** — To which you can set or get null, which corresponds to setting or getting the `Unknown value (?)` for an ABL primitive type.
- **isNull** — A `System.Boolean` property that returns true if the `Value` property of the holder class is set to null.

The following table shows ABL primitive types that have a default .NET value-type mapping that cannot accept `Unknown value (?)`. To accept the `Unknown value (?)`, these ABL primitive types are mapped to holder classes in the `Progress.Open4GL` namespace, as shown in the table.
Table 6: ABL primitive type to holder class mappings

<table>
<thead>
<tr>
<th>This ABL primitive type...</th>
<th>Maps to this holder class...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>Progress.Open4GL.DateHolder</td>
</tr>
<tr>
<td>DATETIME</td>
<td></td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Progress.Open4GL.DecimalHolder</td>
</tr>
<tr>
<td>INT64</td>
<td>Progress.Open4GL.LongHolder</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Progress.Open4GL.IntHolder</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Progress.Open4GL.BooleanHolder</td>
</tr>
<tr>
<td>RECID</td>
<td>Progress.Open4GL.LongHolder</td>
</tr>
</tbody>
</table>

Setting an INPUT or INPUT-OUTPUT parameter to the Unknown value (?)

For INPUT or INPUT-OUTPUT parameters, you can set an input value to the Unknown value (?) by setting the Value property on the holder object to null.

So, in C#, to set the value of a holder object (Holder) to the Unknown value (?), use the Value property, as shown:

**Syntax**

```
Holder.Value=null
```

Testing an INPUT-OUTPUT or OUTPUT parameter for the Unknown value (?)

For OUTPUT and INPUT-OUTPUT parameters, you can find out whether an output Unknown value (?) is returned by testing if the IsNull property is true or the Value property on the holder object equals null.

For example, in C#, to test the value of a holder object (Holder), check the isNULL or Value property, as shown:
Syntax

```java
if (Holder.isNull) | if (Holder.Value == null)
```

ABL temp-table to ADO.NET DataTable mappings

The ABL temp-table maps closely to an ADO.NET DataTable. The temp-table can be passed in a remote ABL procedure or user-defined function, between .NET and the AppServer, as a parameter or as part of a ProDataSet parameter (see ABL temp-table to ADO.NET DataTable mappings on page 56). For static temp-table parameters, ProxyGen creates extended ADO.NET DataTable classes for each unique temp-table definition. For dynamic temp-tables, ProxyGen uses the standard ADO.NET DataTable class for the parameters.

In ABL, you can define a temp-table as being LIKE a database table, and you can add fields of any type to this definition, or you can define the table as a set of fields independent of any particular database table. You can also define indexes to manage large sets of data efficiently.

An ADO.NET DataTable shares all of these characteristics. Thus, OpenEdge supports:

- The creation of a temp-table on the AppServer
- Passing that temp-table to a .NET client application
- Allowing that application to manipulate the data using an ADO.NET DataTable
- Returning data changes to the AppServer to update any data source for the temp-table

To support the mapping of ABL temp-tables to ADO.NET DataTables, OpenEdge establishes detailed correspondences between temp-table components and similar ADO.NET objects. This includes the mapping of such elements as temp-table field data types and temp-table component attributes. For more information on the mapping of temp-table field data types, see Passing TABLE and TABLE-HANDLE parameters on page 66. The following sections describe the mapping of temp-table components and their attributes:

- Mapping temp-table components on page 56
- Mapping temp-table component attributes on page 57
- Resolving differences between ABL and .NET date and time implementations on page 62
- Mapping the XML-NODE-TYPE attribute to ADO.NET on page 62

Mapping temp-table components

The following table shows how a given ABL temp-table and its related components map to its companion objects in ADO.NET.
Table 7: ABL temp-table component to ADO.NET object mapping

<table>
<thead>
<tr>
<th>ABL component</th>
<th>.NET object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer / Temp-Table</td>
<td>System.Data.DataTable</td>
</tr>
<tr>
<td></td>
<td>System.Data.DataRow</td>
</tr>
<tr>
<td></td>
<td>or strongly typed DataTable</td>
</tr>
<tr>
<td>Buffer Field</td>
<td>System.Data.DataColumn</td>
</tr>
<tr>
<td>Primary Key</td>
<td>System.Data.UniqueConstraint</td>
</tr>
<tr>
<td>Unique non-primary index</td>
<td>System.Data.UniqueConstraint</td>
</tr>
</tbody>
</table>

Mapping temp-table component attributes

In addition to mapping temp-table components, OpenEdge maps relevant attributes of temp-table components for use in a .NET Open Client. These ABL attributes are similar to .NET properties and some of them map to actual ADO.NET properties.

Attributes on temp-table and related ABL components are similar to object properties. They allow you to query and modify information for these components in ABL. Many of these attributes are not meaningful in the .NET environment and require no mapping by OpenEdge. Among those attributes that are meaningful, some correspond to equivalent ADO.NET properties that already maintain the same information for equivalent .NET objects. Still others require some form of mapping by OpenEdge in order to maintain the corresponding information in ADO.NET.

For those ABL attributes that require mapping, OpenEdge maps some of them to existing ADO.NET properties. For other mapped attributes, OpenEdge provides static methods on a Progress.Open4GL.ProDataTable utility class that allow you to maintain these values in the corresponding ADO.NET DataTable.

Note: Progress.Open4GL.ProDataTable participates in the class hierarchy of strongly-typed DataTable classes generated by ProxyGen. However, you only use Progress.Open4GL.ProDataTable directly as a utility class in order to access its static methods.

The following sections describe the attribute mappings supported by OpenEdge using existing ADO.NET properties or OpenEdge ProDataTable class methods. For more information on mappings that use an OpenEdge ProDataTable class method, see the corresponding method description in ProDataTable class methods on page 82.

Temp-table and buffer attribute mappings

The following table lists ABL attributes of a temp-table parameter and its buffer (or of a temp-table that is part of a ProDataSet parameter) and how they map to the corresponding ADO.NET DataTable passed in a .NET Open Client.
Table 8: Mapping ABL temp-table and buffer attributes to ADO.NET

<table>
<thead>
<tr>
<th>ABL temp-table or buffer attribute</th>
<th>ADO.NET mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE-TABLE</td>
<td>Progress.Open4GL.ProDataTable.GetBImageFlag(System.Data.DataTable dt)</td>
<td>For the ADO.NET DataTable specified by <code>dt</code>, these static methods get or set an indication of whether the DataTable corresponds to an ABL temp-table that has a before-table defined for it on the AppServer using the BEFORE-TABLE option (or attribute). For more information, see Before-image methods on page 83.</td>
</tr>
<tr>
<td>DATA-SOURCE-MODIFIED</td>
<td>Progress.Open4GL.ProDataTable.GetDataSourceModified(System.Data.DataTable dt)</td>
<td>For the ADO.NET DataTable specified by <code>dt</code>, this static method returns an indication whether any data source row is modified from the original temp-table values passed from the AppServer as part of a ProDataSet. For more information, see GetDataSourceModified( ) on page 88.</td>
</tr>
<tr>
<td>ERROR-STRING</td>
<td>Progress.Open4GL.ProDataTable.GetErrorString(System.Data.DataTable dt)</td>
<td>For the ADO.NET DataTable specified by <code>dt</code>, these static methods get or set the value of the corresponding temp-table ERROR-STRING attribute. For more information, see Error handling methods on page 87. The RowError property gets or sets the ERROR-STRING attribute on the corresponding temp-table buffer (row value).</td>
</tr>
<tr>
<td>NAME</td>
<td>System.Data.DataTable.TableName</td>
<td>Gets or sets the name of the corresponding temp-table.</td>
</tr>
<tr>
<td>NAMESPACE-PREFIX</td>
<td>System.Data.DataTable.Prefix</td>
<td>Gets or sets the XML namespace prefix on the specified ADO.NET DataTable for the corresponding ABL temp-table.</td>
</tr>
<tr>
<td>NAMESPACE-URI</td>
<td>System.Data.DataTable.Namespace</td>
<td>Gets or sets the XML namespace on the specified ADO.NET DataTable for the corresponding ABL temp-table.</td>
</tr>
<tr>
<td>PRIMARY</td>
<td>System.Data.UniqueConstraint.System.Data.DataTable.PrimaryKey</td>
<td></td>
</tr>
</tbody>
</table>
### ABL temp-table to ADO.NET DataTable mappings

<table>
<thead>
<tr>
<th>ABL temp-table or buffer attribute</th>
<th>ADO.NET mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>The <code>DataTable.PrimaryKey</code> maps to the corresponding temp-table's primary index. It contains an array of <code>DataColmun</code> objects that compose the primary key. This same array of columns is contained in the <code>DataTable.Constraints</code> collections as a <code>UniqueConstraint</code> object, where the <code>UniqueConstraint.isPrimary</code> property is <code>true</code>.</td>
</tr>
<tr>
<td>REJECTED</td>
<td><code>Progress.Open4GL.ProDataTable.GetRejected(System.Data.DataTable dt)</code></td>
<td>For the ADO.NET DataTable specified by <code>dt</code>, this static method returns an indication of whether any changes from the corresponding temp-table values in the original ProDataSet were rejected on the AppServer. For more information, see <code>GetRejected()</code> on page 88.</td>
</tr>
<tr>
<td>ROW-STATE</td>
<td><code>System.Data.DataRow.RowState</code></td>
<td>Returns the current change state of the corresponding row (buffer) of a temp-table in a ProDataSet parameter.</td>
</tr>
<tr>
<td>UNDO</td>
<td><code>Progress.Open4GL.ProDataTable.GetUndo(System.Data.DataTable dt)</code></td>
<td>For the ADO.NET DataTable specified by <code>dt</code>, this static method returns an indication of whether the corresponding temp-table is defined as <code>UNDO</code> on the AppServer. For more information, see <code>GetUndo()</code> on page 90.</td>
</tr>
</tbody>
</table>

### Buffer-field attribute mappings

The following table lists ABL attributes of a buffer-field component from a row of a temp-table parameter (or of a temp-table that is part of a ProDataSet parameter) and how they map to the corresponding ADO.NET DataColumn in a row of the corresponding DataTable passed in a .NET Open Client.
Table 9: Mapping ABL buffer-field attributes to ADO.NET

<table>
<thead>
<tr>
<th>ABL buffer-field attribute</th>
<th>ADO.NET mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER-VALUE</td>
<td>System.Data.DataRow.Item</td>
<td>Gets or sets the current data value of a corresponding buffer-field.</td>
</tr>
<tr>
<td>COLUMN-LABEL</td>
<td>System.Data.DataColumn.Caption</td>
<td>Gets or sets the label for the column of data associated with the corresponding buffer-field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> If this attribute is blank, the Caption property maps the LABEL attribute, and if the LABEL attribute is blank, the property maps the NAME attribute.</td>
</tr>
<tr>
<td>DATA-TYPE</td>
<td>Progress.Open4GL.ProDataTable.GetColumnProType()</td>
<td>For a given ADO.NET DataTable or DataColumn (depending on overloading), these static methods get or set the ABL data type of the corresponding buffer-field. For more information, see Column-related (buffer-field) methods on page 84. The DataType property gets or sets a .NET data type that maps to the ABL data type of the corresponding buffer-field according to the mappings shown in Data type mapping for temp-table fields on page 67.</td>
</tr>
<tr>
<td></td>
<td>Progress.Open4GL.ProDataTable.SetColumnProType()</td>
<td>System.Data.DataColumn.DataType</td>
</tr>
<tr>
<td></td>
<td>Progress.Open4GL.ProDataTable.IsExtentColumn(System.Data.DataTable dt, string colName)</td>
<td>For the ADO.NET DataTable specified by $dt$, these static methods indicate if a given DataColumn is part of a corresponding ABL array field (IsExtentColumn( )) or set the location and number of DataColumns that comprise the extent associated with a corresponding ABL array field (SetExtentColumns( )). For more information, see Array methods on page 82.</td>
</tr>
<tr>
<td></td>
<td>Progress.Open4GL.ProDataTable.SetExtentColumns(System.Data.DataTable dt, string proFieldName, string startColName, int extent)</td>
<td>System.Data.DataColumn.DefaultValue</td>
</tr>
</tbody>
</table>

**Initial**
<table>
<thead>
<tr>
<th>ABL buffer-field attribute</th>
<th>ADO.NET mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Gets or sets the default value for the specified ADO.NET DataColumn when creating new rows. This value corresponds to the value of the initial schema field (always CHARACTER) in the corresponding ABL buffer-field object, formatted with the specified buffer-field format.

**Note:** When working with ABL DATE, DATETIME, and DATETIME-TZ temp-table fields, you must work around differences between the corresponding ABL and .NET data types. For more information, see Resolving differences between ABL and .NET date and time implementations on page 62.

<table>
<thead>
<tr>
<th>ABL buffer-field attribute</th>
<th>ADO.NET mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>System.Data.DataColumn.Unique</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indicates if the corresponding buffer-field participates in a temp-table index.

<table>
<thead>
<tr>
<th>ABL buffer-field attribute</th>
<th>ADO.NET mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL</td>
<td>System.Data.DataColumn.Caption</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Gets or sets the label for the corresponding buffer-field.

**Note:** Maps if the COLUMN-LABEL attribute for the corresponding buffer-field is blank. However, if LABEL is also blank, maps to the NAME attribute.

<table>
<thead>
<tr>
<th>ABL buffer-field attribute</th>
<th>ADO.NET mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANDATORY</td>
<td>System.Data.DataColumn.AllowDBNull</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indicates if a corresponding buffer-field is a required field.

<table>
<thead>
<tr>
<th>ABL buffer-field attribute</th>
<th>ADO.NET mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>System.Data.DataColumn.ColumnName</td>
</tr>
<tr>
<td></td>
<td>DataColumn.Caption</td>
</tr>
<tr>
<td></td>
<td>Progress.Open4GL.ProDataTable.GetArrayFieldName( )</td>
</tr>
</tbody>
</table>

**Description**

Gets or sets the name of the corresponding buffer-field. If the COLUMN-LABEL and LABEL attributes are both blank for a corresponding buffer-field, the Caption property also maps this attribute value. For a given ADO.NET DataTable or DataColumn (depending on overloading), the static method (GetArrayFieldName( )) gets the name of the corresponding array buffer-field of which the specified column is a part. For more information, see Array methods on page 82.
Resolving differences between ABL and .NET date and time implementations

An ABL DATE field can have its initial value set to the ABL TODAY built-in function, which returns the current date. Similarly, an ABL DATETIME or DATETIME-TZ field can have its initial value set to the ABL NOW built-in function, which returns the current date and time. When you create a new record for a temp-table in ABL that contains fields with these data types, and these fields are defined as initialized with the corresponding built-in functions, they are initialized to the date or date and time (respectively) when the record is created.

However, .NET does not provide exactly the same capability for the corresponding ADO.NET DataColumn of type System.DateTime. The System.Data.DataColumn.DefaultValue property sets a specific value for an object (the DataColumn). So, for a DateTime DataColumn, you must set a specific (and static) date or date and time.

To work around this limitation, you can programmatically set the value for any ADO.NET DataColumn of type System.DateTime using two .NET properties that correspond to the ABL TODAY and NOW built-in functions, thus:

1. The System.DateTime.Today property returns a DateTime instance initialized with the current date.
2. The System.DateTime.Now property returns a DateTime instance initialized with the current date and time.

Mapping the XML-NODE-TYPE attribute to ADO.NET

As shown in the above table, the XML-NODE-TYPE attribute on a given ABL buffer-field directly corresponds to the System.Data.DataColumn.ColumnMapping property for the corresponding ADO.NET DataColumn. The following table shows how the CHARACTER value of the ABL attribute corresponds to the ADO.NET MappingType Enumeration member values.
Table 10: Mapping the XML-NODE-TYPE attribute

<table>
<thead>
<tr>
<th>This XML-NODE-TYPE value . . .</th>
<th>Corresponds to this ADO.NET MappingType member value . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ATTRIBUTE&quot;</td>
<td>Attribute</td>
</tr>
<tr>
<td>&quot;ELEMENT&quot;</td>
<td>Element</td>
</tr>
<tr>
<td>&quot;HIDDEN&quot;</td>
<td>Hidden</td>
</tr>
<tr>
<td>&quot;TEXT&quot;</td>
<td>SimpleContent</td>
</tr>
</tbody>
</table>

ABL ProDataSet to ADO.NET DataSet mappings

The ABL DATASET (ProDataSet) maps closely to an ADO.NET DataSet. The ProDataSet is passed between .NET and the AppServer as a parameter to a procedure or user-defined function. For static ProDataSets, ProxyGen creates extended ADO.NET DataSet classes for each unique ProDataSet definition. For dynamic ProDataSets, ProxyGen uses the standard ADO.NET DataSet class for the parameters.

The ProDataSet is a collection of one or more member temp-table buffers. It also optionally contains a collection of data-relations among the member temp-table buffers. Each of these temp-table buffers can also have a data source associated with it. You can define temp-tables for ProDataSets in exactly the same way as for passing temp-table parameters (see ABL temp-table to ADO.NET DataTable mappings on page 56).

An ADO.NET DataSet shares all of these characteristics. Thus, OpenEdge supports:

- The creation of a ProDataSet on the AppServer
- Passing that ProDataSet to a .NET client application
- Allowing that application to manipulate the data using an ADO.NET DataSet
- Returning any data changes to the AppServer to update the data source for the ProDataSet

To support the mapping of ABL ProDataSets to ADO.NET DataSets, OpenEdge establishes detailed correspondences between ProDataSet components and similar ADO.NET objects. This includes the mapping of such elements as temp-table field data types and ProDataSet component attributes. For more information on the mapping of temp-table field data types, see Passing TABLE and TABLE-HANDLE parameters on page 66.

The following sections describe the mapping of ProDataSet components and their attributes:

- Mapping ProDataSet components on page 64
- Mapping ProDataSet component attributes on page 64
Mapping ProDataSet components

The following table shows how a given ABL ProDataSet and its related components map to its companion objects in ADO.NET.

Table 11: ABL ProDataSet component to ADO.NET object mapping

<table>
<thead>
<tr>
<th>ABL component</th>
<th>.NET object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDataSet</td>
<td>System.Data.Dataset or strongly typed DataSet</td>
</tr>
<tr>
<td>Buffer/Temp-Table</td>
<td>System.Data.DataTable System.Data.DataRow or strongly typed DataTable</td>
</tr>
<tr>
<td>Buffer Field</td>
<td>System.Data.DataColumn</td>
</tr>
<tr>
<td>Data-Relation</td>
<td>System.Data.DataRelation</td>
</tr>
<tr>
<td>Primary Key</td>
<td>System.Data.UniqueConstraint</td>
</tr>
<tr>
<td>Unique non-primary index</td>
<td>System.Data.UniqueConstraint</td>
</tr>
</tbody>
</table>

Mapping ProDataSet component attributes

In addition to mapping ProDataSet components, OpenEdge maps relevant attributes of ProDataSet components for use in a .NET Open Client. These ABL attributes are similar to .NET properties and some of them map to actual ADO.NET properties.

Attributes on ProDataSet and related ABL components are similar to object properties. They allow you to query and modify information for these components in ABL. Many of these attributes are not meaningful in the .NET environment and require no mapping by OpenEdge. Among those attributes that are meaningful, some correspond to equivalent ADO.NET properties that already maintain the same information for equivalent .NET objects. Still others require some form of mapping by OpenEdge in order to maintain the corresponding information in ADO.NET.

For those ABL attributes that require mapping, OpenEdge maps some of them to existing ADO.NET properties. For other attributes, OpenEdge provides static methods on a Progress.Open4GL.ProDataSet utility class that allow you to maintain these values in the corresponding ADO.NET DataSet.
Note: Progress.Open4GL.ProDataSet participates in the class hierarchy of strongly-typed DataSet classes generated by ProxyGen. However, you only use Progress.Open4GL.ProDataSet directly as a utility class in order to access its static methods.

The following sections describe the attribute mappings supported by OpenEdge using existing ADO.NET properties or OpenEdge ProDataSet class methods. For more information on mappings where OpenEdge provides a ProDataSet class method, see the corresponding method description in ProDataSet class methods on page 90.

ProDataSet attribute mappings

The following table lists ABL attributes of an ABL ProDataSet parameter and how they map to the corresponding ADO.NET DataSet parameter passed in a .NET Open Client.

Table 12: Mapping ABL ProDataSet attributes to ADO.NET

<table>
<thead>
<tr>
<th>ABL ProDataSet attribute</th>
<th>ADO.NET mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA-SOURCE-MODIFIED</td>
<td>Progress.Open4GL.ProDataSet.GetDataSourceModified(System.Data.DataSet ds)</td>
<td>For the ADO.NET DataSet specified by ds, this static method returns an indication whether any data source row is modified from the original temp-table values passed from the AppServer as part of the ProDataSet. For more information, see GetDataSourceModified() on page 88.</td>
</tr>
<tr>
<td>NAME</td>
<td>System.Data.DataSet.DataSetName</td>
<td>Gets or sets the name of the corresponding ProDataSet.</td>
</tr>
<tr>
<td>NAMESPACE-PREFIX</td>
<td>System.Data.DataSet.Prefix</td>
<td>Gets or sets the XML namespace prefix on the specified ADO.NET DataSet for the corresponding ABL ProDataSet.</td>
</tr>
<tr>
<td>NAMESPACE-URI</td>
<td>System.Data.DataSet.Namespace</td>
<td>Gets or sets the XML namespace on the specified ADO.NET DataSet for the corresponding ABL ProDataSet.</td>
</tr>
<tr>
<td>REJECTED</td>
<td>Progress.Open4GL.ProDataSet.GetRejected(System.Data.DataSet ds)</td>
<td>For the ADO.NET DataSet parameter specified by ds, this static method returns an indication of whether any changes from original ProDataSet values were rejected on the AppServer. For more information, see GetRejected() on page 88.</td>
</tr>
</tbody>
</table>

Data-relation attribute mappings

The following table lists ABL attributes of an ABL data-relation component passed as part of a ProDataSet parameter and how they map to the corresponding ADO.NET DataRelation of the corresponding ADO.NET DataSet parameter passed in a .NET Open Client.
Table 13: Mapping ABL data-relation attributes to ADO.NET

<table>
<thead>
<tr>
<th>ABL data-relation attribute</th>
<th>ADO.NET mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>System.Data.DataRelation.RelationName</td>
<td>Gets or sets the name of the corresponding ABL data-relation.</td>
</tr>
<tr>
<td>NESTED</td>
<td>System.Data.DataRelation.Nested</td>
<td>Gets or sets a logical (.NET boolean) value indicating whether child rows of the specified ADO.NET DataRelation are nested within parent rows when they are written as XML. If the value is true, the child row elements are nested within respective parent row elements. If the value is false, the child row elements follow their respective parent row elements in sequence.</td>
</tr>
</tbody>
</table>

Passing TABLE and TABLE-HANDLE parameters

This section describes mapping ABL temp-tables to ADO.NET DataTables, using the TABLE and TABLE-HANDLE parameters.

ABL procedures can pass 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 .NET Open Client environment provides the same capability using an ADO.NET DataTable. A .NET Open Client passes an ABL TABLE parameter using an ADO.NET strongly typed DataTable object that is generated by ProxyGen and mapped to the specified temp-table. This strongly typed DataTable object inherits from Progress.Open4GL.ProDataTable, which in turn inherits from the standard ADO.NET class, System.Data.DataTable.

TABLE–HANDLE parameters allow the transfer of dynamic temp-tables between Open Clients and the AppServer, without requiring a complete, static definition of the table on each side of the transfer (as with TABLE parameters). A .NET Open Client passes an ABL TABLE–HANDLE parameter using an ADO.NET DataTable object, System.Data.DataTable.

For each unique static temp-table (TABLE parameter), ProxyGen generates a strongly typed DataTable class, TypedDataTable.cs. TypedDataTable is the name of the static temp-table in the ABL procedure, appended with DataTable. For example, in a procedure with a static temp-table CustTT, TypedDataTable is CustTTDataTable. If other temp-table parameters in other methods in the proxy share the same schema, ProxyGen does not create additional strongly typed DataTable classes but uses the same strongly typed DataTable class for these methods. The name of the temp-table parameter does not matter when comparing the schema of temp-table parameters. However, field names, types, and indexes do need to match. (This differs from ABL in which field names and indexes do not need to match.)
Also, if ProxyGen detects two or more temp-table parameters with the same name but different schemas, ProxyGen creates multiple strongly typed DataTable classes, one for each unique schema. ProxyGen appends a unique number to the end of the temp-table name, to create unique class names. For example, if multiple procedures define a static temp-table CustTT, each with a different schema, TypedDataTable can be CustTTDataTable, CustTT2DataTable, and so on.

All strongly typed DataTable classes are defined in the following namespace:

```
[namespace.]StrongTypesNS
```

Where `namespace` is the General namespace you optionally entered in the .NET Client Details group of the ProxyGen Generate Proxies dialog box. (See Namespace for proxy objects on page 28.)

This section covers the following topics:

- Static and dynamic temp-tables on page 67
- Data type mapping for temp-table fields on page 67
- Unknown value (?) on page 69
- Temp-table array fields on page 69
- Indexes for temp-tables on page 69
- Passing a TABLE or TABLE-HANDLE as an INPUT parameter on page 69
- Providing ABL extensions for INPUT TABLE-HANDLE parameters on page 70
- Passing a TABLE or TABLE-HANDLE as an INPUT parameter on page 69
- Passing a TABLE or TABLE-HANDLE as an OUTPUT parameter on page 71
- Temp-table examples on page 71

**Static and dynamic temp-tables**

In ABL, you can pass a static temp-table using the TABLE parameter, in which case the table is copied from one procedure to the other. You also can pass a temp-table as a dynamic object, by using the TABLE-HANDLE parameter. In this case, the table also is copied, but without the expectation that there is a static temp-table definition in the procedure using the TABLE-HANDLE parameter.

The TABLE and TABLE-HANDLE parameters can be intermixed freely in procedure calls.

**Data type mapping for temp-table fields**

The following table shows the mappings between individual fields in an ABL temp-table to their corresponding data types in an individual ADO.NET DataTable.
## Table 14: Data type conversions for temp-table fields

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>System.Byte[ ]</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>System.String</td>
</tr>
<tr>
<td>CLOB</td>
<td>System.String</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>System.Int64</td>
</tr>
<tr>
<td>DATE</td>
<td>System.DateTime</td>
</tr>
<tr>
<td>DATETIME</td>
<td>System.DateTime</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>System.Decimal</td>
</tr>
<tr>
<td>HANDLE</td>
<td>System.Int64</td>
</tr>
<tr>
<td>INT64</td>
<td>System.Int6</td>
</tr>
<tr>
<td>INTEGER</td>
<td>System.Int32</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>System.Boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>System.Byte[ ]</td>
</tr>
<tr>
<td>RECID</td>
<td>System.Int64</td>
</tr>
<tr>
<td>ROWID</td>
<td>System.Byte[ ]</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>System.Int64</td>
</tr>
</tbody>
</table>
Unknown value (?)

All temp-table fields can contain the Unknown value (?). For an ADO.NET DataTable, this is specified by each corresponding ADO.NET DataColumn's AllowDBNull property being set to true. (The default value of a DataColumn's AllowDBNull property is true.)

Temp-table array fields

In ABL, fields in a temp-table can be defined with an extent option to create a one-dimensional array of the specified data type, with the number of elements specified by the extent. In .NET, an array field in an ABL temp-table is flattened, such that each array element becomes a separate column of the DataTable.

For static temp-table parameters, the strongly typed class generated by ProxyGen automatically flattens array fields in the table into the required number of columns. The meta data for the table knows which columns are part of the array and maps the data between the flattened model in .NET and the array model in ABL.

For dynamic temp-table parameters, the parameter is mapped to a .NET System.Data.DataTable class. For OUTPUT parameters, this DataTable is created with any array fields flattened, such that each array element becomes a separate column of the DataTable. For INPUT and INPUT-OUTPUT parameters, the client must create an instance of a .NET System.Data.DataTable to provide the schema and data. For temp-tables with array fields, the client must provide ABL-specific mappings for the DataTable. For more information, see Passing a TABLE or TABLE-HANDLE as an INPUT-OUTPUT parameter on page 70 and Providing ABL extensions for INPUT TABLE-HANDLE parameters on page 70.

Indexes for temp-tables

In ABL, you can define indexes for temp-tables, one of which can be marked as the primary index. An index can consist of one of more fields in the temp-table.

.NET DataTables can only represent unique indexes (unique primary indexes and unique non-primary indexes).

If an ABL procedure has a temp-table parameter or a ProDataSet parameter (which contains temp-tables) and a temp-table has one or more unique indexes, the resulting .NET DataTable has a UniqueConstraint object for each index. For the primary index, its corresponding UniqueConstraint.PrimaryKey property is set to true. All other non-unique indexes are ignored because there is no corresponding .NET constraint to which they can be mapped.

For more information on .NET DataTable constraints, see the Microsoft .NET documentation.

Passing a TABLE or TABLE-HANDLE as an INPUT parameter

When the client application passes a static temp-table (TABLE parameter) as an INPUT parameter, the client must provide an instance of the strongly typed DataTable class provided by ProxyGen. In this case, the definition of the temp-table is known at compile time, and the proxy already knows the schema (meta data) for the temp-table; therefore, the client application does not need to supply the schema, only the data. The client can add rows to the temp-table using the Rows.Add() method on the strongly typed DataTable class.
When the client application passes a dynamic temp-table (TABLE-HANDLE parameter) as an INPUT parameter, the proxy does not know the schema (meta data) for the temp-table. In this case, the client must provide an instance of System.Data.DataTable that contains both the schema and data for the TABLE-HANDLE parameter. The client can use the methods on the System.Data.DataTable class to add the schema and rows for the temp-table. Also, the client might need to define ABL-specific mappings for the DataTable. See Providing ABL extensions for INPUT TABLE-HANDLE parameters on page 70.

**Note:** It is important that the schema field order defined for the .NET DataTable match the physical order defined for the temp-table in ABL. See Providing ABL extensions for INPUT TABLE-HANDLE parameters on page 70.

Since all strongly typed DataTable classes inherit from System.Data.DataTable and contain extensions necessary to provide the ABL-specific mappings, these classes can also be passed as an INPUT TABLE-HANDLE parameter, as long as the ABL procedure is written to process the schema and data provided in the strongly typed DataTable class.

For INPUT parameters, all the data must be put into the parameter before the proxy call is made.

### Providing ABL extensions for INPUT TABLE-HANDLE parameters

For dynamic temp-table INPUT parameters, you can create an instance of the ADO.NET System.Data.DataTable class with the appropriate schema, data, and ABL-specific mappings, or you can use an existing instance (if appropriate) of a strongly-typed DataTable class as defined by ProxyGen.

To support ABL-specific features for a new DataTable instance, you might also need to specify ABL extensions for the DataTable that support the mapping to an ABL temp-table. ABL provides the Progress.Open4GL.ProDataTable utility class to define these ABL extensions for a DataTable. For more information, see ProDataTable class methods on page 82.

### Passing a TABLE or TABLE-HANDLE as an INPUT-OUTPUT parameter

INPUT-OUTPUT TABLE and TABLE-HANDLE parameters are a combination of INPUT and OUTPUT parameters. The client application must pass an object instance for the parameter as defined for input parameters. For a TABLE parameter, the client must provide an instance of the strongly typed DataTable class provided by ProxyGen. For a TABLE-HANDLE parameter, the client must provide a strongly typed DataTable (if applicable) or an instance of System.Data.DataTable that contains both the schema and data, as well as any ABL-specific mappings. For more information, see Passing a TABLE or TABLE-HANDLE as an INPUT parameter on page 69.

OUTPUT TABLE-HANDLE parameters can be returned as Unknown or undefined. As a result, you must ensure the client code is written to handle these cases.

For INPUT-OUTPUT parameters, all the data must be put into the parameter before the proxy call is made. Any data in the parameter at the time of the method is replaced on return from the proxy call. All output data is immediately available to the client after the proxy call returns.

During this process, the input object is supplied by the client, then cleared, repopulated with output data, and retuned. For this reason, the schema for the object must match on both sides of the call.
Passing a TABLE or TABLE-HANDLE as an OUTPUT parameter

When the client application calls a method with a static temp-table (TABLE parameter) OUTPUT parameter, the client must provide a variable of the strongly typed DataTable class provided by ProxyGen. In this case, the client does not need to instantiate the object.

**Note:** This strongly typed DataTable class instance can then be used to pass an appropriate proxy INPUT temp-table parameter.

When the client application calls a method with a dynamic temp-table (TABLE-HANDLE parameter) OUTPUT parameter, the client must provide a variable of type System.Data.DataTable. Again, the client does not need to instantiate the object. In this case, the parameter can be returned as Unknown or undefined. As a result, you must ensure the client code is written to handle these cases.

On return from the method, the client can access rows of the temp-table using the `Rows` property on the strongly typed `DataTable` or `System.Data.DataTable` class.

Any data in the parameter at the time of the method call is replaced on return from the proxy object. All output data is immediately available to the client after the proxy call returns.

Temp-table examples

This section provides some code examples for temp-tables.

**Note:** These samples are not available for download on the Progress Communities Web site.

Sample static temp-table

The following example shows a static temp-table definition in a persistent procedure, `PaymentInfo.p`.

**Static temp-table in a persistent procedure**

```openedge
DEFINE TEMP-TABLE PaymentInfoTT
FIELD payment-date AS DATE
FIELD payee-id AS INTEGER
FIELD payee-name AS CHARACTER
FIELD amount AS DECIMAL
FIELD cleared AS LOGICAL.

PROCEDURE getPaymentsInfo:
  DEFINE INPUT PARAMETER fromDate AS DATE.
  DEFINE OUTPUT PARAMETER TABLE FOR PaymentInfoTT.
```

For this parameter, ProxyGen generates the following:

- `PaymentInfoTT.cs` with the following class definition:

```csharp
Acme.StrongTypesNS.PaymentInfoTTDataTable
```
The following method in the proxy:

```java
public void getPaymentsInfo(System.DateTime fromDate,
    out Acme.StrongTypesNS.PaymentInfoTTDataTable PaymentInfoTT)
```

The following example shows a static temp-table definition in an external procedure, setPaymentInfo.p.

**Static temp-table in an external procedure**

```abl
DEFINE TEMP-TABLE MySetPaymentInfoTT
FIELD payment-date AS DATE
FIELD payee-id AS INTEGER
FIELD payee-name AS CHARACTER
FIELD amount AS DECIMAL
FIELD cleared AS LOGICAL.
DEFINE INPUT PARAMETER TABLE FOR MySetPaymentInfoTT.
```

Since the schema for these two static temp-table parameters is the same, ProxyGen uses the previously generated, strongly typed DataTable in the proxy method generated for this external procedure. For example:

```java
public void setPaymentInfo(System.DateTime fromDate,
    out Acme.StrongTypesNS.PaymentInfoTTDataTable MySetPaymentInfoTT)
```

For **INPUT** and **INPUT-OUTPUT** parameters, the .NET client code must supply an instance of the strongly typed DataTable object. For **OUTPUT** parameters, the strongly typed DataTable variable must be declared, but the instance is created by ABL and returned to the .NET client.

**Sample dynamic parameter**

The following example shows a **TABLE-HANDLE** parameter.

**TABLE-HANDLE parameter**

```abl
PROCEDURE getOrderInfo:
    DEFINE INPUT PARAMETER TABLE-HANDLE tth1.
```

For this parameter, ProxyGen generates the following method in the proxy:

```java
public void getOrderInfo(System.Data.DataTable tth1)
```

For **INPUT** and **INPUT-OUTPUT** parameters, the .NET client code must supply an instance of a strongly typed DataTable or an instance of a `System.Data.DataTable` object that contains the appropriate schema and data, as well as any required ABL-specific mappings. For **OUTPUT** parameters, the DataTable variable must be declared, but the instance is created by ABL and returned to the .NET client with all necessary ABL mappings. For more information on these mappings, see ABL temp-table to ADO.NET DataTable mappings on page 56.
Sample .NET Open Client application using a temp-table

The following examples show sample code for calling a procedure with a temp-table.

### Calling a procedure with an output dynamic temp-table

```csharp
// Calling a procedure with an output dynamic temp-table.
Account appObj = new Account("AppServer://myhost/asbroker1", "", "", "");
System.Data.DataTable outDT = null;

// Call the procedure
appObj.GetTTOut(out outDT);

// Run through the ABL DataTable. First output the schema.
int numCols = outDT.Columns.Count;
for (int ix = 0; ix < numCols; ix++)
{
    System.Console.WriteLine(outDT.Columns[ix].ColumnName + " " + outDT.Columns[ix].DataType.ToString( ));
}
// Output the data.
foreach (DataRow dr in outDT.Rows)
{ // Print the first column in the row.
    System.Console.WriteLine(" Column 1 " + dr.Columns[0]);
}
```

### Calling a procedure with an input dynamic temp-table

```csharp
// Calling a procedure with an input dynamic temp-table.
// The ABL DataTable has 2 columns, integer and string.
Account appObj = new Account("AppServer://myhost/asbroker1", "", "", "");
System.Data.DataTable inDT = new System.Data.DataTable( );
inDT.Columns.Add("acctnum", typeof(int));
inDT.Columns.Add("Name", typeof(string));
DataRow testRow;

// Add 3 rows of data.
for (int i = 1; i <= 3; i++)
{
    testRow = inDT.NewRow( );
    testRow[0] = i;
    testRow[1] = "Test String " + i.ToString( );
    inDT.Rows.Add(testRow);
}

// Call the procedure
appObj.SetTTIn(inDT);
```

### Passing DATASET and DATASET-HANDLE parameters

This section describes mapping ABL ProDataSets to ADO.NET DataSets, using the DATASET and DATASET-HANDLE parameters.
ABL procedures can pass DATASET or DATASET-HANDLE parameters to an AppServer. This is a convenient and efficient way to pass relational data between the client and the AppServer code. The .NET Open Client environment provides the same capability using an ADO.NET DataSet. A .NET Open Client passes an ABL DATASET parameter using an ADO.NET strongly typed DataSet object that is generated by ProxyGen and mapped to the specified ProDataSet. This strongly typed DataSet object inherits from Progress.Open4GL.ProDataSet, which in turn inherits from the standard ADO.NET System.Data.DataSet class.

DATASET-HANDLE parameters allow the transfer of dynamic ProDataSets between Open Clients and the AppServer, without requiring a complete, static definition of the ProDataSet on each side of the transfer (as with DATASET parameters). A .NET Open Client passes an ABL DATASET-HANDLE parameter using an ADO.NET strongly typed DataSet object, System.Data.DataSet.

For each unique static ProDataSet (DATASET parameter), ProxyGen generates a strongly typed DataSet class, TypedDataSet.cs. TypedDataSet is the name of the static ProDataSet in the ABL procedure, appended with DataSet. When comparing the static definitions of two ProDataSets, the name of the ProDataSet and the member buffers do not need to match for the ProDataSets to map to a single ADO.NET DataSet. However, field names, types, and indexes do need to match. (This differs from ABL in which field names and indexes do not need to match.) For example, in a procedure with a static ProDataSet CustOrderDS, TypedDataSet is CustOrderDSDataSet. Each temp-table in the ProDataSet is created as a DataTable within the DataSet class. If other ProDataSet parameters in other methods in the proxy share the same schema, ProxyGen does not create additional strongly typed DataSet classes but uses the same strongly typed DataSet class for these methods. The name of the ProDataSet parameter does not matter when comparing the schema of ProDataSet parameters.

Also, if ProxyGen detects two or more ProDataSet parameters with the same name but different schemas, ProxyGen creates multiple strongly typed DataSet classes, one for each unique schema. ProxyGen appends a unique number to the end of the DataSet name, to create unique class names. For example, if multiple procedures define a static ProDataSet CustOrderDS, each with a different schema, TypedDataSet can be CustOrderDSDataSet, CustOrderDS2DataSet, and so on.

All strongly typed DataSet classes are defined in the following namespace:

```
[namespace.]StrongTypesNS
```

Where `namespace` is the General namespace you optionally entered in the .NET Client Details group of the ProxyGen Generate Proxies dialog box. (See Namespace for proxy objects on page 28.)

This section covers the following topics:

- Static and dynamic ProDataSets on page 75.
- Data-type mapping for ProDataSets on page 75.
- Passing a DATASET or DATASET-HANDLE as an INPUT parameter on page 75.
- Providing ABL extensions for INPUT DATASET-HANDLE parameters on page 75.
- Passing a DATASET or DATASET-HANDLE as an INPUT-OUTPUT parameter on page 76.
- Passing a DATASET or DATASET-HANDLE as an OUTPUT parameter on page 76.
- Updating a DataSet on page 76.
- ProDataSet examples on page 77.
Static and dynamic ProDataSets

In ABL, you can pass a static ProDataSet using the `DATASET` parameter, in which case the ProDataSet is copied from one procedure to the other. You also can pass a ProDataSet as a dynamic object, by using the `DATASET-HANDLE` parameter. In this case, the ProDataSet also is copied, but without the expectation there is a static ProDataSet definition in the procedure using the `DATASET-HANDLE` parameter.

The `DATASET` and `DATASET-HANDLE` parameters can be intermixed freely in procedure calls.

Data-type mapping for ProDataSets

In ABL, since ProDataSets contain a collection of temp-tables, the data-type mapping defined for temp-tables is used for ProDataSets. See Data type mapping for temp-table fields on page 67.

Passing a DATASET or DATASET-HANDLE as an INPUT parameter

When the client application passes a static ProDataSet (`DATASET` parameter) as an INPUT parameter, the client must provide an instance of the strongly typed DataSet class provided by ProxyGen. In this case, the definition of the ProDataSet is known at compile time, and the proxy already knows the schema (meta data) for the ProDataSet, including the tables it contains; therefore, the client application does not need to supply the schema, only the data. The client can add rows to each of the tables within the dataset using the `Rows.Add( )` method on the DataTable class.

When the client application passes a dynamic ProDataSet (`DATASET-HANDLE` parameter) as an INPUT parameter, the proxy does not know the schema (meta data) for the ProDataSet. In this case, the client must provide an instance of `System.Data.DataSet` that contains both the schema and data for the `DATASET-HANDLE` parameter. The client can use the methods on the `System.Data.DataSet` class to add the schema, tables, and rows for each table of the ProDataSet. Also, the client might need to define ABL-specific mappings for the DataSet. See Providing ABL extensions for INPUT DATASET-HANDLE parameters on page 75.

Since all strongly typed DataSet classes inherit from `System.Data.DataSet` and contain extensions necessary to provide the ABL-specific mappings, these classes can also be passed for an INPUT DATASET-HANDLE parameter, as long as the ABL procedure is written to process the schema and data provided in the strongly typed DataSet class.

For INPUT parameters, all the data must be put into the parameter before the proxy call is made.

Providing ABL extensions for INPUT DATASET-HANDLE parameters

For dynamic ProDataSet INPUT parameters, you can create an instance of the ADO.NET `System.Data.DataSet` class, or you can use an existing instance (if appropriate) of a strongly-typed DataSet class as defined by ProxyGen. For a new `DataSet` instance, you must then define the schema of the DataSet and add the tables and any data.

To support ABL-specific features for a new `DataSet` instance, you might also need to specify ABL extensions for the DataTables within the DataSet that provide mappings to the corresponding ABL temp-tables. ABL provides the `Progress.Open4GL.ProDataTable` utility class to define these ABL extensions for each DataTable. For more information, see ProDataTable class methods on page 82. You might also need to specify ABL extensions for the DataSet that support the mapping to an ABL ProDataSet. ABL provides the `Progress.Open4GL.ProDataSet` utility class to define these ABL extensions for a DataSet. For more information, see ProDataSet class methods on page 90.
Passing a DATASET or DATASET-HANDLE as an INPUT-OUTPUT parameter

INPUT-OUTPUT DATASET and DATASET-HANDLE parameters are a combination of INPUT and OUTPUT parameters. The client application must pass an object instance for the parameter as defined for input parameters. For a DATASET parameter, the client must provide an instance of the strongly typed DataSet class provided by ProxyGen. For a DATASET-HANDLE parameter, the client must provide a strongly typed DataSet (if applicable) or an instance of System.Data.DataSet that contains both the schema and data, as well as any ABL-specific mappings.

DATASET-HANDLE parameters can be returned as Unknown or undefined. As a result, you must ensure the client code is written to handle these cases.

On return from the method, the client can access the rows of the tables contained in the ProDataSet using the Tables and Rows properties on the strongly typed DataSet or System.Data.DataSet class.

For INPUT-OUTPUT parameters, all the data must be put into the parameter before the proxy call is made. Any data in the parameter at the time of the method is replaced on return from the proxy call. All output data is immediately available to the client after the proxy call returns.

During this process, the input object is supplied by the client, then cleared, repopulated with output data, and returned. For this reason, the schema for the object must match on both sides of the call.

Passing a DATASET or DATASET-HANDLE as an OUTPUT parameter

When the client application calls a method with a static ProDataSet (DATASET parameter) OUTPUT parameter, the client must provide a variable of the strongly typed DataSet class provided by ProxyGen. In this case, the client does not need to instantiate the object.

Note: This strongly typed DataSet class instance can then be used to pass an appropriate proxy INPUT ProDataSet parameter.

When the client application calls a method with a dynamic ProDataSet (DATASET-HANDLE parameter) OUTPUT parameter, the client must provide a variable of type System.Data.DataSet. Again, the client does not need to instantiate the object. In this case, the parameter can be returned as Unknown or undefined. As a result, you must ensure the client code is written to handle these cases.

On return from the method, the client can access the rows of the tables contained in the ProDataSet using the Tables and Rows properties on the strongly typed DataSet or System.Data.DataSet class.

Any data in the parameter at the time of the method call is replaced on return from the proxy object. All output data is immediately available to the client after the proxy call returns.

Updating a DataSet

DataSets are useful for passing updated data between a client and server. Typically, the client obtains the DataSet from the AppServer, makes changes to the DataSet, and sends those changes back to the AppServer. The AppServer matches the records back to the original datasource, verifies that the records were not changed by other users, and then applies the changes to the datasource. The server passes back the final versions of records that may have been further changed on the server, along with any errors.
DataSets in both .NET and ABL have built-in support for updating. In addition to being an in-memory cache of data, DataSets keep track of which records were changed (modified, added, or deleted). They also keep track of the original or before-image versions of modified or deleted records, so the server can compare those records against the current records in the datasource. (ProDataSets on the AppServer need to define a before table to keep track of original values.)

The normal processing of DataSets for update is as follows:

1. Get an original DataSet to the client, normally using an OUTPUT parameter.
2. Make changes to the DataSet on the client.
3. When you are ready to send updates to the server, call the GetChanges method on the System.Data.DataSet class. This returns you a new System.Data.DataSet with only the changed (modified, added, or deleted) records, including the original versions of the modified and deleted records. This is the DataSet that should be sent back to the AppServer for update, since there is no reason to send back any records that did not change.

   **Note:** You may find it best to send updates to the AppServer whenever a single set of related records are updated, for example, a Customer record and all associated orders. Packaging updates simplifies the reconciliation process back from the AppServer to the original DataSet on the client.

4. Send the changes-only DataSet from Step 3 to the AppServer. Typically, this DataSet is sent as an INPUT-OUTPUT parameter, so the results from the AppServer’s DataSet update can be passed back to the client.
5. Process the return from the AppServer. On output, the DataSet contains the changes-only DataSet, including any further updated records or errors from the AppServer. Note that only updates to the current set of changed records are included, not updates by other users to records not in the current set.
6. Match each row in the changes-only DataSet back to the row in the original DataSet, and refresh the client DataSet as necessary. Once the refresh is complete, call the AcceptChanges( ) method on your DataSet. Your DataSet is now ready for more changes, as in Step 2.

### ProDataSet examples

This section provides some code examples for ProDataSets.

**Note:** These samples are not available for download on the Progress Communities Web site.

### Sample static ProDataSet

The following example shows a static ProDataSet definition in a persistent procedure, CustomersAndOrders.p.
Static ProDataSet in a persistent procedure

```plaintext
DEFINE TEMP-TABLE OrderInfo
    FIELD order-date AS DATE
    FIELD quantity AS INTEGER.

DEFINE TEMP-TABLE CustomerInfo
    FIELD cust-name AS CHARACTER
    FIELD address AS CHARACTER.

DEFINE DATASET OrderDS for OrderInfo, CustomerInfo.

PROCEDURE getOrders:
    DEFINE OUTPUT PARAMETER DATASET FOR OrderDS.
```

For this parameter, ProxyGen generates the following:

- OrderDS.cs with the following class definition:

```plaintext
Acme.StrongTypesNS.OrderDSDataSet
```

- The following method in the proxy:

```plaintext
public void getOrders(out Acme.StrongTypesNS.OrderDSDataSet orderDS)
```

The following example shows a static ProDataSet definition in an external procedure, SetCustomersAndOrders.p.

Static ProDataSet in a procedure

```plaintext
DEFINE TEMP-TABLE OrderInfo
    FIELD order-date AS DATE
    FIELD quantity AS INTEGER.

DEFINE TEMP-TABLE CustomerInfo
    FIELD cust-name AS CHARACTER
    FIELD address AS CHARACTER.

DEFINE DATASET MySetOrderDS FOR OrderInfo, CustomerInfo.

DEFINE INPUT PARAMETER DATASET FOR MySetOrderDS.
```

Since the schema for these two static DataSet parameters is the same, ProxyGen uses the previously generated, strongly typed DataSet in the proxy method generated for this external procedure. For example:

```plaintext
public void SetCustomersAndOrders(Acme.StrongTypesNS.OrderDSDataSet mySetOrderDS)
```
For **INPUT** and **INPUT-OUTPUT** parameters, the .NET client code must supply an instance of the strongly typed DataSet object. For **OUTPUT** parameters, the strongly typed DataSet variable must be declared, but the instance is created by ABL and returned to the .NET client as a parameter.

**Sample dynamic ProDataSet**

The following example shows a **DATASET-HANDLE** parameter.

**DATASET-HANDLE parameter**

```plaintext
PROCEDURE getOrders:
    DEFINE OUTPUT PARAMETER DATASET-HANDLE dsetHndl.
```

For this parameter, ProxyGen generates the following method in the proxy:

```plaintext
public void getOrders(out System.Data.DataSet dsetHndl)
```

For **INPUT** and **INPUT-OUTPUT** parameters, the .NET client code must supply an instance of a strongly typed DataSet or an instance of the `System.Data.DataSet` object that contains the appropriate schema and data, as well as any required ABL-specific mappings. For **OUTPUT** parameters, the DataSet variable must be declared, but the instance is created by ABL and returned to the .NET client as a parameter with all the necessary ABL mappings. For more information on these mappings, see [ABL ProDataSet to ADO.NET DataSet mappings](#) on page 63.

**Sample .NET Open Client application using a ProDataSet**

The following example shows sample code for calling a procedure with a ProDataSet.

**Calling a procedure with an output static ProDataSet**

```csharp
// The specified namespace is Acme
using Acme.StrongTypesNS;

// Calling a procedure with an output static ProDataSet.
Account appObj = new Account("AppServer://myhost/asbroker1", ",", ",", ");
OrderDSDataSet outDS = null;

// Call the procedure.
appObj.GetCustomersAndOrders(out outDS);

// Assign the ProDataSet to a DataGrid.
myForm.dataGrid1.DataSource = outDS;
myForm.dataGrid1.DataMember = "Customers";
myForm.dataGrid1.ReadOnly = true;
myForm.dataGrid1.CaptionText = "Number of Customers: " +
    outDS.Tables["ttCust"].Rows.Count;
```
Sample ProDataSet update

The code examples in this section show how to pass a changes-only DataSet (that is, a DataSet that contains only records that have been added, updated, or deleted) to the AppServer and properly process the changes.

The following example shows one way to handle updates in a .NET Open Client environment.

.NET update DataSet procedure

```csharp
//Define two strongly-typed DataSets
using Acme.StrongTypesNS;
dsOrderDataSet dsOrders;
dsOrderDataSet dsUpdatedRecs;

// Populate the dsUpdatedRecs DataSet with only the changed rows
dsUpdatedRecs = (dsOrderDataSet)(dsOrders.GetChanges( ));
// Confirm that there are updates before continuing
if (dsUpdatedRecs != null)
{
    appObj.UpdateDS(ref dsUpdatedRecs);
    if (dsUpdatedRecs.Tables["orderline"].HasErrors)
    {
        displayMessage = "The following Orderline rows were not updated:";
        foreach(DataRow failedRow in dsUpdatedRecs.Tables["orderline"].Rows)
        {
            if (failedRow.HasErrors)
            {
                displayMessage += "\n\r Line number: " + failedRow["linenum"].ToString( ) + " ,
                Error Message: " + failedRow.RowError;
            }
        }
    }
    else
    {
        dspMsg = "All updates succeeded";
        dspCaption = "Save completed";
        dspButton = MessageBoxButtons.OK;
        resultDlog = MessageBox.Show(this, dspMsg, dspCaption, dspButton,
                                   MessageBoxIcon.Information,
                                   MessageBoxButtonsDefaultButton.Button1,
                                   MessageBoxIconOptions.RightAlign);
    }
}
else
{
    dspMsg = "All updates succeeded";
    dspCaption = "Save completed";
    dspButton = MessageBoxButtons.OK;
    resultDlog = MessageBox.Show(this, dspMsg, dspCaption, dspButton,
                                  MessageBoxIcon.Information,
                                  MessageBoxButtonsDefaultButton.Button1,
                                  MessageBoxIconOptions.RightAlign);
}
// Remove the pending additional order lines before merging to prevent // duplicate rows if the key fields on the row number were changed during the // update.
foreach(DataRow curRow in dtTableTwo.Select("", "", DataViewRowState.Added))
{
    dtTableTwo.Rows.Remove(curRow);
}
// Merge the changes
dsOrders.Merge(dsUpdatedRecs);

    // Reset the row state for the modified rows now that changes
dsOrders.AcceptChanges( );
```
Note: For the sake of simplicity, this example does not include standard error checking. Make sure that you include error checking in your application code.

The following example shows temp-table definitions in an ABL include file, sOrderTables.i.

Include file definition of temp-table definitions

```abl
/* dsOrderTables.i -- include file for Temp-Table definitions */
DEFINE TEMP-TABLE ttOrder FIELDS (...)
   INDEX OrderNum IS UNIQUE PRIMARY OrderNum.
DEFINE TEMP-TABLE ttOLine FIELDS (...) BEFORE-TABLE ttOlineBefore
   INDEX orderline IS UNIQUE PRIMARY Ordemnum Linenum.
```

The following example shows a static ProDataSet definition in an ABL include file, dsOrderDef.i.

Include file definition of DataSet dsOrder

```abl
/* dsOrderDef.i -- include file definition of DATASET dsOrder. */
DEFINE DATASET dsOrder FOR ttOrder, ttOLine
   DATA-RELATION OrderLine FOR ttOrder, ttOLine
      RELATION-FIELDS (OrderNum, OrderNum).
```

The following example shows an ABL procedure that updates the database with a DataSet that contains only records that have been modified (added, updated, or deleted).

ABL update DataSet procedure

```abl
{OrderTables.i}
{OrderDS.i}
DEFINE INPUT-OUTPUT PARAMETER DATASET FOR dsOrder.
DEFINE VARIABLE hSrcOline AS HANDLE NO-UNDO.
DEFINE VARIABLE returnValue AS LOGICAL NO-UNDO.
DEFINE DATA-SOURCE srcOline FOR Orderline.
hSrcOline = DATA-SOURCE srcOline:HANDLE.
BUFFER ttOLine:ATTACH-DATA-SOURCE(hSrcOline).
FOR EACH ttOlineBefore TRANSACTION:
   returnValue = BUFFER ttOlineBefore:SAVE-ROW-CHANGES("Orderline") NO-ERROR.
END.
DELETE OBJECT hSrcOline.
RETURN.
```
**ProDataTable class methods**

The Open Client toolkit provides a Progress.Open4GL.ProDataTable utility class that you can use to extend the standard functionality available in the ADO.NET System.Data.DataTable class. These extensions complete features that allow any DataTable object to fully map an ABL temp-table. You can add these extensions to any DataTable using ProDataTable class (static) methods. These methods associate extended property name/value pairs with a specified ADO.NET DataTable object. These extended properties map meaningful functionality in a .NET environment that is available in ABL temp-tables but is not otherwise supported by the ADO.NET System.Data.DataTable class. You can use these methods to get or set the specified property values for both DataTable instances that you create and strongly-typed DataTable instances generated by ProxyGen.

The syntax for the methods available on this class are given in the following sections.

**Array methods**

The following methods provide information about array fields in a temp-table. (For more information, see Column-related (buffer-field) methods on page 84.)

**GetArrayFieldName( )**

Returns the array field name with which the specified DataColumn is associated. If it is not part of an array, it returns null. The syntax is:

**Syntax**

```
static string GetArrayFieldName(System.Data.DataTable dt, string colName)
static string GetArrayFieldName(System.Data DataColumn dc)
```

The array field name for a DataColumn is set when ProDataTable.SetExtentColumns() is called.

**IsExtentColumn( )**

Returns true if the specified column in the specified DataTable is part of a temp-table array field. The syntax is:

**Syntax**

```
static bool IsExtentColumn(System.Data.DataTable dt, string colName)
```

Where colName is the name of the column in the DataTable.

**SetExtentColumns( )**

Specifies a set of columns that are the flattened representation of a temp-table array field for the specified DataTable. The syntax is:
Syntax

```csharp
static void SetExtentColumns(System.Data.DataTable dt, string proFieldName,
                              string startColName, int extent)
```

Where:

- `proFieldName` is the name of the array field in the temp-table. If `proFieldName` is not specified, the value of `startColName` is used.
- `startColName` is the name of the first flattened column in the DataTable.
- `extent` is the number of columns, including `startColName`, to include in the array.

For example, a DataTable with the columns Name, Phone, Month1, Month2, Month3, Month4, Month5, Month6, Month7, Month8, Month9, Month10, Month11, and Month12 that mapped to a temp-table with three columns, Name, Phone, and Month with extent 12, would need to have a call to this method as follows:

```csharp
ProDataTable.SetExtentColumns(myDataTbl, "Month", "Month1", 12)
```

Before-image methods

The following methods get or set a flag based on whether the corresponding ABL temp-table has a before-table defined for it on the AppServer using the ABL `BEFORE-TABLE` option (or attribute). This before-table is a temp-table that contains the original records of an ABL temp-table that have been modified as part of a changed ProDataSet, and it corresponds to the set of original DataRow instances maintained internally for a changed ADO.NET DataTable.

**GetBImageFlag( )**

**Syntax**

Returns `true` if the specified DataTable maps to an ABL temp-table that has a before-table defined for it. The syntax is:

```csharp
static bool GetBImageFlag(System.Data.DataTable dt)
```

**SetBImageFlag( )**

Sets a value for `flag` that indicates if the DataTable specified by `dt` maps to an ABL temp-table that has a before-table defined for it on the AppServer. The syntax is:

```csharp
static void SetBImageFlag(System.Data.DataTable dt, bool flag)
```
If you set `flag` to `true`, you can only pass `dt` back to an AppServer as part of an ADO.NET DataSet that corresponds to a ProDataSet `INPUT` parameter. You must also ensure that the corresponding ABL temp-table has a before-table defined for it on the AppServer.

**Column-related (buffer-field) methods**

The following methods access information about temp-table fields (buffer-fields) that correspond to a specified DataTable column. (See also, Array methods on page 82.) For methods that get or set the ABL data type of a DataTable column, the `proType` argument must correspond to one of the valid ABL temp-table field data types found in the `Progress.Open4GL.Parameter` class, as shown in the following table.
<table>
<thead>
<tr>
<th>ABL temp-table field data type</th>
<th>Progress.Open4GL.Parameterclass constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>PRO_BLOB</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>PRO_CHARACTE</td>
</tr>
<tr>
<td>CLOB</td>
<td>PRO_CLOB</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>PRO_COMHANDLE</td>
</tr>
<tr>
<td>DATE</td>
<td>PRO_DAT</td>
</tr>
<tr>
<td>DATETIME</td>
<td>PRO_DATETIME</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>PRO_DATETIMETZ</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PRO_DECIMAL</td>
</tr>
<tr>
<td>INT64</td>
<td>PRO_INT64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>PRO_INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>PRO_LOGICAL</td>
</tr>
<tr>
<td>RAW</td>
<td>PRO_RAW</td>
</tr>
<tr>
<td>RECID</td>
<td>PRO_RECID</td>
</tr>
<tr>
<td>ROWID</td>
<td>PRO_ROWID</td>
</tr>
<tr>
<td>WIDGETHANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
</tbody>
</table>
GetColumnProType( )

Returns the integer value representing the ABL data type (see Table 15) for the specified DataColumn. The syntax is:

**Syntax**

```csharp
static int GetColumnProType(System.Data.DataTable dt, string colName)
```

Where `colName` is the name of the column in the DataTable.

```csharp
static int GetColumnProType(System.Data.DataColumn dc)
```

Where `dc` is the DataColumn.

GetPosition( )

Gets the position in the original database record of the database field that corresponds to the temp-table buffer-field object mapped to the `DataColumn` object, `dc`. The syntax is:

**Syntax**

```csharp
static int GetPosition(System.Data.DataColumn dc)
```

**Note:** This setting applies only to ABL buffer-field objects that originate in an OpenEdge RDBMS.

getUserOrder( )

Returns the 0-based display position of the specified DataColumn within its DataTable. The syntax is:

**Syntax**

```csharp
static int GetUserOrder(System.Data.DataColumn dc)
```
**SetColumnProType( )**

Sets the ABL data type for the specified data column, where `proType` must correspond to one of the valid ABL-specific class constants found in the `Progress.Open4GL.Parameter` class (see Table 15). The syntax is:

**Syntax**

```csharp
static void SetColumnProType(System.Data.DataTable dt, string colName, int proType)
```

Where `colName` is the name of the column in the DataTable.

```csharp
static void SetColumnProType(System.Data.DataColumn dc, int proType)
```

Where `dc` is the DataColumn.

**SetUserOrder( )**

Sets the 0-based display position of the specified DataColumn. The syntax is:

**Syntax**

```csharp
static void SetUserOrder(System.Data.DataColumn, int userOrder)
```

**Error handling methods**

The following methods get or set an error string associated with the specified DataTable returned from or sent to ABL. These methods map to the temp-table `ERROR-STRING` attribute.

**GetErrorString( )**

When passed a DataTable that corresponds to an `INPUT-OUTPUT` or `OUTPUT` temp-table parameter, returns any error information provided by the remote ABL method. The syntax is:

**Syntax**

```csharp
static string GetErrorString(System.Data.DataTable dt)
```
SetErrorString( )

Sets the ABL error string associated with the specified DataTable. When passed a DataTable that corresponds to an INPUT or INPUT-OUTPUT temp-table parameter, you can use the following method to relay the error string to the remote ABL method:

Syntax

```csharp
static void SetErrorString(System.Data.DataTable dt, string newValue)
```

Where `newValue` is the error string value.

Data source related methods

The following methods return information about updates to the AppServer data source for a specified temp-table that is passed as part of a ProDataSet parameter.

GetDataSourceModified( )

Returns `true` if a data source row associated with the specified DataTable was modified from original temp-table values. This method maps to the temp-table DATA-SOURCE-MODIFIED attribute that is typically set during invocation of the ABL buffer object `SAVE-ROW-CHANGES( )` method on the AppServer. The syntax is:

Syntax

```csharp
static bool GetDataSourceModified(System.Data.DataTable dt)
```

You typically query this value on the corresponding temp-table from an output ProDataSet parameter that was previously passed as input with changes from the .NET Open Client.

GetRejected( )

Returns `true` if any changes to the specified DataTable from the corresponding temp-table values in the original ProDataSet were rejected on the AppServer. This method maps to the temp-table REJECTED attribute. The syntax is:

Syntax

```csharp
static bool GetRejected(System.Data.DataTable dt)
```

You typically query this value on the corresponding temp-table of an output ProDataSet parameter that was previously passed as input with changes from the .NET Open Client.
Schema marshaling methods

The following methods verify or specify if schema information is to be marshalled with a given DataTable that is passed to the AppServer.

**Note:** ABL temp-tables support a `SCHEMA-MARSHAL` attribute that controls the marshalling of schema information from the AppServer to the client. However, these methods do not map that attribute, but provide corresponding functionality for a DataTable.

### GetNoSchemaMarshal( )

Returns `true` if the DataTable parameter is not marshaled with schema information to the AppServer; it returns `false` if schema information is marshaled. The syntax is:

**Syntax**

```csharp
static bool GetNoSchemaMarshal(System.Data.DataTable dt)
```

### SetNoSchemaMarshal( )

When `flag` is set to `true`, allows the specified DataTable to be marshaled from the .NET client to the AppServer without schema information. The syntax is:

**Syntax**

```csharp
static void SetNoSchemaMarshal(System.Data.DataTable dt, bool flag)
```

Using this method suppresses index descriptions and all field information and aids in faster transmission of data, thereby increasing the performance of your application.

You can use this method when the receiving side knows the schema definition for the table and validation is not necessary.

The corresponding ABL temp-table on the AppServer must have the same schema as the DataTable on the client. You can use the `SetNoSchemaMarshal( )` method with a strongly typed DataTable parameter or a DataTable within a strongly typed DataSet parameter. If set to `true`, when the DataTable is sent to the AppServer, the AVM generates a `Progress.Open4GL.Exceptions.Runtime4GLErrorException` if the ABL temp-table's schema does not match the incoming data from the .NET client.

Also see `PROGRESS.Session.NoSchemaMarshal` on page 114 for related functionality.

### Transaction methods

OpenEdge provides a single method that returns information about how the corresponding temp-table behaves during an OpenEdge transaction.
GetUndo( )

Returns true if the corresponding temp-table (passed as a parameter or as part of a ProDataSet parameter) is defined statically as UNDO or has its temp-table object UNDO attribute set to TRUE. The syntax is:

Syntax

```
static bool GetUndo(System.Data.DataTable dt)
```

ProDataSet class methods

The Open Client toolkit provides a Progress.Open4GL.ProDataSet utility class that you can use to extend the standard functionality available in the ADO.NET System.Data.DataSet class. These extensions complete features that allow any DataSet object to fully map an ABL ProDataSet. You can add these extensions to any ADO.NET DataSet using ProDataSet class (static) methods. These methods associate extended property name/value pairs with a specified ADO.NET DataSet object. These extended properties map meaningful functionality in a .NET environment that is available in ABL ProDataSets but is not otherwise supported by the ADO.NET System.Data.DataSet class. You can use these methods to get or set the specified property values for both DataSet instances that you create and the strongly-typed DataSet instances generated by ProxyGen.

The syntax for the methods available on this class are given in the following sections.

Data source update-related methods

The following methods return information about updates to the AppServer data sources for a specified ProDataSet parameter.

GetDataSourceModified( )

Returns true if any data source row in DataTables associated with the specified DataSet was modified from original ProDataSet values. This method maps to the ProDataSet DATA-SOURCE-MODIFIED attribute that is typically set during invocation of the temp-table buffer object SAVE-ROW-CHANGES( ) method. The syntax is:

Syntax

```
static bool GetDataSourceModified(System.Data.DataSet ds)
```

You typically query this value on an output ProDataSet parameter that was previously passed as input with changes from the .NET Open Client.

GetRejected( )

Returns true if any changes to the specified DataSet from the corresponding values in the original ProDataSet were rejected on the AppServer. This method maps to the ProDataSet REJECTED attribute. The syntax is:
Syntax

```csharp
static bool GetRejected(System.Data.DataSet ds)
```

You typically query this value on an output ProDataSet parameter that was previously passed as input with changes from the .NET Open Client.
Accessing Proxy Properties

The Open Client Runtime provides several properties that determine the behavior of the client application. These properties are called *proxy properties*. You can specify proxy properties default behavior with an application configuration file and/or by calling methods on the OpenEdge-provided classes, `Progress.Open4GL.RunTimeProperties` and `Progress.Open4GL.Connection`.

There are several properties that govern behavior across an entire application. These properties do not affect any particular object, but they affect the behavior of all objects created by the application. They are referred to as *proxy run-time properties*.

Other properties govern behavior across a single application service. These properties are called *Connection properties*.

This chapter describes the different ways you can access proxy properties.

For details, see the following topics:

- Application configuration file
- `RunTimeProperties` and `Connection` classes
- Accessing properties with methods
- Accessing properties directly
- Relationship between `RunTimeProperties` and `Connection` objects
- Modifying an instantiated `Connection` object's properties
- Alphabetical listing of properties
Application configuration file

You can control some of the capabilities of a client application using an application configuration file. This allows the application capabilities to be modified without requiring any changes to the client code. The application configuration file is optional; if none is provided, the default behavior is used.

For a list of the properties that you can set in the configuration file, see Alphabetical listing of properties on page 108.

To create an application configuration file, you can define an XML file for the application, with the extension .config. For example, if the client application is named MyApp.exe, create an XML file named MyApp.exe.config (in the same directory as the .exe file). A sample application configuration file follows:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="PROGRESS.Session.LoggingLevel" value="2" />
    <add key="PROGRESS.Session.EnableTracing" value="0" />
  </appSettings>
</configuration>
```

For more information about application configuration files in .NET, see the Microsoft .NET documentation.

RunTimeProperties and Connection classes

.NET Open Client provides two classes that you can use to access the properties of an application:

- RunTimeProperties class
- Connection class

RunTimeProperties class

The Progress.Open4GL.RunTimeProperties class contains static methods and global property values. You can call the methods on this class to change the value of the proxy properties. You can also access the properties directly.

The RunTimeProperties class has a set of default values. Any properties set using an application configuration file are represented by the RunTimeProperties class.

ServerName RunTime Property

You can set the Subject Name Indicator (SNI) name for a .NET client by using the following property:

```csharp
Progress.Open4GL.RunTimeProperties.ServerName = "servername";
```

servername

Specifies the server name in SSL connection initialization for client that it sends to the server as part of the TLS negotiation.
For example, `Progress.Open4GL.RunTimeProperties.ServerName = "www.example.com";`

If the .NET client does not send any valid SNI name available in the server configuration, the server selects a default virtual hostname and returns the respective certificate.

### Connection class

The `Progress.Open4GL.Connection` class stores AppServer connection information, which can be passed to the AppObject constructor when connecting to the AppServer. It contains methods and property values; however, it does not contain any static methods. You must create an instance of the class and call methods to change the value of the proxy properties.

When you create an instance of the `Connection` class (a `Connection` object), it inherits the initial values of its properties from the `RunTimeProperties` class. You can then override these values programmatically.

When you create an instance of the `Connection` class (a `Connection` object), it inherits the initial values of its properties from the `RunTimeProperties` class. You can then override these values programmatically. These changes only affect the proxy objects that are associated with this `Connection` object.

If you change the value of a property in a `Connection` object after it has been instantiated, the `Connection` object might not recognize the change, depending on the session model. For more information, see [Modifying an instantiated Connection object’s properties](#) on page 106.

**Note:** Changes to the properties of a `Connection` object have no effect on the global values of the `RunTimeProperties` class.

---

### Accessing properties with methods

You can access properties on the `RunTimeProperties` class and on instances of the `Connection` class using a set of general property accessor methods. These methods access properties according to the method type (get or set) and the property name:

- **General property accessor methods for the `RunTimeProperties` class** on page 95
- **General property accessor methods for `Connection` objects** on page 96
- **Available properties** on page 97
- **Accessing properties using purposed accessor methods** on page 99

### General property accessor methods for the `RunTimeProperties` class

You can access global properties by property name using the `get` and `set` general purpose methods. When you use these methods, you must call the correct accessor based on the data type of the property. The following is the syntax for these general methods:
Chapter 5: Accessing Proxy Properties

Get property methods syntax

```csharp
public static string GetStringProperty(string propName)
public static int GetIntProperty(string propName)
public static long GetLongProperty(string propName)
public static boolean GetBooleanProperty(String propName)
```

Set property methods syntax

```csharp
public static void SetStringProperty(string propName, string propValue)
public static void SetIntProperty(string propName, int propValue)
public static void SetLongProperty(string propName, long propValue)
public static void SetBooleanProperty(string propName, boolean propValue)
```

`propName`

The property name. This can be any of the properties listed in the tables in Available properties on page 97.

`propValue`

The value for `propName`.

Example

For example, to set the `PROGRESS.Session.InitialConnections` property to 5, use the following method to set an integer property:

```csharp
RunTimeProperties.SetIntProperty("PROGRESS.Session.InitialConnections", 5);
```

General property accessor methods for Connection objects

You can access `Connection` object properties by property name using the `get` and `set` general purpose methods. When you use these methods, you must call the correct accessor based on the data type of the property. The following is the syntax for these general methods:

Get property methods syntax

```csharp
public string GetStringProperty(string propName)
public int GetIntProperty(string propName)
public long GetLongProperty(string propName)
public boolean GetBooleanProperty(String propName)
```
Set property methods syntax

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void SetStringProperty(string propName, string propValue)</td>
<td></td>
</tr>
<tr>
<td>public void SetIntProperty(string propName, int propValue)</td>
<td></td>
</tr>
<tr>
<td>public void SetLongProperty(string propName, long propValue)</td>
<td></td>
</tr>
<tr>
<td>public void SetBooleanProperty(string propName, boolean propValue)</td>
<td></td>
</tr>
</tbody>
</table>

**propName**

The property name. This can be any of the properties listed in the tables in Available properties on page 97.

**propValue**

The value for propName.

**Example**

For example, to set the PROGRESS.Session.InitialConnections property to 5, use the following method to set an integer property:

```csharp
ConnectObj.SetIntProperty("PROGRESS.Session.InitialConnections", 5);
```

Where ConnectObj is an instance of the Progress.Open4GL.Proxy.Connection class.

**Available properties**

The following table lists all of the available properties according to their functional category. For more information on these property categories, see Accessing properties directly on page 100. For a detailed description of each property, see Alphabetical listing of properties on page 108.
## Table 16: Available properties

<table>
<thead>
<tr>
<th>Category</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Session</strong></td>
<td>• PROGRESS.Session.SessionModel on page 119</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.InitialConnections on page 112</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.MaxConnections on page 113</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.MinConnections on page 113</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.IdleConnectionTimeout on page 111</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.ConnectionLifetime on page 110</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.NsClientMinPort on page 115</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.NsClientMaxPort on page 115</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.NsClientPortRetry on page 116</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.NsClientPortRetryInterval on page 117</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.NsClientPicklistSize on page 116</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.NsClientPicklistExpiration on page 116</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.RequestWaitTimeout on page 119</td>
</tr>
<tr>
<td><strong>Thread control</strong></td>
<td>• PROGRESS.Session.TcpKeepAliveInterval on page 120</td>
</tr>
<tr>
<td><strong>Proxy server</strong></td>
<td>• PROGRESS.Session.ProxyHost on page 117</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.ProxyPort on page 118</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.ProxyUserId on page 118</td>
</tr>
<tr>
<td></td>
<td>• PROGRESS.Session.ProxyPassword on page 117</td>
</tr>
<tr>
<td><strong>Secure Sockets Layer</strong></td>
<td>• PROGRESS.Session.NoHostVerify on page 113</td>
</tr>
<tr>
<td><strong>management</strong></td>
<td>• PROGRESS.Session.NoSslSessionReuse on page 115</td>
</tr>
</tbody>
</table>
## Accessing properties using purposed accessor methods

In addition to the general property accessor methods, you can access some properties using purposed methods.

The methods in the following table provide tracing functionality to the Open Client application. These methods apply to the RunTimeProperties class only. (They do not apply to Connection objects.)

### Table 17: Tracing methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TraceOn(String ConnectObj.SetIntProperty, int ConnectObj.SetIntProperty)</code></td>
<td>Sets these properties:</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.EnableTracing</code> on page 111=true</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.LogFileName</code> on page 112</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.LoggingLevel</code> on page 112</td>
</tr>
<tr>
<td><code>TraceOn(int ConnectObj.SetIntProperty)</code></td>
<td>Sets these properties:</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.EnableTracing</code> on page 111=true</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.LoggingLevel</code> on page 112</td>
</tr>
<tr>
<td><code>TraceOn(String ConnectObj.SetIntProperty)</code></td>
<td>Sets these properties:</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.EnableTracing</code> on page 111=true</td>
</tr>
<tr>
<td></td>
<td>• <code>PROGRESS.Session.LogFileName</code> on page 112</td>
</tr>
</tbody>
</table>
Accessing properties directly

In addition to using the property accessor methods, you can access properties directly.

When accessing these properties on the RunTimeProperties class, use the class name to access the property. For example:

```
RunTimeProperties.InitialConnections = 5
```

When calling these properties on a Connection object, use the instance of the object to access the property. For example:

```
ConnectObj.SetIntProperty.InitialConnections = 5
```

Where ConnectObj is an instance of the Progress.Open4GL.Proxy.Connection class.

Session properties

The following table lists the session properties. For a detailed description of each property, see Alphabetical listing of properties on page 108.
Table 18: Session properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionModel</td>
<td>Sets and gets <code>PROGRESS.Session.SessionModel</code> on page 119</td>
</tr>
<tr>
<td>InitialConnections</td>
<td>Sets and gets <code>PROGRESS.Session.InitialConnections</code> on page 112</td>
</tr>
<tr>
<td>MaxConnections</td>
<td>Sets and gets <code>PROGRESS.Session.MinConnections</code> on page 113</td>
</tr>
<tr>
<td>MinConnections</td>
<td>Sets and gets <code>PROGRESS.Session.MinConnections</code> on page 113</td>
</tr>
<tr>
<td>IdleConnectionTimeout</td>
<td>Sets and gets <code>PROGRESS.Session.IdleConnectionTimeout</code> on page 111</td>
</tr>
<tr>
<td>ConnectionLifetime</td>
<td>Sets and gets <code>PROGRESS.Session.ConnectionLifetime</code> on page 110</td>
</tr>
<tr>
<td>NsClientMinPort</td>
<td>Sets and gets <code>PROGRESS.Session.NsClientMinPort</code> on page 115</td>
</tr>
<tr>
<td>NsClientMaxPort</td>
<td>Sets and gets <code>PROGRESS.Session.NsClientMaxPort</code> on page 115</td>
</tr>
<tr>
<td>NsClientPortRetry</td>
<td>Sets and gets <code>PROGRESS.Session.NsClientPortRetry</code> on page 116</td>
</tr>
<tr>
<td>NsClientRetryInterval</td>
<td>Sets and gets <code>PROGRESS.Session.NsClientPortRetryInterval</code> on page 117</td>
</tr>
<tr>
<td>NsClientPicklistSize</td>
<td>Sets and gets <code>PROGRESS.Session.NsClientPicklistSize</code> on page 116</td>
</tr>
<tr>
<td>NsClientPicklistExpiration</td>
<td>Sets and gets <code>PROGRESS.Session.NsClientPicklistExpiration</code> on page 116</td>
</tr>
<tr>
<td>RequestWaitTimeout</td>
<td></td>
</tr>
</tbody>
</table>
Thread control property

The AppServer can be accessed from a multi-threaded client. However, the AppServer itself is not multi-threaded. As a result, only one request can be serviced at a time. If a method 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 an exception

The application can set the run-time properties to do either and can switch the setting at any time. The default is to throw an exception (WaitIfBusy=false).

The property in the following table provides thread control to the Open Client Runtime. For a detailed description of this property, see Alphabetical listing of properties on page 108.

Table 19: Thread control property

<table>
<thead>
<tr>
<th>Property</th>
<th>Description/Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaitIfBusy</td>
<td>Sets and gets PROGRESS.Session.TcpKeepAliveInterval on page 120</td>
</tr>
</tbody>
</table>

Proxy server specification properties

You use proxy server specification properties if a Proxy Web server exists between the client and the Web server hosting the AppServer Internet Adapter (AIA).

The properties in the following table provide Proxy Web server support to a .NET Open Client application. For a detailed description of each property, see Alphabetical listing of properties on page 108.
Table 20: Proxy server specification properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProxyHost</td>
<td>Sets and gets <code>PROGRESS.Session.ProxyHost</code> on page 117</td>
</tr>
<tr>
<td>ProxyPort</td>
<td>Sets and gets <code>PROGRESS.Session.ProxyPort</code> on page 118</td>
</tr>
<tr>
<td>ProxyUserId</td>
<td>Sets and gets <code>PROGRESS.Session.ProxyUserId</code> on page 118</td>
</tr>
<tr>
<td>ProxyPassword</td>
<td>Sets and gets <code>PROGRESS.Session.ProxyPassword</code> on page 117</td>
</tr>
</tbody>
</table>

**Secure Sockets Layer management properties**

When using HTTPS or direct SSL to access the AppServer, you might need to configure the SSL connections with your client application.

The properties in the following table provide SSL connection options to the Open Client application. For a detailed description of each property, see Alphabetical listing of properties on page 108.

Table 21: Secure Sockets Layer management properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoHostVerify</td>
<td>Sets and gets <code>PROGRESS.Session.NoHostVerify</code> on page 113</td>
</tr>
<tr>
<td>NoSessionReuse</td>
<td>Sets and gets <code>PROGRESS.Session.NoSslSessionReuse</code> on page 115</td>
</tr>
</tbody>
</table>

**Tracing property (RunTimeProperties only)**

You can use tracing to log highlights of proxy execution, including data received by the client and data passed to the AppServer. The property in the following table provides tracing functionality to the Open Client application. This property applies to the `RunTimeProperties` class only. (It does not apply to `Connection` objects.) For a detailed description of this property, see Alphabetical listing of properties on page 108.

Table 22: Tracing property

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnableTracing</td>
<td>Gets and sets <code>PROGRESS.Session.EnableTracing</code> on page 111</td>
</tr>
</tbody>
</table>
Compression property

You can compress messages exchanged between the .NET Open Client and the AppServer. The properties listed in the following table provide compression functionality to the .NET Open Client.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompressionEnabled()</td>
<td>Gets and sets \texttt{PROGRESS.Session.TcpKeepAliveInterval} on page 120</td>
</tr>
<tr>
<td></td>
<td>Returns \texttt{TRUE} if \texttt{getCompressionEnabled()} is called. The default value is \texttt{FALSE},</td>
</tr>
<tr>
<td>CompressionLevel()</td>
<td>Gets and sets \texttt{PROGRESS.Session.CompressionLevel} on page 109</td>
</tr>
<tr>
<td>CompressionThreshold()</td>
<td>Gets and Sets \texttt{PROGRESS.Session.CompressionThreshold} on page 110</td>
</tr>
</tbody>
</table>

You can enable compression for the .NET Open Client using the following syntax:

```csharp
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Progress.Open4GL.Proxy;
using Progress.Open4GL.DynamicAPI;

namespace AppServerCompressionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Connection conn = new Connection(null, null, "AppServer://localhost:5162/asbroker1");
            conn.CompressionEnabled = true;
            conn.CompressionLevel = 5;
            conn.CompressionThreshold = 256;

            OpenAppObject app = new OpenAppObject(conn, "testapp");
            ParameterSet pset = new ParameterSet(2);
            pset.setLongCharParameter(0, "longchar value", ParameterSet.INPUT);
            pset.setLongCharParameter(1, "", ParameterSet.OUTPUT);
            app.runProcedure("echo.p", pset);

            String echoret = (String) pset.getOutputParameter(1);
            Console.WriteLine("Return : " + echoret);

            app.Dispose();
            conn.ReleaseConnection();
        }
    }
}
```
The following example uses the RunTimeProperties object code to illustrate that compression is enabled, compression level is set to 5, and the message size is set to 256 bytes.

```csharp
Progress.Open4GL.RunTimeProperties.CompressionEnabled = true;
Progress.Open4GL.RunTimeProperties.CompressionLevel = 5;
Progress.Open4GL.RunTimeProperties.CompressionThreshold = 256;
```

### Relationship between RunTimeProperties and Connection objects

The values set in a Connection object supersede the static RunTimeProperties values. This relationship is particularly important if a static RunTimeProperties object is updated after a Connection object has been instantiated. In this situation, the effect on a Connection object instance depends on whether that specific property was overridden in that object after it was instantiated, as described in the following table.

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

**Note:** The converse of this relationship is not true. That is, changing the value of a property for a Connection object does not affect the static value of that property, nor does it affect the value of that property in any other object instance.
The following example uses the `WaitIfBusy` property to illustrate the relationship between `RunTimeProperties` and `Connection` object properties:

```csharp
Connection conn;
bool ret;

RunTimeProperties.WaitIfBusy = true;
conn = new Connection(url, userid, password, appserverInfo);
ret = conn.GetBooleanProperty("PROGRESS.Session.WaitIfBusy");
System.Console.WriteLine("(A) WaitIfBusy= " + ret);

RunTimeProperties.WaitIfBusy = false; // change static property
ret = conn.WaitIfBusy; // equivalent accessor method
System.Console.WriteLine("(B) WaitIfBusy= " + ret);

conn.WaitIfBusy = true; // change conn object property
ret = conn.GetBooleanProperty("PROGRESS.Session.WaitIfBusy");
System.Console.WriteLine("(C) WaitIfBusy= " + ret);
RunTimeProperties.SetBooleanProperty("PROGRESS.Session.WaitIfBusy", false);
ret = conn.WaitIfBusy;
System.Console.WriteLine("(D) WaitIfBusy= " + ret);

RunTimeProperties.WaitIfBusy = true;
ret = conn.GetBooleanProperty("PROGRESS.Session.WaitIfBusy");
System.Console.WriteLine("(E) WaitIfBusy= " + ret);
```

When executed, this program prints the following result:

(A) WaitIfBusy= true  
(B) WaitIfBusy= false  
(C) WaitIfBusy= true  
(D) WaitIfBusy= true  
(E) WaitIfBusy= true

**Modifying an instantiated Connection object's properties**

If you change the value of a property in a `Connection` object after it has been instantiated, the `Connection` object might not recognize the change, depending on the session model as described in the following table.
### Table 25: Relationship between Connection object properties and session model

<table>
<thead>
<tr>
<th>If this Connection object property is updated . . .</th>
<th>Then the update is . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRESS.Session.AppServerKeepalive on page 108</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Recognized once pick list is refreshed</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Recognized once pick list is refreshed</td>
</tr>
<tr>
<td>PROGRESS.Session.ConnectionLifetime on page 110</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.ConnectionTimeout on page 110</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Recognized once pick list is refreshed</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Recognized once pick list is refreshed</td>
</tr>
<tr>
<td>PROGRESS.Session.IdleConnectionTimeout on page 111</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.InitialConnections on page 112</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.MaxConnections on page 113</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.MinConnections on page 113</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.NoHostVerify on page 113</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.NsClientMaxPort on page 115</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.NsClientMinPort on page 115</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td>PROGRESS.Session.NsClientPicklistExpiration on page 116</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Recognized when pick list is refreshed</td>
</tr>
<tr>
<td>PROGRESS.Session.NsClientPicklistSize on page 116</td>
<td><strong>Session-managed</strong></td>
</tr>
<tr>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td><strong>Session-free</strong></td>
</tr>
<tr>
<td></td>
<td>Recognized when pick list is refreshed</td>
</tr>
</tbody>
</table>
Alphabetical listing of properties

This section contains a reference to the following properties that you can set for a RuntimeProperties object or a Connection object, including the applicable session model, a complete description of usage, and the default value for each property.

**PROGRESS.Session.AppServerKeepalive**

**Data Type:**
String
Session model:

All

The AppServerKeepalive (ASK) property indicates whether or not the client will participate in a "keep alive" relationship with the AppServer. If configured for the AppServer, the Keepalive feature allows the AppServer to ping the client at set intervals to determine if the client is still viable. If configured for the client, the Keepalive feature allows the client to ping the AppServer at set intervals to determine if the AppServer is still viable. Valid values are:

• AllowServerASK
• DenyServerASK
• AllowClientASK
• DenyClientASK

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

Default:

AllowServerASK, DenyClientASK

PROGRESS.Session.CompressionEnabled

Data Type:

Logical

Session model:

Session-free and Session-managed

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

Default:

False

PROGRESS.Session.CompressionLevel

Data Type:

Integer

Session model:

Session-free and Session-managed

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

**Data Type:**
Integer

**Session model:**
Session-free and Session-managed

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

**Default:**
256

**PROGRESS.Session.ConnectionLifetime**

**Data Type:**
Integer

**Session model:**
Session-free only

The maximum number of seconds that a given connection can be used before it is destroyed. Connections whose lifetime exceeds the specified value are destroyed as they become available. (This might cause the number of connections in the pool to temporarily fall below the PROGRESS.Session.MinConnections on page 113 value.)

An available connection is one that is not currently reserved to run a request. Bound connections associated with remote persistent procedures are not available for reuse until the persistent procedure is deleted. As a result, bound connections remain available as long as necessary, even if they exceed the specified value.

**Default:**
300

**PROGRESS.Session.ConnectionTimeout**

**Data Type:**
Integer

**PROGRESS.Session.MaxConnections**

**Data Type:**
Integer

**Session model:**
Session-free and Session-managed

The maximum number of connections that can be active in a session at any one time. When the maximum number of connections is reached, a new connection is refused.

**Default:**
1000

**PROGRESS.Session.MinConnections**

**Data Type:**
Integer

**Session model:**
Session-free only

The minimum number of connections that can be active in a session at any one time. When the minimum number of connections is reached, a new connection is granted.

**Default:**
1
**Session model:**

Session-managed and Session-free only

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

**Default:**

0 (indefinite wait)

**PROGRESS.Session.EnableTracing**

**Data Type:**

Integer

**Session model:**

Session-free and session-managed

Determines whether tracing is on (1) or off (0). Tracing logs highlights of proxy execution, including data received by the client and data passed to the AppServer.

This property applies to the `RunTimeProperties` class only (it does not apply to `Connection` objects).

**Default:**

0

**PROGRESS.Session.IdleConnectionTimeout**

**Data Type:**

Integer

**Session model:**

Session-free only

The amount of time, in seconds, that the client waits before it attempts to shut down idle network connections to the AppServer, based on client demand. However, the number of connections does not fall below the `PROGRESS.Session.MinConnections` value on page 113.

If this value is zero, idle network connections remain connected.

**Default:**

0
**PROGRESS.Session.InitialConnections**

**Data Type:**
Integer

**Session model:**
Session-free only

The number of socket connections established when the AppObject is instantiated. If the specified number of connections cannot be created, the AppObject constructor fails and any connections that were successfully created are closed.

**Default:**
1

**PROGRESS.Session.LogFileName**

**Data Type:**
String

**Session model:**
Session-free and session-managed

The name of the log file to which trace messages are sent. If this property is not set, the tracing output is Standard Error.

This property applies to the RunTimeProperties class only (it does not apply to Connection objects).

**Default:**
None

**PROGRESS.Session.LoggingLevel**

**Data Type:**
Integer

**Session model:**
Session-free and session-managed

Determines the level of information logged for tracing. 0 is minimum logging; 7, maximum. The higher the value for this property, the greater the level of information. For more information about logging levels, see OpenEdge Development: Debugging and Troubleshooting. RunTimeProperties Connection

**Default:**
2
This property applies to the class only (it does not apply to objects).

**PROGRESS.Session.MaxConnections**

**Data Type:**
Integer

**Session model:**
Session-free only

The maximum number of connections that can be established for a given AppObject. The value must be greater than or equal to zero.

If this value is zero, there is no limit to the number of connections that can be created.

**Default:**
0

**PROGRESS.Session.MinConnections**

**Data Type:**
Integer

**Session model:**
Session-free only

The minimum number of connections that can be established for a given AppObject. The value should not be greater than the value of `PROGRESS.Session.MaxConnections` on page 113.

If the number of connections falls below the `PROGRESS.Session.MinConnections` on page 113 value, then sufficient new connections are created (when the next request is run) to bring the size of the connection pool back up to the `PROGRESS.Session.MinConnections` on page 113 value.

**Default:**
0

**PROGRESS.Session.NoHostVerify**

**Data Type:**
Boolean

**Session model:**
Session-free and session-managed
Controls domain name checking. If specified, turns off host verification for an SSL-enabled connection, either using HTTPS with the AIA or using a direct connection to an SSL-enabled AppServer. Without this property specified, the client compares the host name specified in the URL with the Common Name specified in the server certificate, and raises an error if they do not match. With this property specified, the client never raises the error. For more information on SSL-enabled connections, see the sections on PKI and SSL in OpenEdge Getting Started: Core Business Services - Security and Auditing.

Specifying FALSE enables domain name checking. Specifying TRUE disables domain name checking.

Default:

FALSE

PROGRESS.Session.NoSchemaMarshal

Data Type:

Boolean

Session model:

Session-free and session-managed

When set to true, this property prevents .NET from sending schema information to the AppServer for input-output and output DataSet and DataTable properties. The property applies to all DataSet and DataTable parameters for subsequent method calls. Set the property before a proxy method is called. Reset the NoSchemaMarshal property back to false for subsequent proxy method calls if needed.

This property must be used for output only DataSet or DataTable parameters, and not the ProDataTable.SetNoSchemaMarshal( ) method.

This is because the c# compiler considers an output parameter unassigned and it must be assigned before its value is used, therefore, the .NET proxy code is not able to look at the output DataSet or DataTable object to detect if SetNoSchemaMarshal( ) was called.

The property can be used in place of the ProDataTable.SetNoSchemaMarshal( ) method. Use this method if you need all DataSet and DataTable parameters in a single proxy method to not pass schema info from .NET to the AppServer.

For example:

```csharp
RunTimeProperties.NoSchemaMarshal = true;
appObj.getMyDSet(1, out myOutDS);
```

See SetNoSchemaMarshal( ) on page 89 for more details on disabling schema marshaling.

This property applies to the RunTimeProperties class only (it does not apply to Connection objects).

**Note:** This property affects schema marshaling from the .NET Open Client to the AppServer only. To alter marshaling from the AppServer to the client, you can use the -ttmarshal startup parameter, or you can use the temp table properties NO-SCHEMA-MARSHAL and MIN-SCHEMA-MARSHAL. For more information, see OpenEdge Deployment: Startup Command and Parameter Reference and OpenEdge Development: ABL Reference.
Default:
False

PROGRESS.Session.NoSslSessionReuse

Data Type:
Boolean

Session model:
Session-free and session-managed
If specified, the connection does not reuse the session IDs to shorten the overhead in reconnecting to the same SSL-enabled server (either a Web server with HTTPS or an SSL-enabled AppServer).

Default:
FALSE

PROGRESS.Session.NsClientMaxPort

Data Type:
Integer

Session model:
Session-free and session-managed
The minimum value for the UDP port number used by the client when communicating with the Name Server. If this value is zero, OpenEdge chooses the NameServer client port randomly.
This value should be greater than or equal to the value of the PROGRESS.Session.NsClientMinPort on page 115 property.

Default:
0

PROGRESS.Session.NsClientMinPort

Data Type:
Integer

Session model:
Session-free and session-managed
The minimum value for the UDP port number used by the client when communicating with the NameServer. If this value is zero, OpenEdge chooses the NameServer client port randomly.
PROGRESS.Session.NsClientPicklistExpiration

Data Type: Integer

Session model: Session-free only

The maximum amount of time, in seconds, that the client retains an AppServer pick list for an application service.

Default: 300

PROGRESS.Session.NsClientPicklistSize

Data Type: Integer

Session model: Session-free only

The number of AppServer picks to request from the NameServer each time it looks up the available AppServer connections for a given application service name.

Default: 8

PROGRESS.Session.NsClientPortRetry

Data Type: Integer

Session model: Session-free and session-managed

The maximum number of attempts that the client makes to get a valid local UDP port number when attempting to communicate with the NameServer.

Default: 0
**PROGRESS.Session.NsClientPortRetryInterval**

*Data Type:*  
Integer

*Session model:*  
Session-free and session-managed

The interval, in milliseconds, that the client waits between attempts to get a valid UDP port number when attempting to communicate with the NameServer.

*Default:*  
0

**PROGRESS.Session.ProxyHost**

*Data Type:*  
String

*Session model:*  
Session-free and session-managed

The name of the host or the IP address of the host at which the Proxy Web server is located. This value can be a string with either the DNS name of the Proxy Web server or its dot-formatted IP address.

When you specify this parameter, all connections made by an AppServer client using the HTTP or HTTPS protocol connect to an AppServer Internet Adapter (AIA) instance using the proxy server at the specified host and port.

*Note:* If you specify the **PROGRESS.Session.ProxyHost** on page 117 property, you must also specify the **PROGRESS.Session.ProxyPort** on page 118 property.

*Default:*  
None

**PROGRESS.Session.ProxyPassword**

*Data Type:*  
String

*Session model:*  
Session-free and session-managed

Use to authenticate the AppServer client to the Proxy Web server. The value can be a string of up to 512 printable ASCII characters.
You should use `PROGRESS.Session.ProxyPassword` on page 117 when `PROGRESS.Session.ProxyHost` on page 117 and `PROGRESS.Session.ProxyPort` on page 118 are specified and the Proxy server requires authentication.

If `PROGRESS.Session.ProxyUserId` on page 118 is not specified, the connection ignores any value for `PROGRESS.Session.ProxyPassword` on page 117.

**Default:**
None

**PROGRESS.Session.ProxyPort**

**Data Type:**
Integer

**Session model:**
Session-free and session-managed

When you specify this property, all connections made by an AppServer client using the HTTP or HTTPS protocol connect to an AppServer Internet Adapter (AIA) instance using the proxy server at the specified host and port.

**Note:** If you specify the `PROGRESS.Session.ProxyPort` on page 118 property, you must also specify the `PROGRESS.Session.ProxyHost` on page 117 property.

**Default:**
None

**PROGRESS.Session.ProxyUserId**

**Data Type:**
String

**Session model:**
Session-free and session-managed

Use to authenticate an AppServer client to the Proxy Web server. The user-id can be a string of up to 512 printable ASCII characters, including the space character.

**Default:**
None
PROGRESS.Session.RequestWaitTimeout

Data Type:
Integer

Session model:
Session-free only

The way requests are handled when no connections are available and the connection pool is full, specified by one of the values shown in the following table.

Table 26: requestWaitTimeout property values

<table>
<thead>
<tr>
<th>If the value is ...</th>
<th>The Open Client Runtime ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Queues the request indefinitely until an AppServer session becomes available.</td>
</tr>
<tr>
<td>0</td>
<td>Rejects the request and returns an error message to the client indicating that there are too many concurrent requests.</td>
</tr>
<tr>
<td>$N$</td>
<td>Queues the request the number of seconds specified by the value ($N$ must be greater than zero) until the AppServer becomes available. If no connection becomes available in that time, an error message is returned to the client.</td>
</tr>
</tbody>
</table>

Default:
-1

PROGRESS.Session.SessionModel

Data Type:
Integer

Session model:
Session-free and session-managed

Session model supported by the AppServer operating mode, specified by one of the values shown in the following table.

Table 27: sessionModel property values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Session-managed</td>
</tr>
<tr>
<td>1</td>
<td>Session-free</td>
</tr>
</tbody>
</table>

This property must be set for session-free application. It is optional for session-managed applications. This value must match the AppServer operating mode or the connection fails.
PROGRESS.Session.TcpKeepAliveInterval

**Data Type:**
Integer

**Session model:**
All

Indicates the interval the client waits before resending a TCP KeepAlive message to the AppServer. You can customize the time interval by any of the following methods:

- Setting the value of the property in the .NET application configuration file. For more information, see the section on checking for failed client connections section in *OpenEdge Application Server: Administration*.
- Passing the value as an integer parameter in the `setTcpKeepAliveProperty` method. If a user passes an invalid value for the parameter, `TcpKeepAliveInterval`, the system takes the default value. For more information, see the section on checking for failed client connections section in *OpenEdge Application Server: Administration*.

**Default:**
1000 ms

**Note:** For more information about the parameter, see Microsoft Windows documentation.

PROGRESS.Session.TcpKeepAliveTime

**Data Type:**
Integer

**Session model:**
All

Indicates the duration (in milliseconds), during which the TCP connection between a .NET client and its corresponding AppServer remains idle before the client sends a TCP KeepAlive message to the AppServer. You can customize the time duration by any of the following methods:

- Setting the value of the property in the .NET application configuration file. For more information, see the section on checking for failed client connections section in *OpenEdge Application Server: Administration*.
- Passing the value as an integer parameter in the `setTcpKeepAliveProperty` method. If a user passes an invalid value for the parameter, `TcpKeepAliveTime`, the system takes the default value. For more information, see the section on checking for failed client connections section in *OpenEdge Application Server: Administration*.

**Default:**
7,200,000 ms
**PROGRESS.Session.TcpMaxDataRetranmission**

**Data Type:**
Integer

**Session model:**
All

Indicates the maximum number of times a client is allowed to send TCP KeepAlive message to the AppServer, before the client hangs up the connection. **TcpMaxDataRetranmission** is a system property. The system administrator must configure this property in the system registry.

To edit the number of TCP KeepAlive retrials, you must set the **TcpMaxDataRetransmission** system property. Navigate in your registry to:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
```

and set **TcpMaxDataRetransmission**.

**Default:**
5

**PROGRESS.Session.WaitIfBusy**

**Data Type:**
Integer

**Session model:**
Session-managed only

An integer value that determines how to handle client requests to a service that is busy processing a prior request. If the value is 1, the Open Client Runtime queues multiple requests for this service and executes them one at a time until the queue is empty. If the value is 0 and the Open Client Runtime is executing a prior request for the service, each subsequent request for the same service fails until the client completes the request it is currently executing.

**Default:**
0
Handling Errors

.NET clients handle errors by catching exceptions. To enable .NET proxies to throw exceptions, OpenEdge establishes two exception class hierarchies:

- **General ABL exceptions** — A hierarchy of exception classes that extend the `Open4GLEException` class
- **DataSet/DataTable exceptions** — A `ProDataException` class that extends the standard `System.Data.DataException` class

The following section in this chapter describes general ABL exceptions in detail.

For details, see the following topics:

- **General ABL exceptions**

**General ABL exceptions**

This section presents the hierarchy of ABL exception classes and then describes each one.
Exception class hierarchy

The ABL exceptions classes are shown below, hierarchically:

```
System.Exception
 Progress.Open4GL.Exceptions.ProException
  Progress.Open4GL.Exceptions.Open4GLException
   Progress.Open4GL.Exceptions.SystemErrorException
   Progress.Open4GL.Exceptions.SchemaValidationException
   Progress.Open4GL.Exceptions.Runtime4GLException
    Progress.Open4GL.Exceptions.Runtime4GLErrorException
    Progress.Open4GL.Exceptions.Runtime4GLQuitException
    Progress.Open4GL.Exceptions.Runtime4GLStopException
   Progress.Open4GL.Exceptions.OutputSetException
   Progress.Open4GL.Exceptions.MetaDataException
   Progress.Open4GL.Exceptions.BusySessionException
  Progress.Open4GL.Exceptions.ConnectException
   Progress.Open4GL.Exceptions.HostUnknownException
   Progress.Open4GL.Exceptions.BadURLException
   Progress.Open4GL.Exceptions.BrokerIOException
   Progress.Open4GL.Exceptions.ConnectFailedException
   Progress.Open4GL.Exceptions.ConnectProtocolException
   Progress.Open4GL.Exceptions.InvalidNameServerPortException
   Progress.Open4GL.Exceptions.NameServerClientPortException
   Progress.Open4GL.Exceptions.NameServerClientPortRangeException
   Progress.Open4GL.Exceptions.NameServerClientPortRetryException
   Progress.Open4GL.Exceptions.NameServerCommunicationsException
   Progress.Open4GL.Exceptions.NameServerInterruptException
   Progress.Open4GL.Exceptions.NameServerMessageFormatException
   Progress.Open4GL.Exceptions.NoSuchAppServiceException
   Progress.Open4GL.Exceptions.UnknownHostnameException
```

Exception descriptions

The following table shows the general ABL exception classes, in alphabetical order.
Table 28: ABL exceptions for .NET proxies

<table>
<thead>
<tr>
<th>Exception class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadURLException</td>
<td>Invalid URL format in the connection parameter.</td>
</tr>
<tr>
<td>BrokerIOException</td>
<td>AppServer communications error. The message contains the specific reason for the failure.</td>
</tr>
<tr>
<td>BusySessionException</td>
<td>Attempt to use a session already in use by another client thread.</td>
</tr>
<tr>
<td>ConnectException</td>
<td>Error connecting to an AppServer. The base class of several specific exceptions.</td>
</tr>
<tr>
<td>ConnectFailedException</td>
<td>Failed AppServer connection attempt. The message contains the specific reason for the failure.</td>
</tr>
<tr>
<td>ConnectProtocolException</td>
<td>Attempt to use an unknown or unsupported protocol.</td>
</tr>
<tr>
<td>HostUnknownException</td>
<td>Invalid NameServer host specified in the URL.</td>
</tr>
<tr>
<td>InvalidNameServerPortException</td>
<td>Invalid NameServer port specified in the URL.</td>
</tr>
<tr>
<td>MetaDataException</td>
<td>An inconsistency was detected during meta data validation.</td>
</tr>
<tr>
<td>NameServerClientPortException</td>
<td>NameServer was not found at the specified port number.</td>
</tr>
<tr>
<td>NameServerClientPortRangeException</td>
<td>Specified NameServer port was outside of the acceptable port range (minimum: 1024, maximum: 65535).</td>
</tr>
<tr>
<td>NameServerClientPortRetryException</td>
<td>Attempt to contact NameServer exceeded the maximum retry count.</td>
</tr>
<tr>
<td>NameServerCommunicationsException</td>
<td>NameServer communications error. The message contains the specific reason for the failure.</td>
</tr>
<tr>
<td>NameServerInterruptException</td>
<td>NameServer communications broken. Either no response was received from the NameServer or the thread was interrupted before the response was received.</td>
</tr>
<tr>
<td></td>
<td>Invalid format for message received from NameServer.</td>
</tr>
<tr>
<td>Exception class</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NameServerMessageFormatException</td>
<td>Application Service specified in URL is unknown to the NameServer.</td>
</tr>
<tr>
<td>NoSuchAppServiceException</td>
<td>The root exception for all Open Client errors. Used for miscellaneous client-side errors.</td>
</tr>
<tr>
<td>Open4GLException</td>
<td>The base class for all customer-visible Open Client errors.</td>
</tr>
<tr>
<td>OutputSetException</td>
<td>Output DataSet/DataTable error; for example, trying to get a column value out of column order or to make an illegal type conversion.</td>
</tr>
<tr>
<td>ProException</td>
<td>ABL ERROR condition.</td>
</tr>
<tr>
<td>RunTime4GLErrorException</td>
<td>ABL run-time errors.</td>
</tr>
<tr>
<td>RunTime4GLException</td>
<td>ABL QUIT condition.</td>
</tr>
<tr>
<td>RunTime4GLQuitException</td>
<td>ABL STOP condition.</td>
</tr>
<tr>
<td>RunTime4GLStopException</td>
<td>An inconsistency was detected during schema validation.</td>
</tr>
<tr>
<td>SchemaValidationException</td>
<td>An unexpected error indicating an OpenEdge bug or an unusual environmental problem, such as running out of disk space.</td>
</tr>
<tr>
<td>SystemErrorException</td>
<td>AppService host name is unknown.</td>
</tr>
</tbody>
</table>
Using OpenEdge .NET Proxy Objects in Microsoft Visual Studio .NET

This chapter describes how to use Microsoft Visual Studio .NET to build .NET Open Client applications.

For details, see the following topics:

• Types of .NET open clients
• Getting started building a .NET Open Client application using Microsoft Visual Studio .NET
• Using the samples

Types of .NET open clients

The following types of .NET Open Clients can use proxies created by ProxyGen:

• Windows forms (GUI) on page 128
• Web forms (ASP.NET) on page 128
• .NET Web services (ASP Web services) on page 128
Windows forms (GUI)

The Windows forms model is the GUI development and deployment model for .NET applications. With Windows forms, you can build rich, interactive client interfaces that are deployed directly to client machines. Open Client proxy objects are accessible to Windows forms clients as native .NET objects, including support for advanced data binding.

To develop and deploy Windows forms that use Open Client proxy objects, the OpenEdge .NET Open Client Runtime assemblies (Progress.o4glrt.dll, Progress.Messages.dll, and Progress.ssl.dll) and the message resource files must be directly available to the client application. These files, along with your proxies, must be deployed on the client machine.

Web forms (ASP.NET)

The Web forms model is the active server page development and deployment model for .NET. Web forms are for browser-based .NET clients. In this model, an Internet Information Services (IIS) server is responsible for the generation and delivery of the browser-based client code. Your Open Client proxy objects fully support a Web forms model, as long as the Progress.o4glrt.dll, Progress.Messages.dll, and Progress.ssl.dll assemblies, the message resource files, and your proxies are accessible to the server-side code for your Web forms.

.NET Web services (ASP Web services)

The .NET Web service model is a similar deployment model to Web forms. Proxy objects can be integrated into your .NET classes, which then can be deployed as Web services. As with Web forms, your run-time assemblies and proxies must be available to the server-side code. While .NET Web services are possible with your .NET proxy objects, OpenEdge tools such as Web services tools provide a more seamless approach by providing native Web service capability directly to the OpenEdge AppServer.

Getting started building a .NET Open Client application using Microsoft Visual Studio .NET

This section describes how to build a .NET Open Client application using Microsoft Visual Studio .NET.

To begin building a client application that uses a .NET proxy:

1. Run Microsoft Visual Studio .NET
2. Create a new project
3. In your Solution Explorer window, add references to the following:
   - The proxy .dll file (created by ProxyGen)
   - The main OpenEdge .NET Open Client Runtime assembly (Progress.o4glrt.dll) in the ProxyGen output directory
Once you add a reference to the proxy .dll file, your new client application can access AppServer functionality through methods on the generated proxy objects. To view these proxy classes, bring up the Object Browser in Microsoft Visual Studio .NET. The Object Browser allows you to examine objects (namespaces, classes, and so on) and their members in your project.

Using the samples

This section demonstrates using one of the .NET Open Client samples available in the Documentation and Samples directory. This sample application (written in VB.NET) is in the following location:

```
Doc_and_Samples_Install\src\samples\open4gl\dotnet\OrderInfo\n```

For more information on accessing and installing these samples, see the information on sample .NET client applications in Configuring and Deploying .NET Open Client Applications on page 21.

In this directory, you can find the following files:

- `Demo.vb` — VB.NET client source code
- `OrderInfoDemo.sln` — .NET Visual Studio solution file
- `OrderInfoDemo.vbproj` — .NET Visual Studio project file
- `Readme.txt` — Instructions on using the sample

Open the solution file provided, `OrderInfoDemo.sln`. You will see the reference for `OrderInfo`, the sample's ProxyGen-generated proxy assembly (`OrderInfo.dll`). You can look at the `OrderInfo` reference in the Object Browser. Expand this assembly, and you will see the following namespaces:

- `OrderProxy`
- `OrderProxy.StrongTypesNS`

`OrderProxy` is the `General` namespace you entered in the .NET Client Details group of the ProxyGen Generate Proxies dialog box. Under the `OrderProxy` namespace, you will see the `OrderInfo` and `CustomerOrder` classes for the AppObject and a ProcObject, respectively. Double-click on each class, to see the methods and properties available to your client.

For example, under the `OrderInfo` class are the four `OrderInfo(...)` constructors your client can call to create an `OrderInfo` instance. These are shown with a method name of `New`. You also will see the `FindCustomerByName( )` method, which maps to its corresponding remote ABL method.

For proxies that contain methods with a static temp-table or static DataSet parameter, ProxyGen creates a namespace called `namespace.StrongTypesNS`, which contains the corresponding strongly typed DataTable and DataSet classes for that object. Here, `namespace` is the `General` namespace entered in the .NET Client Details group of the ProxyGen Generate Proxies dialog box, or it may be omitted. In this example, the `namespace` is `OrderProxy.StrongTypesNS`.

If you expand the `OrderProxy.StrongTypesNS` namespace, you will see, for example, the `OrderDetailsDataTable` class. This is the strongly typed DataTable class created by ProxyGen for the static temp-table definition `OrderDetails`. You use this class in your client application when calling the corresponding AppServer method that has the static temp-table parameter. You can expand the `OrderProxy.CustomerOrder` class and examine the signature of the `GetOrderDetails(..) and UpdateOrderDetails(..)` methods. Both these methods contain parameters of the `OrderProxy.StrongTypesNS.OrderDetailsDataTable` class.
Microsoft Visual Studio .NET IntelliSense provides easy access to these objects when you write your client application in the text editor. If you type the namespace of the class (for example OrderProxy.), the following list of objects (in this example, in the OrderProxy namespace) is displayed:

- OrderInfo (class)
- CustomerOrder (class)
- StrongTypesNS (nested namespace)

You can select the desired class or member and easily add it to your code.

Open the client source file, Demo.vb. This is the sample client code that accesses the proxy classes. First, notice that the client code creates variables for each of the proxy objects, as shown:

```
' Declare the proxy objects
Dim m_conn As Connection = Nothing
Dim m_order As OrderInfo = Nothing
Dim m_custOrder As CustomerOrder = Nothing
Dim m_custOrderDetails As StrongTypesNS.OrderDetailsDataTable = Nothing
```

In the subroutine for the Connect button click, btnConnect_Click, there is the code to establish the connection to the AppServer. This code creates a connection object and then passes that object into the constructor for the AppObject, OrderInfo. This code is written within a try...catch block, because a connection failure will be returned as an exception from the Open Client Runtime, as shown:

```
Try
' Instantiate the proxy objects and connect
    m_conn = New Connection(txtURL.Text, "", "", "")
    m_order = New OrderInfo(m_conn)
    ...
    ...
Catch conEx As Exception
    ...
    ...
End Try
```
In the subroutine for the **Search** button, `btnFindByNum_Click`, you will find the code to run the non-persistent procedure `FindCustomerByNum`. This runs the procedure on the AppServer using the connection already established. Again the proxy call is written inside a `try...catch` block, as shown in the following code:

```vbnet
' This finds a Customer Name from the Customer Number
Dim custNum As IntHolder = New IntHolder()
Dim custName As String = Nothing

Try
' Send a request to the AppServer to get the Customer Name
m_custNum = CInt(txtCustNum.Text)
custNum.Value = m_custNum
m_order.FindCustomerByNum(custNum, custName)
.
.
Catch procEx As Exception
.
.
End Try
```

Now the client code executes a persistent procedure on the AppServer, passing it the customer number obtained from the call to the non-persistent procedure above. After the code creates the `ProcObject` for the persistent procedure, it calls a method on the `ProcObject`, `GetOrderDetails`, which executes the corresponding internal procedure in the persistent procedure on the AppServer. In both cases, the code is written inside a `try...catch` block, as shown:

```vbnet
' Run CustomerOrder.p on the AppServer
m_custOrder = m_order.CreatePO_CustomerOrder(custNum)
m_custOrder.GetOrderDetails(m_custOrderDetails)
.
.
```

Finally, the client code must disconnect from the AppServer. This code can be found in the subroutine for the **Disconnect** button, `btnDisconnect_Click`. Again the proxy calls are written inside a `try...catch` block, as shown in the following code:

```vbnet
Try
' Tell the AppServer to release the Persistent Proc
If Not (m_custOrder Is Nothing) Then
    m_custOrder.Dispose()
End If
m_order.Dispose()
.
.
Catch relEx As Exception
.
.
End Try
```
Using the Open Client .NET OpenAPI to Directly Access the AppServer

The Open Client .NET OpenAPI is a generic set of .NET classes that can be used instead of the generated .NET proxies produced by ProxyGen. Without proxies, you must do all of the setup work in your client code to access an application service on the AppServer. You must know the procedure names, including relative path information (for the equivalent ABL \texttt{RUN} statement), the number of parameters along with their type, and the return type for user-defined functions. When using the OpenAPI, there is no compile time checking so all function prototype error checking is done at run time.

For details, see the following topics:

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

Overview of .NET OpenAPI classes

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

---

OpenEdge Development: .NET Open Clients
**Table 29: .NET OpenAPI Classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress.Open4GL.Proxy.OpenAppObject</td>
<td>For accessing external ABL procedures in an application service using the OpenAPI</td>
</tr>
<tr>
<td>Progress.Open4GL.Proxy.OpenProcObject</td>
<td>For accessing an ABL persistent, single-run, or singleton procedure using the OpenAPI</td>
</tr>
<tr>
<td>Progress.Open4GL.Proxy.ParamArray</td>
<td>An object containing an array for passing parameters to an ABL procedure or user-defined function</td>
</tr>
<tr>
<td>Progress.Open4GL.Proxy.ParamArrayMode</td>
<td>Constants for specifying the mode of an ABL procedure or user-defined function parameter (INPUT, INPUT-OUTPUT, or OUTPUT)</td>
</tr>
<tr>
<td>Progress.Open4GL.Parameter</td>
<td>Constants for specifying the ABL data type of a procedure or user-defined function parameter, return type, or temp-table field</td>
</tr>
<tr>
<td>Progress.Open4gl.ProcedureType</td>
<td>A .NET enum for specifying a procedure type as Persistent, SingleRun, or Singleton</td>
</tr>
<tr>
<td>Progress.Open4GL.Proxy.ProDataRelationMetaData</td>
<td>For defining a ProDataSet data-relation</td>
</tr>
<tr>
<td>Progress.Open4GL.Proxy.ProDataSetMetaData</td>
<td>For defining the schema of an ABL ProDataSet mapped to an ADO.NET DataSet</td>
</tr>
<tr>
<td>Progress.Open4GL.Proxy.TempTableMetaData</td>
<td>For defining the schema of an ABL temp-table mapped to an ADO.NET DataTable</td>
</tr>
</tbody>
</table>

This chapter describes how to use these classes to access an AppServer using the .NET OpenAPI.

**Connecting to the AppServer**

To establish a connection to an AppServer using the .NET OpenAPI, you must instantiate a Progress.Open4GL.Proxy.OpenAppObject. You can create an OpenAppObject using one of the following constructors:
Syntax

```java
public OpenAppObject(Connection connectObj, string appservice)
public OpenAppObject(string url, string userid, string password,
                        string appserver-info, string appservice)
public OpenAppObject(string userid, string password,
                        string appserver-info, string appservice)
public OpenAppObject(string appservice)
```

**connectObj**

Specifies a `Progress.Open4GL.Proxy.Connection` object, which defines connection parameters to access the AppServer. For more information on this object, see Connecting to an AppServer on page 39.

**appservice**

Specifies the name of the application service.

**url**

Specifies the URL to connect to an AppServer URL. Defaults to `AppServer://localhost:5162/appService`. For more information on forming this URL, see the sections on connecting to the AppServer using a URL in OpenEdge Application Server: Developing AppServer Applications.

**Note:** For an HTTPS connection using the AppServer Internet Adapter (AIA) with the Web server running on the .NET Open Client machine, you must specify the Internet host name (`not "localhost"`) in the URL exactly as it appears in the CN field of the Web server certificate.

**userid**

Specifies a user ID required to access the AppServer.

**password**

Specifies a password required to access the AppServer.

**appserver-info**

Specifies information required by the application service at connection time.

You can set a number of session and run-time properties to manage access to an application service. For example, to access a session-free AppServer using the .NET OpenAPI, you must set the proxy property `PROGRESS.Session.SessionModel` to 1. For more information on setting this and other session properties see Accessing Proxy Properties on page 93.
Running procedures and user-defined functions

After connecting to an AppServer and creating a `Progress.Open4GL.Proxy.OpenAppObject`, you can invoke application service code as follows:

1. Run any external non-persistent, persistent, single-run, or singleton procedure that is available on the `OpenAppObject`. (Running a persistent, single-run, or singleton procedure instantiates an `OpenProcObject`.)
2. Run any available internal procedure or user-defined function on an instantiated `OpenProcObject`.

Before running any procedure or user-defined function, you must set up any parameters and return types required by the procedure or user-defined function. For more information, see Setting up parameters and return types on page 138. You can then pass the parameters to the procedure or user-defined function and manage them according to their modes (INPUT, INPUT-OUTPUT or OUTPUT). For more information, see Passing parameters on page 164.

After running any procedure or user-defined function, you must handle any output (returned values) or error conditions. For more information, see Handling returned values on page 167.

The following sections describe:

- Running a non-persistent procedure on an `OpenAppObject` on page 136
- Running a persistent procedure (`OpenProcObject`) on an `OpenAppObject` on page 137
- Running a single-run or singleton procedure (`OpenProcObject`) on an `OpenAppObject` on page 137
- Running an internal procedure or user-defined function on an `OpenProcObject` on page 138

For examples that show how to run application service code, see Sample .NET OpenAPI code on page 169.

Running a non-persistent procedure on an `OpenAppObject`

Once you create your `Progress.Open4GL.Proxy.OpenAppObject` instance, you can run any non-persistent procedure on the connected AppServer using the following method on the `OpenAppObject`:

**Syntax**

```csharp
public void RunProc(string procName, ParamArray paramArray)
```

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

- **paramArray**
  Specifies a `Progress.Open4GL.Proxy.ParamArray` object that holds the procedure parameters.
  For more information, see Setting up a parameter array on page 139.
Running a persistent procedure (OpenProcObject) on an OpenAppObject

Once you create your Progress.Open4GL.Proxy.OpenAppObject instance, you can run any persistent procedure on the connected AppServer using the following method on the OpenAppObject:

**Syntax**

```csharp
public OpenProcObject CreatePO(string procName, ParamArray paramArray)
```

*procName*

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

*paramArray*

Specifies a Progress.Open4GL.Proxy.ParamArray object that holds the procedure parameters. For more information, see Setting up a parameter array on page 139. A procedure with no parameters in the main block can also be run as persistent using an alternative syntax (see Running a single-run or singleton procedure (OpenProcObject) on an OpenAppObject on page 137), or the null value can be passed through paramArray.

This method returns a Progress.Open4GL.Proxy.OpenProcObject, which you will use to run internal procedures and user-defined functions provided by the persistent procedure.

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

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

**Syntax**

```csharp
public OpenProcObject CreatePO(String procName, ProcedureType procType)
```

*procName*

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

*procType*

A .NET enum type that specifies the procedure as persistent (ProcedureType.Persistent), single-run (ProcedureType.SingleRun), or singleton (ProcedureType.Singleton).

This method returns a Progress.open4gl.Proxy.OpenProcObject, which you can use to run internal procedures and user-defined functions provided by the procedure.
Note: The main block of a singleton or single-run procedure cannot have any parameters, so there is no `paramArray` available with this syntax. Although persistent procedures can have parameters in general, only persistent procedures without parameters can be run using this syntax.

The `OpenProcObject` also contains the public property `ProcedureType`, which holds a value of `Persistent`, `SingleRun`, or `Singleton`, depending on the procedure type of the object:

Running an internal procedure or user-defined function on an `OpenProcObject`

Once you obtain your `Progress.Open4GL.Proxy.OpenProcObject` instance, you can run any internal procedure or user-defined function defined in the object. However, if you are calling a user-defined function, you must identify the return type of the function before calling it. For more information, see Defining the return type for a user-defined function on page 160. For information on accessing the return value after calling the function, see Accessing user-defined function return values on page 168.

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

**Syntax**

```
public void RunProc(string procName, ParamArray paramArray)
```

`procName`

The name of an internal procedure or user-defined function provided by the `OpenProcObject`.

`paramArray`

Specifies a `Progress.Open4GL.Proxy.ParamArray` that holds the parameters for the internal procedure or user-defined function. For more information, see Setting up a parameter array on page 139.

Setting up parameters and return types

Before running an application service procedure or user-defined function, it might require parameters that you must pass. Before passing the parameters, you must set them up in two basic steps using the .NET OpenAPI.

1. Create a variable for each parameter.
2. Add each variable to a parameter array as required for the procedure or function you are calling.

The following sections describe how to complete these steps:

- Creating variables for parameters on page 139.
- Setting up a parameter array on page 139.
- Adding parameters using data type-specific methods on page 140.
Creating variables for parameters

Before passing a parameter, you must create a variable of the correct .NET data type. If you want to pass the ABL Unknown value (?) (null in .NET) for .NET value types (System.Boolean, System.Decimal, System.Int32, System.Int64, and System.DateTime), you must use the corresponding holder object. The ABL to .NET mappings are the same as the ones defined for passing parameters to Open Client proxy methods. For more information on the supported mappings between ABL and .NET data types, see Passing Parameters on page 45.

Setting up a parameter array

You must place all parameters for a procedure (external, persistent, and internal) or user-defined function into a Progress.Open4GL.Proxy.ParamArray object before running the procedure or user-defined function.

Creating a parameter array

You can create a ParamArray using the following constructor:

Syntax

```csharp
public ParamArray(int numParams)
```

`numParams`

Specifies the total number of parameters (including all INPUT, INPUT-OUTPUT, and OUTPUT parameters).

You can add each parameter to the ParamArray using a set of methods that add a parameter based on the data type of the parameter or a single generic add parameter method. You must identify the parameter position as well as the mode of the parameter. For temp-tables and ProDataSets, you must identify the meta data. You must also pass in an initial value for input and input-output parameters.

If you want to reuse the ParamArray for another call:

- The number of parameters for the next call must match the number of parameters in the ParamArray object
• You must clear any current values in the ParamArray using the Clear( ) method, as shown:

    // Create the ParamArray
    ParamArray parms = new ParamArray(1);
    // Clear for reuse
    parms.Clear( );

Adding parameters using data type-specific methods

You can set each parameter using one of the methods on the Progress.Open4GL.Proxy.ParamArray class that is specified for the corresponding ABL data type. Data types that are mapped to supported value types in .NET (System.Boolean, System.Decimal, System.Int32, System.Int64, and System.DateTime) have the following three sets of methods:

• One for the value type, which cannot be set to the ABL Unknown value (?) (null in .NET)
• One for a corresponding .NET nullable value type that can accept the ABL Unknown value (?).
• One for a corresponding OpenEdge type-holder class in the Progress.Open4GL namespace that can accept the ABL Unknown value (?) (supported for backward compatibility only).

Methods to add arrays of each data type are also provided.

For more information on the use of .NET nullable types and OpenEdge type-holder classes, see Unknown value (?) as a parameter on page 50.

General syntax for add parameter methods

The general syntax of these methods is as follows:

Table 30: Syntax

```
public void AddProgressType(int position, DataType value, int mode, [
    int extentValue | MetaType metaData]

throws Open4GLException
```

ProgressType

Indicates the ABL data type of the parameter, such as Decimal in AddDecimal( ), for adding a parameter of data type DECIMAL, or DecimalArray in AddDecimalArray( ), for adding a parameter of data type DECIMAL EXTENT.

position

Specifies a 0-based index indicating the parameter position.
**DataType**

Specifies the .NET data type of the parameter, for example, Decimal, Decimal?, Progress.Open4GL.DecimalHolder, Decimal[], Decimal?[], or Progress.Open4GL.DecimalHolder[]. In the syntax that follows, for any DataType not qualified by a namespace, System is assumed.

**value**

Specifies a variable that contains the value of the parameter, or null for an OUTPUT parameter.

**mode**

Specifies the ABL mode (passing direction) of the parameter. This can be one of the following constant values:

- ParamArrayMode.INPUT
- ParamArrayMode.INPUT_OUTPUT
- ParamArrayMode.OUTPUT

**extentValue**

Specifies the extent of an array for methods that add an array parameter (where ProgressType is appended with Array, as in AddDecimalArray()).

**MetaType**

Specifies one of the following classes for methods that add a temp-table or ProDataSet parameter:

- **TempTableMetaData** — Provides the schema information for temp-table parameters or temp-tables in a ProDataSet. For more information on defining this schema, see Defining the schema for a temp-table parameter on page 153.

- **ProDataSetMetaData** — Provides the schema information for ProDataSet parameters. For information on defining the schema for ProDataSet parameters, see Defining the schema for a ProDataSet parameter on page 157.

**metaData**

Specifies an instance of MetaType. When mode is ParamArrayMode.OUTPUT and ProgressType is DatasetHandle or TableHandle (passing a dynamic ProDataSet or temp-table parameter), you can set this parameter to null.

**Syntax summaries of all data type-specific add parameter methods**

The following sections list syntax summaries for all of the data type-specific methods for adding parameters to a ParamArray object:

- **CHARACTER** on page 142
- **COM-HANDLE** on page 142
- **DATASET** and **DATASET-HANDLE** on page 143
- **DATE** on page 143
- **DATETIME** on page 143
• DATETIME-TZ on page 144
• DECIMAL on page 144
• HANDLE on page 145
• INTEGER on page 146
• LOGICAL on page 146
• LONGCHAR on page 147
• MEMPTR on page 147
• RAW on page 148
• RECID on page 148
• ROWID on page 148
• TABLE and TABLE-HANDLE on page 149

CHARACTER

Syntax
The following are the data type-specific methods for adding an ABL CHARACTER parameter:

```java
public void AddCharacter(int position, String value, int mode)
public void AddCharacterArray(int position, String[] value, int mode, int extentValue)
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

COM-HANDLE

The following are the data type-specific methods for adding an ABL COM-HANDLE parameter:

Syntax

```java
public void AddCOMHandle(int position,
    Progress.Open4GL.COMHandle value, int mode)
public void AddCOMHandleArray(int position,
    Progress.Open4GL.COMHandle[] value, int mode,
    int extentValue)
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.
**DATASET and DATASET-HANDLE**

The following are the data type-specific methods for adding an ABL ProDataSet ([DATASET or DATASET-HANDLE](#)) parameter:

**Syntax**

```csharp
public void AddDataset(int position, System.Data DataSet value, int mode,
                        ProDataSetMetaData metaData)
public void AddDatasetHandle(int position, System.Data DataSet value,
                                int mode, ProDataSetMetaData metaData)
```

For information on the parameters to these methods, see [General syntax for add parameter methods](#) on page 140.

**DATE**

The following are the data type-specific methods for adding an ABL [DATE](#) parameter:

**Syntax**

```csharp
public void AddDate(int position, DateTime value, int mode)
public void AddDate(int position, DateTime? value, int mode)
public void AddDate(int position,
                    Progress.Open4GL.DateHolder value, int mode)
public void AddDateArray(int position, DateTime[] value, int mode,
                         int extentValue)
public void AddDateArray(int position,
                         DateTime?[] value, int mode, int extentValue)
public void AddDateArray(int position,
                         Progress.Open4GL DateHolder[] value, int mode,
                         int extentValue)
```

For information on the parameters to these methods, see [General syntax for add parameter methods](#) on page 140.

**DATETIME**

The following are the data type-specific methods for adding an ABL [DATETIME](#) parameter:
Syntax

public void AddDateTime(int position, DateTime value, int mode)
public void AddDateTime(int position, DateTime? value, int mode)
public void AddDateTime(int position, Progress.Open4GL.DateHolder value, int mode)
public void AddDateTimeArray(int position, DateTime[] value, int mode, int extentValue)
public void AddDateTimeArray(int position, DateTime?[] value, int mode, int extentValue)
public void AddDateTimeArray(int position, Progress.Open4GL.DateHolder[] value, int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

**DATETIME-TZ**

The following are the data type-specific methods for adding an ABL **DATETIME-TZ** parameter:

Syntax

public void AddDateTimeTZ(int position, DateTime value, int mode)
public void AddDateTimeTZ(int position, DateTime? value, int mode)
public void AddDateTimeTZ(int position, Progress.Open4GL.DateHolder value, int mode)
public void AddDateTimeTZArray(int position, DateTime[] value, int mode, int extentValue)
public void AddDateTimeTZArray(int position, DateTime?[] value, int mode, int extentValue)
public void AddDateTimeTZArray(int position, Progress.Open4GL.DateHolder[] value, int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

**DECIMAL**

The following are the data type-specific methods for adding an ABL **DECIMAL** parameter:
Syntax

public void AddDecimal(int position, Decimal value, int mode)
public void AddDecimal(int position, Decimal? value, int mode)
public void AddDecimal(int position,
                       Progress.Open4GL.DecimalHolder value, int mode)
public void AddDecimalArray(int position, Decimal[] value, int mode,
                             int extentValue)
public void AddDecimalArray(int position,
                             Decimal?[] value, int mode,
                             int extentValue)
public void AddDecimalArray(int position,
                             Progress.Open4GL.DecimalHolder[] value,
                             int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

HANDLE

The following are the data type-specific methods for adding an ABL HANDLE parameter:

Syntax

public void AddHandle(int position, Progress.Open4GL.Handle value, int mode)
public void AddHandleArray(int position, Progress.Open4GL.Handle[] value,
                            int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

INT64

The following are the data type-specific methods for adding an ABL INT64 parameter:
Syntax

public void AddInt64(int position, Int64 value, int mode)
public void AddInt64(int position, Int64? value, int mode)
public void AddInt64(int position, Progress.Open4GL.LongHolder value, int mode)
public void AddInt64Array(int position, Int64[] value, int mode, int extentValue)
public void AddInt64Array(int position, Int64?[ ] value, int mode, int extentValue)
public void AddInt64Array(int position, Progress.Open4GL.LongHolder[ ] value, int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

INTEGER

The following are the data type-specific methods for adding an ABL INTEGER parameter:

Syntax

public void AddInteger(int position, Int32 value, int mode)
public void AddInteger(int position, Int32? value, int mode)
public void AddInteger(int position, Progress.Open4GL.IntHolder value, int mode)
public void AddIntegerArray(int position, Int32[] value, int mode, int extentValue)
public void AddIntegerArray(int position, Int32?[ ] value, int mode, int extentValue)
public void AddIntegerArray(int position, Progress.Open4GL.IntHolder[ ] value, int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

LOGICAL

The following are the data type-specific methods for adding an ABL LOGICAL parameter:
For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

### LONGCHAR

The following are the data type-specific methods for adding an ABL LONGCHAR parameter:

**Syntax**

- public void AddLongChar(int position, String value, int mode)
- public void AddLongCharArray(int position, String[] value, int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

### MEMPTR

The following are the data type-specific methods for adding an ABL MEMPTR parameter:

**Syntax**

- public void AddMemptr(int position, Progress.Open4GL.Memptr value, int mode)
- public void AddMemptrArray(int position, Progress.Open4GL.Memptr[] value, int mode, int extentValue)

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.
RAW

The following are the data type-specific methods for adding an ABL RAW parameter:

Syntax

```java
public void AddRaw(int position, Byte[] value, int mode)
public void AddRawArray(int position, Byte[][] value, int mode, int extentValue)
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

RECID

The following are the data type-specific methods for adding an ABL RECID parameter:

Syntax

```java
public void AddRecid(int position, Int64 value, int mode)
public void AddRecid(int position, Int64? value, int mode)
public void AddRecid(int position, Progress.Open4GL.LongHolder value, int mode)
public void AddRecidArray(int position, Int64[] value, int mode, int extentValue)
public void AddRecidArray(int position, Int64?[] value, int mode, int extentValue)
public void AddRecidArray(int position, Progress.Open4GL.LongHolder[] value, int mode, int extentValue)
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

ROWID

The following are the data type-specific methods for adding an ABL ROWID parameter:
Syntax

```java
public void AddRowid(int position, Progress.Open4GL.Rowid value, int mode)
public void AddRowidArray(int position, Progress.Open4GL.Rowid[] value, int mode, int extentValue)
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

**TABLE and TABLE-HANDLE**

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

Syntax

```java
public void AddTable(int position, System.Data.DataTable value, int mode, TempTableMetaData metaData)
public void AddTableHandle(int position, System.Data.DataTable value, int mode, TempTableMetaData metaData)
```

For information on the parameters to these methods, see General syntax for add parameter methods on page 140.

**Adding parameters using a generic method**

You can use the following generic method on the `Progress.Open4GL.Proxy.ParamArray` class to set the parameters in the `ParamArray`:

Syntax

```java
public void AddParameter(Int32 position, Object value, Int32 mode, Int32 proType, Int32 extentValue, Progress.Open4GL.DynamicAPI.MetaDataBase metaData)
```

**position**

Specifies a 0-based index indicating the parameter position.

**value**

Specifies a variable of the appropriate `Object` data type that contains the value for the parameter, or `null` for an `OUTPUT` parameter.
mode

Specifies the ABL mode (passing direction) of the parameter. This can be one of the following constant values:

- ParamArrayMode.INPUT
- ParamArrayMode.INPUT_OUTPUT
- ParamArrayMode.OUTPUT

proType

Specifies the ABL data type of the parameter indicated by a class constant defined in the Progress.Open4GL.Parameter class, as shown in the following table.
<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Progress.Open4GL.Parameter class constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>PRO_CHARACTER</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>PRO_COMHANDLE</td>
</tr>
<tr>
<td>DATASET</td>
<td>PRO_DATASET</td>
</tr>
<tr>
<td>DATASET-HANDLE</td>
<td>PRO_DATASETHANDLE</td>
</tr>
<tr>
<td>DATE</td>
<td>PRO_DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>PRO_DATETIME</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>PRO_DATETIMETZ</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PRO_DECIMAL</td>
</tr>
<tr>
<td>HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
<tr>
<td>INT64</td>
<td>PRO_INT64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>PRO_INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>PRO_LOGICAL</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>PRO_LONGCHAR</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>PRO_MEMPTR</td>
</tr>
</tbody>
</table>
### Table: ABL data type vs Progress.Open4GL.Parameterclass constant

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Progress.Open4GL.Parameterclass constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>PRO_RAW</td>
</tr>
<tr>
<td>RECID</td>
<td>PRO_RECID</td>
</tr>
<tr>
<td>ROWID</td>
<td>PRO_ROWID</td>
</tr>
<tr>
<td>TEMP-TABLE</td>
<td>PRO_TEMPTABLE</td>
</tr>
<tr>
<td>TABLE-HANDLE</td>
<td>PRO_TABLEHANDLE</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
</tbody>
</table>

**extentValue**

Specifies the extent of an array, 0 or 1 for a scalar value.

**metaData**

Specifies the schema for ProDataSet or temp-table parameters, where `proType` is one of the following:

- `Parameter.PRO_DATASET`
- `Parameter.PRO_DATASETHANDLE`
- `Parameter.PRO_TEMPTABLE`
- `Parameter.PRO_TABLEHANDLE`

The value can be specified using one of the following classes:

- **TempTableMetaData** — Defines the schema information for temp-table parameters or temp-tables in a ProDataSet. For more information on defining this schema, see Defining the schema for a temp-table parameter on page 153.
- **ProDataSetMetaData** — Defines the schema information for ProDataSet parameters. For more information on defining the schema for ProDataSet parameters, see Defining the schema for a ProDataSet parameter on page 157.

When `mode` is `ParamArrayMode.OUTPUT` and `proType` is `Parameter.PRO_DATASETHANDLE` or `Parameter.PRO_TABLEHANDLE` (passing a dynamic ProDataSet or temp-table parameter), you can set this parameter to null.
Defining the schema for a temp-table parameter

Defining the schema for an ABL temp-table (TABLE or TABLE-HANDLE) parameter is a multi-step process.

To define the schema for a temp-table parameter:

1. If the parameter is for input or input-output, define an ADO.NET DataTable to hold the parameter value.
2. Define a TempTableMetaData object to hold the parameter schema.
3. Add field descriptions to the TempTableMetaData object.
4. Add the TempTableMetaData object and the DataTable (if applicable) as a temp-table parameter to your ParamArray object using the appropriate set parameter method.

Defining a TempTableMetaData object

For each temp-table (DataTable), you must define a Progress.Open4GL.Proxy.TempTableMetaData object to hold the schema using the following constructor:

Syntax

```csharp
public TempTableMetaData(string dataTableName, string strongTypeName, int numFields, bool bimageFlag, int numIndexes, string multiIxCols, string XMLNamespace, string XMLPrefix)
```

dataTableName

Specifies a name for the specified DataTable. This name is typically identical to any ABL temp-table to which this collection is mapped.

strongTypeName

Specifies the type name for a strongly-typed ADO.NET DataTable, or null.

numFields

Specifies the number of fields in the specified DataTable.

bimageFlag

Specifies true if the corresponding ABL temp-table is defined with the BEFORE-TABLE option, indicating that the temp-table (and hence the DataTable) can be modified. Otherwise, this value must be false and you cannot pass modified data between the Open Client and the AppServer.

numIndexes

Specifies the number of indexes on the table.
multiIdxCols

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

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

primeUniqueFlag

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

primeFld1 [,primeFln]...

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

primeIdxName

Specifies the primary index name.

uniqueIdxfld1 [,uniqueIdxfldn]...

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

uniqueIdxName

Specifies a unique secondary index name.

Thus, a table can have no indexes, a single primary index followed by zero or more secondary unique indexes, or it can have a single secondary unique index followed by zero or more additional secondary unique indexes. See the following examples:

- **Table with a single primary index:**
  
  "1, custNum:CustIndex"

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

- **Table with a single secondary index:**
  
  "orderDate:OrderIndex"

  This is a secondary unique index named OrderIndex with one column named orderDate.

- **Table with two indexes:**
  
  "1,custNum:CustIndex.orderDate,shipDate:OrderIndex"
These two indexes include:

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

Table with three indexes:

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

These three indexes include:

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

**XMLNamespace**

Specifies the namespace for XML serialization or null.

**XMLPrefix**

Specifies the prefix for XML serialization or null.

**Adding field descriptions to the TempTableMetaData object**

Add the meta data for each field of the temp-table by calling the `SetFieldMetaData()` method on the TempTableMetaData object for each column in the temp-table:

**Syntax**

```java
public void SetFieldMetaData(int fieldNumber, string fieldName,
int extentValue, int proType,
int userOrder, int xmlMapping)
```

**fieldNumber**

Specifies a 1-based position that corresponds to the position of a mapped field in an ABL temp-table.

**fieldName**

Specifies a name that is typically identical to a mapped field in the corresponding ABL temp-table. The value cannot be null and must be unique among fields (column properties) in the specified ProDataObject type.
extentValue

Specifies if and how the field represents an array field in the corresponding temp-table. The value must be 0 or greater. If the value is greater than 1, this column property is many-valued (represents an array field) and the value is its extent. If the property represents a BLOB or CLOB field, the value must be 0 or 1.

proType

Specifies the value of a class constant defined in the Progress.Open4GL.Parameter class. The specified class constant indicates the ABL data type of the mapped temp-table field. For more information on these class constants, see the sections on specifying the ABL temp-table field data types for the ProDataTable class in Passing Parameters on page 45. To identify the Java data type that the column property assumes for the specified ABL data type, see the information on data type mapping for temp-table fields in Passing Parameters on page 45.

userOrder

Specifies a 0-based user order position for the column property.

xmlMapping

Reserved for future use. Always specify 0.

Adding the TempTableMetaData with its DataTable as a temp-table parameter

Add the TempTableMetaData object together with ADO.NET DataTable it describes by passing them as parameters of the AddTable() or AddTableHandle() method that you use to add the temp-table parameter to the ParamArray object. For more information, see TABLE and TABLE-HANDLE on page 149.

The following example adds a temp-table parameter defined with no indexes.
Sample OpenAPI fragment adding a temp-table parameter

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

// Create Data table
DataTable table = new DataTable();
...

// Set up metadata
TempTableMetaData metaData1 = new TempTableMetaData
   ("OrderDetails",
    "OrderProxy.StrongTypesNS.OrderDetailsDataTable",
    6, false, 0, null, null,null);

metaData1.SetFieldMetaData
   (1, "OrderNum", 0, Parameter.PRO_INTEGER, 0, 0);
metaData1.SetFieldMetaData
   (2, "SalesRep", 0, Parameter.PRO_CHARACTER 1, 0);
metaData1.SetFieldMetaData
   (3, "OrderDate", 0, Parameter.PRO_DATE 2, 0);
metaData1.SetFieldMetaData
   (4, "ShipDate", 0, Parameter.PRO_DATE 3, 0);
metaData1.SetFieldMetaData
   (5, "TotalDollars", 0, Parameter.PRO_DECIMAL 4, 0);
metaData1.SetFieldMetaData
   (6, "OrderStatus", 0, Parameter.PRO_CHARACTER 5, 0);
parms.AddTable(0, table, ParamArrayMode.INPUT, metaData1);
```

Defining the schema for a ProDataSet parameter

Defining the schema for an ABL ProDataSet (DATASET or DATASET-HANDLE) parameter is a multi-step process.

To define the schema for a ProDataSet parameter:

1. If the parameter is for input or input-output, define an ADO.NET DataSet to hold the parameter value.
2. Define a ProDataSetMetaData object to specify its schema.
3. Define the temp-tables for the ProDataSetMetaData object.
4. Define the data-relations for the ProDataSetMetaData object.
5. Add the ProDataSetMetaData object and the DataSet (if applicable) as a ProDataSet parameter to your ParamArray object using the appropriate set parameter method.

Defining a ProDataSetMetaData object

For each ProDataSet parameter you must define a Progress.Open4GL.Proxy.ProDataSetMetaData object. You use this object to specify the schema for the ProDataSet when you set the DATASET or DATASET-HANDLE parameter in the ParamArray object. You can create an instance of this object using the following constructor:
Syntax

```csharp
public ProDataSetMetaData(string dataSetName, string strongTypeName)
```

```csharp
dataSetName

Specifies the ProDataSet name in ABL.
```

```csharp
strongTypeName

Specifies the type name for a strongly-typed ADO.NET DataSet, or null.
```

**Defining the temp-tables for the ProDataSetMetaData object**

You must define the meta data for each temp-table contained by the `ProDataSetMetaData`. For more information, see Defining a `TempTableMetaData` object on page 153 and Adding field descriptions to the `TempTableMetaData` object on page 155.

You can then add the temp-table meta data (`TempTableMetaData`) to the `ProDataSetMetaData` object using the following `ProDataSetMetaData` method:

**Syntax**

```csharp
public void AddDataTable(TempTableMetaData dtMetaData)
```

```csharp
dtMetaData

Specifies the meta data for a temp-table.
```

**Defining the data-relations for the ProDataSetMetaData object**

You must define any data-relations that are defined for the ABL ProDataSet using `Progress.Open4GL.Proxy.ProDataRelationMetaData` objects. You can create a `ProDataRelationMetaData` object for each data-relation between a parent and child temp-table using the following constructor:

**Syntax**

```csharp
public ProDataRelationMetaData(string dataRelationName, int parentIx, int childIx, int numPairs, string pairsList)
```

```csharp
dataRelationName

Specifies the name of the `ProDataRelationMetaData` object.
```

```csharp
parentIx

Specifies a 0-based index to a parent temp-table (ADO.NET `DataTable`) that corresponds to the order in which you have added the `TempTableMetaData` to the `ProDataSetMetaData` using the
```
ProDataSetMetaData AddDataTable( ) method. If you have the ADO.NET DataSet for the ProDataSet, you can also use the ADO.NET DataSet.Tables property to identify the order for this index value.

*childIx*

Specifies a 0-based index to a child temp-table (ADO.NET DataTable) that corresponds to the order in which you have added the TempTableMetaData to the ProDataSetMetaData using the ProDataSetMetaData AddDataTable( ) method. If you have the ADO.NET DataSet for the ProDataSet, you can also use the ADO.NET DataSet.Tables property to identify the order for this index value.

*numPairs*

Specifies the number of column property pairs (key field pairs) that represent this relationship. This allows multiple fields to represent a key relationship between the parent and child temp-table.

*pairsList*

Specifies a string containing a comma-separated list of field names. The list consists of numPairs field pairs, where the child temp-table field name for each pair is followed by its matching parent temp-table field name. The data types of the named child and parent temp-table field pairs must be comparable.

**Note:** For the equivalent ProDataRelationMetaData( ) constructor in the Java OpenAPI, the order of child and parent field names in the pairsList parameter is reversed. For more information, see OpenEdge Development: Java Open Clients.

You can add each data-relation definition to the ProDataSetMetaData object using the following method:

```csharp
public void AddDataRelation(ProDataRelationMetaData drMetaData)
```

*drMetaData*

Specifies a data-relation to include in the ProDataSetMetaData object.

**Adding the ProDataSetMetaData object with its DataSet as a ProDataSet parameter**

Add the ProDataSetMetaData object together with the ADO.NET DataSet it describes by passing them as parameters of the AddDataset( ) or AddDatasetHandle( ) method that you use to add the ProDataSet parameter to the ParamArray object. For more information, see DATASET and DATASET-HANDLE on page 143.

The following example adds a temp-table parameter defined with no indexes.
Sample OpenAPI fragment setting a ProDataSet parameter

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

// Create the ProDataSet
DataSet proDataSet = new DataSet( );
...

// Create the ProDataSetMetaData
ProDataSetMetaData dsMetaData = new ProDataSetMetaData
    ("dsCustOrd", "ProDataSetTest.StrongTypesNS.dsCustOrdDataSet");

// Create the TempTableMetaData for the Customer table
TempTableMetaData ttCustMD = new TempTableMetaData
    ("ttCust", "ttCustDataTable", 3, false, , 0,
    null, null, null);

ttCustMD.SetFieldMetaData(1, "CustNum", 0, Parameter.PRO_INTEGER, 0, 0);

// Create the TempTableMetaData for the Order table
TempTableMetaData ttOrderMD = new TempTableMetaData
    ("OrderDetails",
    "OrderProxy.StrongTypesNS.OrderDetailsDataTable",
    6, false, , 0, null, null, null);

// Add the tables to the DataSet meta data
dsMetaData.AddDataTable(ttCustMD);
dsMetaData.AddDataTable(ttOrderMD);

// Create and add the relations to the DataSet meta data
ProDataRelationMetaData relation = new ProDataRelationMetaData
    ("custNum", 1, 0, 1, "CustNum", null);
dsMetaData.AddDataRelation(relation);

parms.AddDataset(0, proDataSet, ParamArrayMode.OUTPUT, dsMetaData);
```

Defining the return type for a user-defined function

For user-defined functions, there are four properties of the `Progress.Open4GL.Proxy.ParamArray` class that you can set to define the return type:

- **ReturnType property** on page 161
- **IsReturnExtent property** on page 162
UnknownReturnType property on page 162

IsReturnExtent property on page 162

The following sections describe these properties.

**ReturnType property**

This property sets the return type of the function return value. This is the syntax:

**Syntax**

```plaintext
public int ReturnType
```

You must set this property to the class constant defined in `Progress.Open4GL.Parameter` that corresponds to the ABL data type returned by the user-defined function, as shown in the following table.

**Table 32: User-defined function return types for the OpenAPI**

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Progress.Open4GL.Parameter class constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>PRO_CHARACTER</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>PRO_COMHANDLE</td>
</tr>
<tr>
<td>DATE</td>
<td>PRO_DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>PRO_DATETIME</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>PRO_DATETIMETZ</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PRO_DECIMAL</td>
</tr>
<tr>
<td>HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
<tr>
<td>INT64</td>
<td>PRO_INT64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>PRO_INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>PRO_LOGICAL</td>
</tr>
<tr>
<td>RAW</td>
<td>PRO_RAW</td>
</tr>
<tr>
<td>RECID</td>
<td>PRO_RECID</td>
</tr>
<tr>
<td>ROWID</td>
<td>PRO_ROWID</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>PRO_WIDGETHANDLE</td>
</tr>
</tbody>
</table>

**Note:** User-defined functions returning LONGCHAR or MEMPTR cannot be run across the Open Client interfaces.
For example, to set the return type for a user-defined function to the ABL INTEGER data type, you can set the `ReturnType` property on a `ParamArray` object, `parms`, as follows:

```csharp
parms.ReturnType = Parameter.PRO_INTEGER;
```

**IsReturnExtent property**

If the `ReturnType` property is set, and `IsReturnExtent` is set to `TRUE`, then the return value will be an array of the specified type. The default is `FALSE`. This is the syntax:

**Syntax**

```csharp
public bool IsReturnExtent
```

**UnknownReturnType property**

This property allows you to specify if and how the user-defined function can return the ABL Unknown value (`?`). This is the syntax:

**Syntax**

```csharp
public Progress.Open4GL.UnknownType UnknownReturnType
```

`Progress.Open4GL.UnknownType` is an enumeration type defined with the following values:

```csharp
public enum Progress.Open4GL.UnknownType
{
    NullableType,
    HolderClass,
    None
}
```

The `UnknownType.NullableType` and `UnknownType.HolderClass` value both indicate that the user-defined function can return the Unknown value (`?`), but specify a different mechanism for returning this value when the OpenAPI return type corresponds to a supported .NET value type (`System.Boolean`, `System.DateTime`, `System.Decimal`, `System.Int32`, or `System.Int64`).

The specified mechanism is used when the `ReturnType` property is set to one of the following `Progress.Open4GL.Parameter` class constants, each of which maps to a supported value type:

- `PRO_DATE`
- `PRO_DATETIME`
- `PRO_DATETIMETZ`
- `PRO_DECIMAL`
- `PRO_INTEGER`
- `PRO_INT64`
• PRO_LOGICAL

• PRO_RECID

Thus, if you set UnknownReturnType to:

• **UnknownType.NullableType** — The function can return the Unknown value (?) and returns its value as the matching nullable type (or nullable type array) when the ReturnType property setting corresponds to a supported .NET value type (or value type array). If the ReturnType property setting corresponds to a reference type, the specified reference type is returned.

• **UnknownType.HolderClass** — The function can return the Unknown value (?) and returns its value as the matching OpenEdge type-holder class (or holder class array) when the ReturnType property setting corresponds to a supported .NET value type (or value type array). If the ReturnType property setting corresponds to a reference type, the specified reference type is returned.

**Note:** This value is provided for backward compatibility only.

• **UnknownType.None** — The function cannot return the Unknown value (?), which is the default. In this case, the function returns its value as the .NET value or reference type that corresponds to the ReturnType property setting.

**Note:** If you set the IsReturnUnknown property to TRUE, this sets UnknownReturnType to UnknownType.HolderClass. If you set IsReturnUnknown to FALSE, this sets UnknownReturnType to UnknownType.None. Note also that IsReturnUnknown is supported for backward compatibility only. For more information on the IsReturnUnknown property, see *IsReturnUnknown property* on page 163.

For example, if the ReturnType property represents an ABL INTEGER EXTENT and UnknownReturnType is set to:

• **UnknownType.NullableType** — The ReturnValue property returns a System.Int32?[].

• **UnknownType.HolderClass** — the ReturnValue property returns an IntHolder[].

• **UnknownType.None** — The ReturnValue property returns a System.Int32[].

For more information on the ReturnValue property, see *Accessing user-defined function return values* on page 168.

For more information on the use of .NET nullable types and OpenEdge type-holder classes, see *Unknown value (?) as a parameter* on page 50.

If a scalar return value can be the Unknown value (?), check if the ReturnValue property is null. If it is not null, you can then unbox ReturnValue to get the value on a ParamArray object, parms, as follows:

```csharp
int intRetVal = (int)parms.ReturnValue;
```

**IsReturnUnknown property**

Set this property to TRUE in order to indicate that the user-defined function can return the Unknown value (?). This setting also sets the UnknownReturnType property to the enumeration value, UnknownType.HolderClass. If the function cannot return the Unknown value (?), set this property to FALSE, which also sets the UnknownReturnType property to the enumeration value, UnknownType.None.
Note: This property is supported for backward compatibility only. Progress Software recommends that you directly set and test the UnknownReturnType property instead. For more information, see UnknownReturnType property on page 162.

This is the syntax:

Syntax

```csharp
public bool IsReturnUnknown
```

For example, if the ReturnType property is an INTEGER EXTENT, and this property is set to TRUE, the ReturnValue property returns an IntHolder[]. If it is set to FALSE, the ReturnValue property returns an Int32[]. The default is FALSE. For information on the ReturnValue property, see Accessing user-defined function return values on page 168.

If a scalar return value can be the Unknown value (?), check if the ReturnValue property is null. If it is not null, you can then unbox ReturnValue to get the value on a ParamArray object, parms, as follows:

```csharp
int intRetVal = (int)parms.ReturnValue;
```

Passing parameters

After you have set up the required parameter, you can pass the specified parameters to a procedure or user-defined function. The steps for setting up and passing parameters differs, depending on the parameter mode (INPUT, OUTPUT, or INPUT-OUTPUT).

Passing INPUT parameters

Passing an INPUT parameter requires several steps to provide a value as input to an application service procedure or user-defined function.

To pass an INPUT parameter:

1. Create and initialize a variable for the parameter of the correct .NET data type (see Creating variables for parameters on page 139).
2. Add the parameter to a ParamArray object (see Setting up a parameter array on page 139).
3. Run the procedure or user-defined function (see Running procedures and user-defined functions on page 136).
Passing an INPUT parameter using the .NET OpenAPI

INPUT parameter as Unknown value (?) using .NET OpenAPI

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

```csharp
// Define and initialize the variable for the input parameter
Int32 CustomerNumber = 33;

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

// Add the input parameter to the ParamArray
parms.AddInteger(0, CustomerNumber, ParamArrayMode.INPUT);

// Run the procedure or user-defined function
...
```

The following example passes an INPUT integer parameter that supports the Unknown value (?).

```csharp
// Define and initialize the variable for the input parameter
Int32 CustomerNumber = 33;

... // Define and set the holder class for the input parameter
// to null or an integer based on the variable value
IntHolder hCustomerNumber = new IntHolder();
iCustomerNumber = new Integer(CustomerNumber)
if (CustomerNumber > 33)
    hCustomerNumber.Value = null;
else
    hCustomerNumber.Value = (Object) iCustomerNumber;

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

// Add the input parameter to the ParamArray
parms.AddInteger(0, hCustomerNumber, ParamArrayMode.INPUT);

// Run the procedure or user-defined function
...
```

Passing INPUT-OUTPUT parameters

Passing an INPUT-OUTPUT parameter requires several steps to provide a value as input to an application service procedure or user-defined function, then to return a value as output using the same parameter.

To pass an INPUT parameter:

1. Create and initialize a variable for the parameter of the correct .NET data type (see Creating variables for parameters on page 139).
2. Add the parameter to a ParamArray object (see Setting up a parameter array on page 139).
3. Run the procedure or user-defined function (see Running procedures and user-defined functions on page 136).

4. Get the output value from the ParamArray (see Getting OUTPUT parameter values on page 167). You might also want to reset the variable.

**Passing an INPUT-OUTPUT parameter using the .NET OpenAPI**

The following example passes an INPUT-OUTPUT integer parameter that does not support the Unknown value (?).

```csharp
// Define and initialize the variable for the input-output parameter
Int32 CustomerNumber = 33;

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

// Add the input-output parameter to the ParamArray
parms.AddInteger(0, CustomerNumber, ParamArrayMode.INPUT_OUTPUT);

// Run the procedure
...

// Fill output parameter
// The output value is always returned as an Object
// You need to cast the Object before assigning it.
CustomerNumber = (Int32) parms.GetOutputParameter(0);
```

**Passing OUTPUT parameters**

Passing an OUTPUT parameter requires several steps to provide a parameter as output from an application service procedure or user-defined function, then to get the value returned by the parameter.

To pass an OUTPUT parameter:

1. Create and initialize a variable for the parameter of the correct .NET data type (see Creating variables for parameters on page 139).
2. Add the parameter to a ParamArray object (see Setting up a parameter array on page 139).
3. Run the procedure or user-defined function (see Running procedures and user-defined functions on page 136).
4. Get the output value from the ParamArray (see Getting OUTPUT parameter values on page 167).
Passing an OUTPUT parameter using the .NET OpenAPI

The following example passes an OUTPUT integer parameter that does not support the Unknown value (?).

```csharp
// Define the variable for the output parameter
Int32 CustomerNumber;

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

// Set up output parameter - notice no variable is needed at this point
parms.AddInteger(0, null, ParamArrayMode.OUTPUT);

// Run the procedure
...

// Fill the output parameter
// The output value is always returned as an Object
// You need to cast the Object before assigning it.
CustomerNumber = (Int32) parms.GetOutputParameter(0);
```

Handling returned values

The .NET OpenAPI provides methods to access returned values for:

- Output parameters
- The RETURN-VALUE function set by running an ABL non-persistent, persistent, or internal procedure
- The value returned by an ABL user-defined function

The following sections describe how to access each type of return value.

Getting OUTPUT parameter values

After running a procedure or user-defined function, you can access the OUTPUT parameters using the following ParamArray method:

**Syntax**

```csharp
public System.Object GetOutputParameter(int paramNum)
```

**paramNum**

Specifies the 0-based position of the parameter.
The output value is always returned as a `System.Object`. If the `Object` returned cannot be `null`, you need to cast the `Object` to the output parameter data type and assign it to the output variable you have created to hold the cast value. If the `Object` returned can be `null`, you need to assign it to the `Value` property of a holder object, then test for `null` before obtaining the appropriate data type value as output.

The following example shows getting an output integer parameter (or the output side of an input-output parameter) that might be set to the Unknown value (?) in ABL.

**Getting output parameters using the .NET OpenAPI**

```csharp
IntHolder hCustomerNumber = new IntHolder();
Int32 CustomerNumber = 33;

// Set the parameter and run the procedure
ParamArray parms = new ParamArray(1);
parms.AddInteger(0, CustomerNumber, ParamArrayMode.INPUT-OUTPUT);
...

// Fill the output parameter
hCustomerNumber.Value = parms.GetOutputParameter(0);
if (hCustomerNumber.IsNull == true) ...;
else CustomerNumber = hCustomerNumber.IntValue;
...
```

**Accessing RETURN-VALUE function output from procedures**

You can access the value of the `RETURN-VALUE` function set by running an ABL procedure using the following property, which is available from any of three .NET OpenAPI objects:

**Syntax**

```csharp
public string ProcReturnString
```

You can access this property from one of the following objects:

- `Progress.Open4GL.Proxy.OpenAppObject` — Returns the value set by the last procedure executed on the associated AppServer
- `Progress.Open4GL.Proxy.OpenProcObject` — Returns the value set by the last internal procedure executed in the associated persistent procedure
- `Progress.Open4GLProxy.ParamArray` — Returns the value set by the last non-persistent, persistent, or internal procedure executed and passed the specified `ParamArray` object

For more information on using these objects to execute procedures, see Running procedures and user-defined functions on page 136.

**Accessing user-defined function return values**

You can access the return value after running any user-defined function using the `ReturnValue` property on the `Progress.Open4GL.Proxy.ParamArray` object. This is the syntax:
Syntax

```c
public System.Object ReturnValue
```

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

```c
string retVal = (string) parms.ReturnValue;
```

The subclass of `System.Object` that is returned depends on how you define the return type for a given function. For more information, see Defining the return type for a user-defined function on page 160.

When the return value is an array, the `Progress.Open4GL.Proxy.ParamArray.IsReturnArray` property will be `TRUE`. Since the `ReturnValue` property returns an object, you can cast it to an array.

```c
ParamArray parms = new ParamArray(1);
...
// Set up return type (udf does not return an unknown value)
parms.ReturnType = Parameter.PRO_DECIMAL;
parms.IsReturnExtent = true;
// Run procedure
openFO.RunProc("GetOrderTotalsByDollar", parms);
// Get return value
System.Decimal[] retValArray = (System.Decimal[])parms.ReturnValue;
```

**Note:** User-defined functions returning `LONGCHAR` or `MEMPTR` values cannot be run across the Open Client interface.

---

Sample .NET OpenAPI code

The following sections describe two types of sample code:

- **Non-persistent procedure sample** on page 169
- **Persistent procedure sample** on page 171
- **Single-run singleton procedure example** on page 172

**Note:** These samples are not available for download on the Progress Communities Web site.

### Non-persistent procedure sample

The following example shows a non-persistent procedure, `AddCustomer.p`. 
Sample non-persistent procedure ABL for the OpenAPI

```
DEFINE INPUT PARAMETER CustName AS CHARACTER.
DEFINE INPUT PARAMETER phone AS CHARACTER.
DEFINE INPUT PARAMETER email AS CHARACTER.
DEFINE OUTPUT PARAMETER CustomerNumber AS INTEGER.
```

The following example shows an OpenAPI code to run a non-persistent procedure.

**OpenAPI code to run a non-persistent procedure**

```csharp
namespace BigOrderInfoClient {
    using Progress.Open4GL;
    using Progress.Open4GL.Proxy;
    public class samplecode {
        public static void NonPersistentProcedure() {
            // Connect to the AppServer
            Connection myConn = new Connection
                ("AppServer://localhost/asbroker2", "", "", "");
            OpenAppObject openAO = new OpenAppObject(myConn, "mySvc");

            // Set Session model for state-free
            myConn.SessionModel = 1;

            // Create the parameters
            System.String CustName = "abc";
            System.String phone = "999-555-1234";
            System.String email = "me@foo.com";
            System.Int32 outValue;

            // Create a place for RETURN-VALUE
            System.String retVal;

            // Create the ParamArray
            ParamArray parms = new ParamArray(4);

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

            // Set up Out parameters
            parms.AddInteger(3, null, ParamArrayMode.OUTPUT);

            // Run the procedure
            openAO.RunProc("AddCustomer.p", parms);

            // Get output parameters - use holder to handle unknown value
            outValue = (System.Int32)parms.GetOutputParameter(3);

            // Get RETURN-VALUE. Will return null for AddCustomer() procedure
            retVal = (System.String)parms.ProcReturnString;
            openAO.Dispose();
        }
    }
}
```
Persistent procedure sample

The following example shows a portion of code for a procedure, `CustomerInfo.p`, with a user-defined function.

**Procedure CustomerInfo.p**

```plaintext
DEFINE INPUT PARAMETER custNum AS INTEGER NO-UNDO.
...
FUNCTION GetTotalOrdersByNumber RETURNS INTEGER (threshold AS DECIMAL):
...
END.
...
```

The following example shows OpenAPI code that runs `CustomerInfo.p` as a persistent procedure.
OpenAPI code to run the persistent procedure

```csharp
namespace BigOrderInfoClient {
    using Progress.Open4GL;
    using Progress.Open4GL.Proxy;

    public class samplecode {
        public static void PersistentProcedure() {
            // Connect to the AppServer
            Connection myConn = new Connection
                ("AppServer://localhost/asbroker2", "", "", "");
            OpenAppObject openAO = new OpenAppObject(myConn, "mySvc");

            // Run the persistent procedure
            // First set up parameters
            ParamArray parms = new ParamArray(1);
            System.Int32 custNum = 3;

            // Set up input parameters
            parms.AddInteger(0, custNum, ParamArrayMode.INPUT);
            // Run procedure
            OpenProcObject openPO = openAO.CreatePO("OrderInfo/CustomerOrder.p",
                                                       parms);

            // Call UDF
            // First set up parameters
            System.Decimal retVal;
            int Threshold = 1000;
            parms.Clear(); // Clear ParamArray for next call

            // Set up input parameters
            parms.AddDecimal(0, Threshold, ParamArrayMode.INPUT);

            // Set up return type
            parms.ReturnType = Parameter.PRO_DECIMAL;

            // Run procedure
            openPO.RunProc("GetTotalOrdersByNumber", parms);

            // Get return value
            retVal = (System.Decimal)parms.ReturnValue;
            openPO.Dispose();
            openAO.Dispose();
        }
    }
}
```

Single-run/singleton procedure example

The following example shows a user-defined function defined by a procedure called `CustomerInfo.p`.
User-defined function defined by CustomerInfo.p

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

The version of CustomerInfo.p in Persistent procedure sample on page 171 cannot be run as a single-run (or singleton) because it has a parameter in the main block. Creating another internal procedure to set CustNum would also not work, because a single-run procedure is deleted from the AppServer after each call. Context must be established for each call. So in User-defined function defined by CustomerInfo.p CustNum is passed as a parameter to the user-defined function GetTotalOrdersByNumber instead.

The following OpenAPI code runs CustomerInfo.p as a single-run procedure and then calls its user-defined function. The code also shows how to access the procedure type using the ProcedureType parameter. A singleton procedure can be run in the same way, with ProcedureType.Singleton substituted into the CreatePO method.
namespace BigOrderInfoClient {
    using Progress.Open4GL;
    using Progress.Open4GL.Proxy;
    using Progress.Open4GL.ProcedureType;

    public class samplecode {

        public static void SingleRunProcedure( ) {
            // Connect to the AppServer
            Connection myConn = new Connection
                ("AppServer://localhost/asbroker2", "", "", "");
            OpenAppObject openAO = new OpenAppObject(myConn, "mySvc");

            // Run procedure as single-run
            OpenProcObject openPO = openAO.CreatePO("OrderInfo/CustomerInfo.p",
                ProcedureType.SingleRun);

            // Display procedure type using ProcedureType parameter
                + dynPO.ProcedureType);

            // Call UDF
            // First set up parameters
            System.Int32 custNum = 3;
            System.Decimal retVal;
            int Threshold = 1000;

            // Set up input parameters
            ParamArray parms = new ParamArray(2);
            parms.AddInteger(0, custNum, ParamArrayMode.INPUT);
            parms.AddDecimal(1, Threshold, ParamArrayMode.INPUT);

            // Set up return type
            parms(ReturnType = Parameter.PRO_DECIMAL);

            // Run UDF
            openPO.RunProc("GetTotalOrdersByNumber", parms);

            // Get return value
            retVal = (System.Decimal)parms.ReturnValue;

            // Even though the single-run procedure has been deleted on the
            // AppServer, use Dispose() to clean up the resources on the client.
            openPO.Dispose( );
            openAO.Dispose( );
        }
    }
}

A singleton procedure can be run in the same way, with ProcedureType.Singleton substituted into the CreatePO method. Unlike a single-run procedure, a singleton procedure remains instantiated on the AppServer after execution, but you still need to use Dispose() to clean up resources on the client side.
Index

? (ABL unknown value), See Unknown value (?)

.NET objects
  mappings to ProDataSets 63
  mappings to temp-tables
    XML node types 62
    resolving date/time differences 62

.NET Open Client
  types 127
  application building in Visual Studio .NET 128
  localized messages 23

A

ABL methods 29
ABL unknown value, See Unknown value (?)
Accessing
  proxy properties
    application configuration file 94
    Connection class 95
    directly 100
    methods 95
    RunTimeProperties class 94
Accessor methods
  general property
    Connection class 96
    RunTimeProperties class 95
    purposed property 99
AddDataTable( ) method 158
AddParameter( ) method 149
API, See OpenAPI
Application configuration file 94
Application services
  ABL methods 29
  proxy objects 28
AppObjects
  constructors 40
  example 35
  generation 28
  instantiating 40
  session-free method execution 34
AppServer
  establishing a connection 40
  operating modes 43
AppServerKeepalive property 108
Arrays
  parameter mappings 48
  temp-table parameters 69
Assemblies
  generated by ProxyGen 28
Attribute mappings
  buffer-field 59
  Attribute mappings (continued)
    data-relation 65
    ProDataSet 64–65
    temp-table 57
    temp-table and buffer 57

B

Building .NET client applications 24

C

CancelAllRequests( ) method 33
_CancelAllRequests( ) method 33
Class factory methods 30
Clear( ) method 139
Client programming
  exception handling 123
Common methods 33
Common proxy properties 31
Component mappings
  ProDataSet 64
  temp-table 56
Compression properties
  Compression Enabled 109
  Compression Level 109
  Compression Threshold 110
Connecting to an AppServer
  using the OpenAPI 134
Connection
class
  constructors 39
  definition 95
  general accessor methods 96
  relationship to RunTimeProperties class 105
  releasing objects 43
  session-free AppObjects 34
  methods 29
  pool
    session-free AppObject sharing 34
ConnectionID property 31
ConnectionLifetime property 110
ConnectionTimeout property 110
Constructors
  AppObject 40
  Connection class 39
  holder classes 53
  OpenAppObject 134
  ParamArray 139
  ProDataRelationMetaData
    .NET OpenAPI 158
Constructors (continued)
ProDataSetMetaData 157
TempTableMetaData 153
CreateAO_SubAppObject( ) method 30
CreatePO_ProcObject( ) method 31
CreatePO( ) method
OpenAppObject 137

D
Data type mappings
ABL unknown values 50
ABL to .NET proxy 45
ProDataSets 75
temp-table fields 67
DATASET parameters
See also ProDataSets
passing 73
See also ProDataSets
DATASET-HANDLE parameters
See also dynamic
passing 73
Progress extensions 75
See also dynamic
Date conversions
proxy parameters 48
temp-table field default values 62
Deploying .NET client applications 25
Deployment requirements 25
Digitally-signed only .NET assemblies
 localized messages 23
Open Client Runtime package 21
Dispose( ) method 33

E
EnableTracing property 111
Exceptions
DataSet/DataTable exceptions 123
descriptions 124
general ABL
class hierarchy 123
definition 123
Error handling, proxies 123
Examples
ABL procedure mapped to .NET proxy method 29
AppObject 35
Microsoft Visual Studio .NET 129
.NET OpenAPI code
non-persistent procedure 169
persistent procedure 171
ProcObject 36
ProDataSet parameters
dynamic 79
in .NET Open Client 79
static 77

Examples (continued)
ProDataSet parameters (continued)
updating 80
SubAppObject 36
temp-table parameters
dynamic 72
in .NET Open Client 73
static 71

F
Files
application configuration 94

G
Generating .NET proxies 23
GetArrayFieldName( ) method 82
GetImageFlag( ) method 83
GetColumnProType( ) method 86
GetDataSourceModified( ) method, ProDataSet class 90
GetDataSourceModified( ) method, ProDataTable class 88
GetErrorString( ) method 87
GetNoSchemaMarshal( ) method 89
GetOutputParameter( ) method
ParamArray 167
GetRejected( ) method, ProDataSet class 90
GetRejected( ) method, ProDataTable class 88
GetUndo( ) method 90
GetUserOrder( ) method 86

H
Holder classes
super class definition 53

I
IdleConnections property 112
IdleConnectionTimeout property 111
INPUT parameters
.NET OpenAPI 164
Unknown value (?) for .NET value types
 holder classes 55
 nullable 53
INPUT-OUTPUT parameters
.NET OpenAPI 165
Unknown value (?) for .NET value types
 holder classes 55
 nullable 53
IsExtentColumn( ) method 82
IsReturnExtent property 162
IsReturnUnknown property 163
L

Localized messages
  Digitally-signed only .NET assemblies 23
  Strong-named and signed .NET assemblies 23
  Strong-named only .NET assemblies 23

Log file 28
LogFileProperty 112, 114
LoggingLevel property 112

M

MaxConnections property 113
Methods
  .NET OpenAPI
    data type-specific add parameter 140
generic add parameter 149
AddDataTable( ) 158
CancelAllRequests( ) 33
class factory 30
Clear( ) 139
common 33
connection 29
CreateAO_SubAppObject( ) 30
CreatePO_ProcObject( ) 31
CreatePO( )
  OpenAppObject 137
Dispose( ) 33
GetOutputParameter( ) 167
  CancelAllRequests( ) 33
ProDataSet class
data source update-related 90
ProDataTable class
  array handling 82
  before-image 83
column-related 84
data source related 88
error handling 87
schema marshaling 89
transaction 89
proxy object 95
releaseConnection( ) 43
remote ABL 29
running on session-free AppObjects 34
RunProc( )
  OpenAppObject 136
  OpenProcObject 138
SetFieldMetaData( ) 155
MinConnections property 113

N

Namespace
  ProDataSet 73
proxy objects 28

Namespace (continued)
XML
  ProDataSets and ADO.NET DataSets 65
temp-tables and ADO.NET DataTables 57
.NET OpenAPI, See OpenAPI
NoHostVerify property 113
NoSchemaMarshal property 114
NoSessionReuse property 115
NsClientMaxPort property 115
NsClientMinPort property 115
NsClientPickListExpiration property 116
NsClientPicklistSize property 116
NsClientPortRetry property 116
NsClientPortRetryInterval property 117
Nullable types 51

O

Open Client Runtime
  selecting a package 21
Open Client Runtime package
  Digitally-signed only .NET assemblies 21
  Strong-named and signed .NET assemblies 21
  Strong-named only .NET assemblies 21
Open4GLException class 123
OpenAPI
  See also OpenAppObject class
AppServer connections 134
classes summary 133
parameters
  adding with data type-specific methods 140
  adding with generic methods 149
  CHARACTER methods 142
  COM-HANDLE methods 142
  creating variables 139
  DATASET and DATASET-HANDLE methods 143
  DATE methods 143
  DATETIME methods 143
  DATETIME-TZ methods 144
  DECIMAL methods 144
defining ProDataSet data-relations 158
defining ProDataSet temp-tables 158
HANDLE methods 145
INPUT-OUTPUT, passing 165
INPUT, passing 164
INT64 methods 145
INTEGER methods 146
LOGICAL methods 146
LONGCHAR methods 147
MEMPTR methods 147
OUTPUT, passing 166
OUTPUT, value 167
parameter array 139
ProDataSet instance 159
ProDataSet schema 157
OpenAPI (continued)
parameters (continued)
  RAW methods 148
  RECID methods 148
  ROWID methods 148
  TABLE and TABLE-HANDLE methods 149
  temp-table instance 156
  temp-table schema 153
  UDF return types 160
returned values
  OUTPUT parameters 167
  RETURN-VALUE function from procedures 168
  user-defined function 168
running
  application services 136
  internal procedures and user-defined functions 138
  non-persistent procedures 136
  persistent procedures 137
  user-defined function return types 160
See also OpenAppObject class
OpenAppObject class
  accessing ABL procedure return values 168
  constructors 134
  method parameters, See OpenAPI parameters
running
  application services 136
  non-persistent procedures 136
  persistent procedures 137
OpenProcObject class
  accessing ABL procedure return values 168
  creating instances 137
  method parameters, See OpenAPI parameters
running internal procedures and UDFs 138
OUTPUT parameters
.NET OpenAPI 166

P

ParamArray class
  accessing ABL procedure return values 168
  constructor 139

ParamArrayMode class
  constants passed to data type-specific methods 140
  constants passed to generic methods 149

Parameter class 84
Parameters
  date and time conversions
  temp-table field default values 62
INPUT
  ProDataSets 75
  temp-tables 69
INPUT-OUTPUT
  ProDataSets 76
  temp-tables 70

Parameters (continued)
  OUTPUT
    ProDataSets 76
    temp-tables 71
.NET OpenAPI
    adding with data type-specific methods 140
    INPUT-OUTPUT, passing 165
    adding with generic methods 149
    INPUT, passing 164
    OUTPUT, passing 166
    passing
      by reference 50
    Progress array mappings 48
    Unknown value (?) mappings 50
Passing
  parameters
    by reference 50
  ProcedureType property 32

ProcObjects
  example 36
  generation 28

ProcReturnString property 31, 168

ProDataRelationMetaData class
  constructors 158

ProDataSet class
  strongly typed DataSets 73
  adding for OpenAPI 159
  attribute mappings
    data-relation 65
    ProDataSet 65
  component mappings 64
  data type mappings 75
  defining data-relations for OpenAPI 158
  defining temp-tables for OpenAPI 158
  dynamic 75
  examples
    dynamic 79
    in .NET Open Client 79
    static 77
    updating 80
  mappings to .NET objects 63
  namespace 73
  passing 73
  passing as INPUT 75
  passing as INPUT-OUTPUT 76
  passing as OUTPUT 76
  preparing to pass for OpenAPI 157
  ProDataSet class 90
  static 75
  updating 76
  XML namespace 65

ProDataSetMetaData class
  constructor 157
  defining data-relations for OpenAPI 158
  defining temp-tables for OpenAPI 158
Properties (continued)
WaitIfBusy 121
Proxy
exceptions 123
handling errors 123
methods
ABL return values 30
class factory 30
property accessor 95
running on session-free AppObjects 34
namespace 28
objects
AppObject example 35
AppObject instantiation 40
generated by ProxyGen 28
ProcObject example 36
SubAppObject example 36
objects and methods, See Methods, Proxy objects and methods
properties
common 31
Connection class 95
direct access 100
list of available 97
method access 95
modifying Connection object 106
RunTimeProperties class 94
server specification 102
session 100
setting in application configuration file 94
thread control 102
Tracing 103
requirements
generation 23
using in applications 24
ProxyHost property 117
ProxyPassword property 117
ProxyPort property 118
ProxyUserId property 118
R
releaseConnection( ) method 43
Remote ABL methods 29
RequestID property 32
RequestWaitTimeout property 119
Return values
ABL 30
Return values, ABL
.NET OpenAPI 167
ReturnType property 161
ReturnValue property 168
RunProc( ) method
OpenAppObject 136
OpenProcObject 138
RunTimeProperties class
  definition 94
general accessor methods 95
  purposed accessor methods 99
relationship to Connection class 105

S
Sample .NET client applications
  OpenAPI example code
    non-persistent procedure 169
    persistent procedure 171
Secure Sockets Layer (SSL)
  properties 103
Server specification properties 102
Session properties 100
SessionModel property 119
SetBImageFlag ( ) method 83
SetColumnProType ( ) method 87
SetErrorString ( ) method 88
SetExtentColumns( ) method 82
SetFieldMetaData( ) method 155
SetNoSchemaMarshal( ) method 89
SetUserOrder( ) method 87
Single-run procedure
  OpenAPI
    example 172
  ProxyGen 31
Singleton procedure
  OpenAPI
    example 172
  ProxyGen 31
SSLSubjectName property 32
Strong-named and signed .NET assemblies
  localized messages 23
  Open Client Runtime package 21
Strong-named only .NET assemblies
  localized messages 23
  Open Client Runtime package 21
SubAppObjects
  example 36
  generation 28

T
  TABLE parameter, See Temp-table parameters static
  TABLE-HANDLE parameters
  See also Temp-table parameters dynamic
    Progress extensions 70
    See also Temp-table parameters dynamic
Temp-table parameters
  .NET OpenAPI
    adding 156
    adding field descriptions 155
    defining a TempTableMetaData object 153
    schema definition 153
  Temp-table parameters (continued)
    array fields 69
    attribute mappings
      buffer-field 59
      temp-table and buffer 57
    component mappings 56
    dynamic 67
    examples
      dynamic 72
        in .NET Open Client 73
      static 71
    field data types
      ABL to .NET mappings 67
      specifying ABL types 84
    indexes 69
    mappings to .NET object s
      resolving date/time differences 62
      XML node types 62
    passing 66
    passing as INPUT 69
    passing as INPUT-OUTPUT 70
    passing as OUTPUT 71
    ProDataTable class 82
    static 67
    .NET object mappings 66
    XML namespace 57
TempTableMetaData class
  adding field descriptions for OpenAPI 155
  constructor 153
  defining an instance for OpenAPI 153
Thread control properties 102
Threading model
  session-free 34
Time conversions
  proxy parameters 48
  temp-table field default values 62
Tracing properties 103

U
  Unknown value (?)
    holder classes
      INPUT or INPUT-OUTPUT parameters 55
      INPUT-OUTPUT or OUTPUT parameters 55
    nullable types
      INPUT or INPUT-OUTPUT parameters 53
      INPUT-OUTPUT or OUTPUT parameters 53
    temp-table fields 69
    .NET value-type mappings 51, 53
    .NET client programming 50
  UnknownReturnType property 162
  Updating ProDataSet parameters 76

V
  Visual Studio .NET 128
**W**

WaitIfBusy property  121  
Web forms (ASP.NET) clients  128  
Web services (ASP Web services)  128  
Windows forms (GUI) clients  128  

**X**

XML  
namespace  
ProDataSets and ADO.NET DataSets  65  
temp-tables and ADO.NET DataTables  57  
node type specifications  62  
