Progress®



OpenEdge® Development:
GUI for .NET Programming

OpenEdge®

Copyright

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

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

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

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....11
Purpose....11
Audience....12
Organization....12
Using ABL documentation....13
References to ABL compiler and run-time features....13
References to ABL data types....13
Examples of syntax descriptions....14
Long syntax descriptions split across lines....15
Complex syntax descriptions with both required and optional elements....16
OpenEdge messages....16
Obtaining more information about OpenEdge messages....17

Chapter 1: Overview....19
General capabilities and limitations....20
Capabilities....20
Limitations....20
Object model and architecture....21
Incorporation of the .NET object model....21
GUI for .NET run-time architecture....24
Performance note....25
Simple example....26

Chapter 2: Accessing and Managing .NET Classes from ABL....29
Supported features and limitations....30
Support for .NET classes....30
Limitations of support for .NET classes....31
Referencing and instantiating .NET classes....34
How .NET organizes types for reference....34
Identifying .NET assemblies to ABL....35
Referencing .NET class and interface types....37
Using case sensitivity with .NET objects....41
Using unqualified .NET type names....41
Instantiating and managing .NET class instances....42
Accessing .NET class members....45
Accessing members of .NET interfaces....47
Accessing static members of a .NET class....49
Specifying .NET constructor and method parameters....50

Accessing .NET indexed properties and collections....52
Handling .NET events....57
Managing .NET events in ABL....58
Identifying the events published by a .NET object....59
Defining handlers for .NET events in ABL....59
Specifying handler subscriptions for .NET events....61
Managing .NET events from ABL-extended .NET classes....61
Event handling example....62
Handling .NET exceptions....65
Using the ABL error trapping constructs....66
Using properties and methods on .NET Exception objects....68
Unique scenarios when handling errors with .NET objects....69
Defining ABL-extended .NET objects....70
Features of ABL-derived .NET classes....70
Deriving .NET classes in ABL....72
Managing events for ABL-derived .NET classes....81
Features of ABL classes that implement .NET interfaces....85
Implementing .NET interfaces in ABL....85
Error handling for ABL-extended .NET classes....88

Chapter 3: Creating and Using Forms and Controls....89

OpenEdge .NET form and control objects....90
Commonly-used .NET public form methods, properties, and events....91
Commonly-used .NET public control methods, properties, and events....94
ABL support for managing .NET forms and controls....98
Initializing and blocking on .NET forms....99
Blocking on non-modal forms....100
Blocking on modal dialog boxes....103
Accessing .NET forms using the SESSION system handle....107
Accessing resource files for .NET forms....109
Creating custom .NET forms and controls....110
Sample ABL-derived .NET non-modal form....112
Sample ABL-derived .NET modal dialog box....115
Sample ABL-derived .NET MDI form....120
Sample ABL-derived .NET user control....126

Chapter 4: Binding ABL Data to .NET Controls....131

ABL and .NET data binding models....132
ABL data binding....132
.NET data binding....133
ABL data binding model for .NET....133
Extent of integration with .NET controls....133
Supported .NET controls....134

Supported ABL data sources....134
Understanding the ProBindingSource....135
Constructors....135
Properties....140
Methods....144
Events....145
Data binding examples....146
Programming considerations....156
Managing data updates....159
Enabling and disabling updates....159
Sorting....160
Maintaining currency with the Position property....163
Synchronizing data....164
Handling user interface events....165
Example of an updatable grid....172
Definitions....173
Main block....175
Internal procedures and functions....175

Chapter 5: Using .NET Forms with ABL Windows....183

Features for using forms and windows together....184
ABL session architecture for forms and windows....184
Shadow windows....186
FormProxy objects....187
Embedded windows....187
Parenting forms and windows to each other....187
Embedding ABL windows in .NET forms....189
Elements of an embedded ABL window....189
Using the ABL features to embed a window....190
Embedding a single window in an MDI child form....190
Embedding one or more windows in a .NET form....192
Behavior of forms with embedded windows....195
Handling form and window input....195
Common event handling for forms and windows....195
Common focus detection for forms and windows....196
Managing form and window run-time behavior....197
Configuring common session features....199
Palette management....199
Font management....199
Regional settings: localization....199

Chapter 6: OpenEdge Installed .NET Controls....201

OpenEdge Controls....201

Microsoft .NET UI Controls....202
OpenEdge Ultra Controls for .NET....202

Chapter 7: Using .NET data types in ABL....205

Overview of .NET data types in ABL....206
Value types in ABL....207
Reference types in ABL....207
General ABL support for .NET types....207
Support for .NET mapped object and primitive data types....207
Support for .NET object types....208
Support for .NET value types as objects....209
Implicit data type mappings....210
Implicit array mappings....212
An ABL INTEGER larger than its .NET destination....212
Negative ABL values and unsigned .NET destinations....213
ABL DECIMAL and .NET System.Decimal....213
Default matching ABL and .NET data types....214
An ABL INT64 larger than its .NET System.UInt32 destination....215
An ABL DECIMAL larger than its .NET System.UInt64 destination....215
ABL DECIMAL and .NET System.Double or System.Single....215
Explicit data type mappings....216
Assigning between ABL primitive or array types and System.Object....218
Passing ABL data types to .NET constructor and method parameters....218
Indicating explicit .NET data types....218
Data type widening....219
Getting .NET data member, property, and method return values....221
.NET boxing support....223
Automatic boxing....223
Manual boxing....225
.NET null values and the ABL Unknown value (?)....228
Support for ADO.NET DataSets and DataTables....228
Accessing and using .NET enumeration types....228
Enumerations in .NET....229
.NET enumerations in ABL....229
Using the Progress.Util.EnumHelper class....230
Working with .NET generic types....232
Identifying generic type parameters....232
Identifying constraints on generic type parameters....232
Accessing and using .NET arrays....235
Accessing .NET arrays....235
Array mapping: conversion between ABL arrays and .NET arrays....238
Array assignment....239
Example: Mapping an ABL array to a .NET array....245
Example: Accessing a .NET array....246

Chapter 8: Notes on using the ProBindingSource 251

Determining what to bind to the ProBindingSource.....251
Repopulating data in an ABL data source bound to a ProBindingSource.....252
Using the ProBindingSource AddNew( ) and Remove( ) methods.....253

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using ABL documentation
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This manual describes how to access .NET objects using ABL to implement the *OpenEdge® GUI for .NET* in your OpenEdge applications. Without the GUI for .NET, ABL provides a native, handle-based, object model that supports built-in visual objects (widgets) that you can use to build a window-based GUI. This traditional OpenEdge GUI provides many of the features of a modern GUI. However, even with support for ActiveX controls, the traditional OpenEdge GUI does not provide the extensibility and flexibility of a GUI that you can build using the Microsoft .NET Framework.

In addition to its built-in handle-based object model, ABL also supports a class-based object model that allows you to build user-defined classes similar to classes in Java or .NET. OpenEdge further extends the ABL class-based object model to include classes built using the Microsoft .NET Framework. Using the Microsoft (and other third-party) .NET classes, you can build an extensible and flexible GUI (the OpenEdge GUI for .NET). In addition, OpenEdge supports its own .NET object types to provide a more natural integration of the GUI for .NET with both the traditional OpenEdge GUI and ABL data.

This manual describes how ABL supports .NET object types in an ABL compile-time and run-time environment. It then describes how you can access .NET classes in assemblies entirely from within an ABL session, where you can work with them in the same way as ABL user-defined classes and interfaces. In addition, this support for the GUI for .NET includes features of .NET objects not currently supported for ABL user-defined objects. Thus, this manual explains how to build ABL applications and GUIs using .NET objects without having to go outside the ABL environment or use any native .NET languages.

Note that this documentation does not replace and, in fact, depends upon both Microsoft and third-party vendor documentation for understanding the .NET classes that vendors provide.

In addition to this manual, which describes how to use the basic ABL elements for programming with the OpenEdge GUI for .NET, you can find more information on using this feature in the following manuals:

- OpenEdge Getting Started: GUI for .NET Primer — A brief overview of the OpenEdge GUI for .NET and the class-based ABL that supports it.
- OpenEdge Development: ABL Reference — The complete reference to the ABL, including all language elements that support the GUI for .NET.
- OpenEdge Development: GUI for .NET Mapping Reference — A short reference to ABL elements and features of the GUI for .NET both as they map to the terminology and C# language syntax of the Microsoft .NET Framework, and as they map to ABL elements and features of the traditional OpenEdge GUI.
- OpenEdge Deployment: Managing ABL Applications — A guide to managing and deploying ABL applications, with sections on requirements and features for deploying GUI for .NET applications.

Audience

This manual assumes that you are an ABL application developer who wants to access .NET objects from your ABL applications, especially for building an OpenEdge GUI for .NET. You should be familiar with object-oriented programming and .NET concepts, and you should also be familiar with object oriented programming using ABL. You can find a description of ABL support for object-oriented programming in OpenEdge Development: Object-oriented Programming. The present manual and related ABL documentation describes .NET objects and their members from an ABL perspective. However, you need to review the native .NET documentation to fully understand how related .NET objects and their members function within a .NET class hierarchy. While doing so, this manual should provide the information necessary for you to interpret any native .NET documentation in an ABL context.

Organization

Overview on page 19

Introduces ABL features and requirements for accessing .NET objects and using the OpenEdge GUI for .NET, describes the supporting object model and architecture, and presents a simple GUI for .NET example.

Accessing and Managing .NET Classes from ABL on page 29

Describes the basic ABL elements for referencing, instantiating, and accessing the members of .NET classes, including event handling, exception handling, and working with ABL-extended .NET classes.

Creating and Using Forms and Controls on page 89

Describes how to access and work with .NET forms and controls in ABL, presenting the ABL foundations for using the OpenEdge GUI for .NET.

Binding ABL Data to .NET Controls on page 131

Describes how to bind ABL data to .NET controls in the GUI for .NET using an OpenEdge extension of the .NET `System.Windows.Forms.BindingSource` class.

Using .NET Forms with ABL Windows on page 183

Describes the features that allow .NET forms to coexist more naturally with ABL windows in the same application.

OpenEdge Installed .NET Controls on page 201

Lists the .NET controls that are installed as visual design components for the Visual Designer in Progress Developer Studio for OpenEdge, including OpenEdge Controls, Microsoft .NET UI Controls, and the OpenEdge Ultra Controls for .NET. This appendix also provides references to third-party online documentation on all .NET objects supported by OpenEdge that are provided by the Microsoft .NET Framework and the Infragistics NetAdvantage for .NET.

Using .NET data types in ABL on page 205

Describes how to access and manage .NET data types in ABL, including implicit and explicit mappings between .NET and ABL primitive types, and working with .NET value types, reference types, boxing, enumerations, arrays, and generic types. This appendix thus serves as a detailed resource for understanding .NET data types as introduced and referenced in the chapters of this book.

Notes on using the ProBindingSource on page 251

Expands upon the information in Binding ABL Data to .NET Controls on page 131. The details provided here further describe how you can work efficiently and successfully with the ProBindingSource.

Using ABL documentation

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

For the latest documentation updates go to https://docs.progress.com.

References to ABL compiler and run-time features

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

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

References to ABL data types

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

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

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

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

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

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

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

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

Syntax

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

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

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ 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 . . . ] }
```

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**.

1

Overview

This OpenEdge® release supports the ability to build ABL applications that use Microsoft® .NET classes much like native ABL classes. Thus, from any ABL class or procedure, you can instantiate and access members of a .NET class in much the same way as you would the members of an ABL class. A key feature of ABL support for .NET objects is the *OpenEdge GUI for .NET*, which allows you to build graphical user interfaces (GUIs) using .NET forms and controls.

With .NET forms and controls, the OpenEdge GUI for .NET allows you to build richer and more flexible user interfaces than the traditional OpenEdge GUI. However, you can use as little or as much of the GUI for .NET as you choose, and you can freely mix .NET forms of the GUI for .NET and ABL windows of the traditional GUI in the same application. This means that you can add GUI for .NET features to an existing application without having to change or remove any of its traditional GUI features.

One of the benefits of the traditional OpenEdge GUI is the ease with which ABL statements can display and allow updates to ABL data using user-interface field and table-level widgets. The OpenEdge GUI for .NET allows you to bind ABL data to .NET controls using an OpenEdge extension of the .NET `System.Windows.Forms.BindingSource` class, Progress.Data.BindingSource (ProBindingSource). Using the ProBindingSource, you can support the display and update of ABL data in .NET controls with an ease that is comparable to using traditional ABL data display and update capabilities.

For details, see the following topics:

- General capabilities and limitations
- Object model and architecture
- Simple example

General capabilities and limitations

In order to access .NET objects, OpenEdge requires .NET Framework 4.0 to be installed on your system. (The OpenEdge installation automatically installs the .NET Framework 4.0, if needed.) Otherwise, OpenEdge supports all ABL application development using .NET objects entirely with OpenEdge tools, especially using Progress Developer Studio for OpenEdge.

Capabilities

OpenEdge support for using the .NET Framework includes the following general capabilities:

- You can access .NET objects from any ABL session running on Windows, whether it is a GUI or non-GUI session. However, each type of ABL session has certain limitations. For more information, see Limitations of support for .NET classes on page 31.
- ABL supports some features of .NET objects that are not supported for ABL class-based objects. For more information, see Limitations of support for .NET classes on page 31.
- ABL allows you to use .NET classes like ABL classes. Thus, you can instantiate .NET classes within a procedure, user-defined function, or method of an ABL class. For more information on instantiating .NET classes, see Instantiating and obtaining instances of .NET classes on page 43.
- ABL allows you to create ABL classes that extend .NET classes (*ABL-extended .NET classes*) similar to extending ABL classes—by inheriting a .NET class or by implementing .NET interfaces in an ABL class definition. You can also define ABL interfaces that inherit the member prototypes from .NET interfaces. For more information on extending .NET classes, see Defining ABL-extended .NET objects on page 70.
- ABL allows you to access .NET class members using ABL data types. ABL maps the ABL data types you specify to the appropriate .NET data types, depending on .NET requirements and how you access a given class member.
- OpenEdge allows ABL to catch .NET exception objects and treat them as `Progress.Lang.Error` objects. For more information, see Handling .NET exceptions on page 65.
- OpenEdge provides its own set of .NET classes and interfaces to facilitate ABL manipulation of .NET forms, and to allow .NET forms to work more naturally together with ABL windows. OpenEdge also provides a set of .NET controls with extended capabilities (OpenEdge Ultra Controls for .NET) that you can use with .NET forms and that Progress Developer Studio for OpenEdge supports as visual design components using Visual Designer. For more information, see OpenEdge .NET form and control objects on page 90.
- OpenEdge provides its own set of .NET classes for binding ABL data to .NET controls. For more information, see Binding ABL Data to .NET Controls on page 131.
- ABL allows you to use both .NET forms and ABL windows in the same application, and to manage them in a common manner. ABL also allows you to embed the ABL frame or frames contained by an ABL window (including the field-level widgets they contain) within the client area of a .NET form. For more information, see Using .NET Forms with ABL Windows on page 183.

Limitations

OpenEdge support for using the .NET Framework includes the following general limitations:

- While you can use both .NET forms and ABL windows together in the same application, you must work with each type of object using the features of its native object model, even when you embed the frames of an

ABL window in a .NET form. However, while you can embed frames from ABL windows in .NET forms, you cannot embed .NET controls in ABL windows.

- Although ABL allows you to use .NET objects, ABL is not a full .NET language in the sense of Visual Basic .NET. In particular, ABL is not a Common Language Specification (CLS)-compliant language. Instead, the .NET Common Language Runtime (CLR) functions as an embedded component that makes .NET objects available to ABL, but does not directly access ABL objects from .NET. For more information, see Object model and architecture on page 21. For more information on the .NET Common Language Specification and Common Language Runtime, see the documentation on these topics at:

  http://msdn2.microsoft.com/en-us/library/a4t23ktk(VS.80).aspx

- Although ABL provides a comprehensive scheme for mapping ABL and .NET data types, this scheme has some limitations. For example, some data type mappings do not preserve precision. For more information, see Using .NET data types in ABL on page 205.
- You cannot access certain .NET classes; for example, any class that relies on threads. For more information, see Limitations of support for .NET classes on page 31.
- While you can access .NET objects from any ABL session running on Windows, from non-GUI ABL sessions, this access has certain limitations. For more information, see Limitations of support for .NET classes on page 31.
- While you can create ABL classes that extend .NET classes and implement .NET interfaces, ABL has limitations on the kinds of classes and interfaces that you can extend and implement, respectively. For more information, see Limitations of support for .NET classes on page 31.

You can find information on more specific limitations when working with .NET objects from within ABL in the following chapters and sections that describe the corresponding features (see Organization on page 12).

Object model and architecture

ABL uses a unique object model and run-time architecture in order to support access to .NET objects. This object model allows ABL to compile and manage references to .NET classes along with ABL classes. The run-time architecture allows an ABL application to transparently use the functionality of .NET objects almost as though ABL were running as a CLS-compliant language. This section describes how this object model and architecture enable an ABL application to effectively work as a .NET application.

Incorporation of the .NET object model

ABL supports access to .NET objects by essentially incorporating the entire .NET class hierarchy within the ABL class hierarchy. ABL does this by viewing the .NET root class, `System.Object`, as an immediate subclass of the ABL root class, `Progress.Lang.Object`. In this way, every .NET class appears to be a part of the ABL class hierarchy and functions like an ABL class when referenced by another ABL class or procedure.

For example, if you reference the .NET class, `System.Windows.Forms.Button`, in an ABL session, ABL shows it to have the following class hierarchy:

```
Progess.Lang.Object<----------------------------- ABL root class
  System.Object    <----------------------------- .NET root class
    System.MarshalByRefObject                     .NET class hierarchy
      System.ComponentModel.Component             |
        System.Windows.Forms.Control              |
          System.Windows.Forms.ButtonBase         |
            System.Windows.Forms.Button           V
```

Thus, any `System.Windows.Forms.Button` instance inherits the properties and methods of `Progress.Lang.Object`. This means that .NET class instances appear on the ABL session object chain, and you can manage them in much the same way as ABL class instances.

Note: In addition to classes, all supported .NET types, such as interfaces, enumerations, and structures are accessible as defined by the .NET type hierarchy derived from `System.Object`. For more information, see the sections on data type mapping in Accessing and Managing .NET Classes from ABL on page 29.

Similarly, if you derive a .NET class with an ABL class, it appears in the class hierarchy as you might expect:

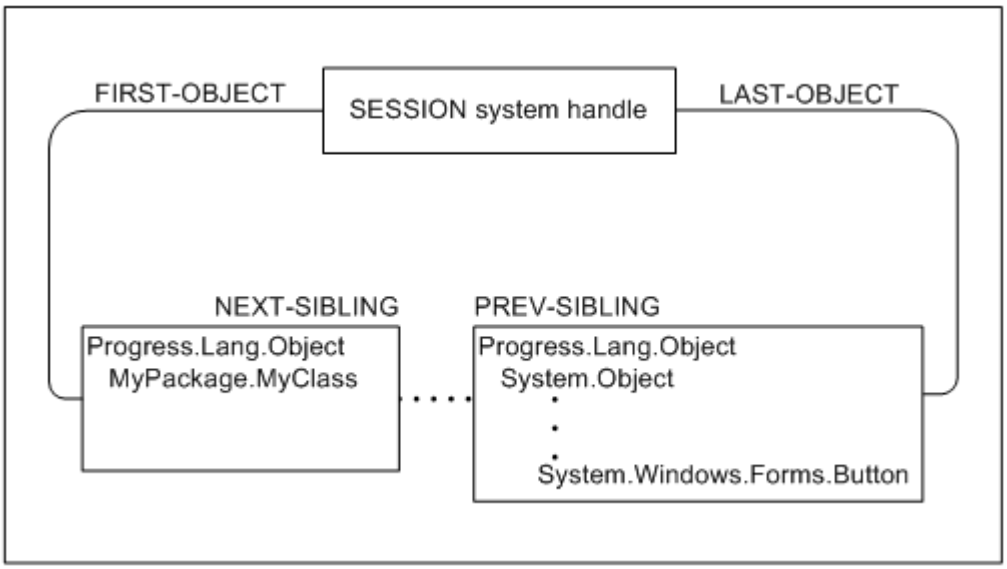
```
Progess.Lang.Object<----------------------------- ABL root class
  System.Object    <----------------------------- .NET root class
    System.MarshalByRefObject                     .NET class hierarchy
      System.ComponentModel.Component             |
        System.Windows.Forms.Control              |
          System.Windows.Forms.ButtonBase         |
            System.Windows.Forms.Button           V
              Acme.Controls.CustomButton           ABL-derived .NET class
```

In reality, such an ABL-derived class exists at run time as both an ABL class in the ABL Virtual Machine (AVM) and as a parallel .NET class in the CLR, hence the reference to an ABL-derived .NET class. Also, if you implement one or more .NET interfaces in an ABL class, that ABL class also exists as an ABL extension of a .NET class in both the AVM and CLR. For more information, see GUI for .NET run-time architecture on page 24.

.NET objects on the session object chain

The following figure shows how the session object chain, which is anchored to the SESSION system handle using the FIRST-OBJECT and LAST-OBJECT attributes, works for .NET objects. You can walk the session object chain for both ABL and .NET objects using the NEXT-SIBLING and PREV-SIBLING properties of Progress.Lang.Object.

Figure 1: .NET objects on the ABL session object chain



Although .NET class instances appear on the ABL session object chain, the actual working instance of each object that ABL references is created in the .NET CLR. (For more information on how this works, see GUI for .NET run-time architecture on page 24.) In addition, because all .NET classes are subclasses of `Progres.Lang.Object`, for any .NET class that has a member with the same name as a corresponding member defined by `Progres.Lang.Object`, the .NET version overrides the `Progres.Lang.Object` version. For example, the `ToString( )` method on the .NET `System.Object` always overrides the ToString( ) method on `Progress.Lang.Object`. Thus, .NET classes in an ABL session follow all the same overriding rules as ABL classes.

Compile-time access to .NET objects

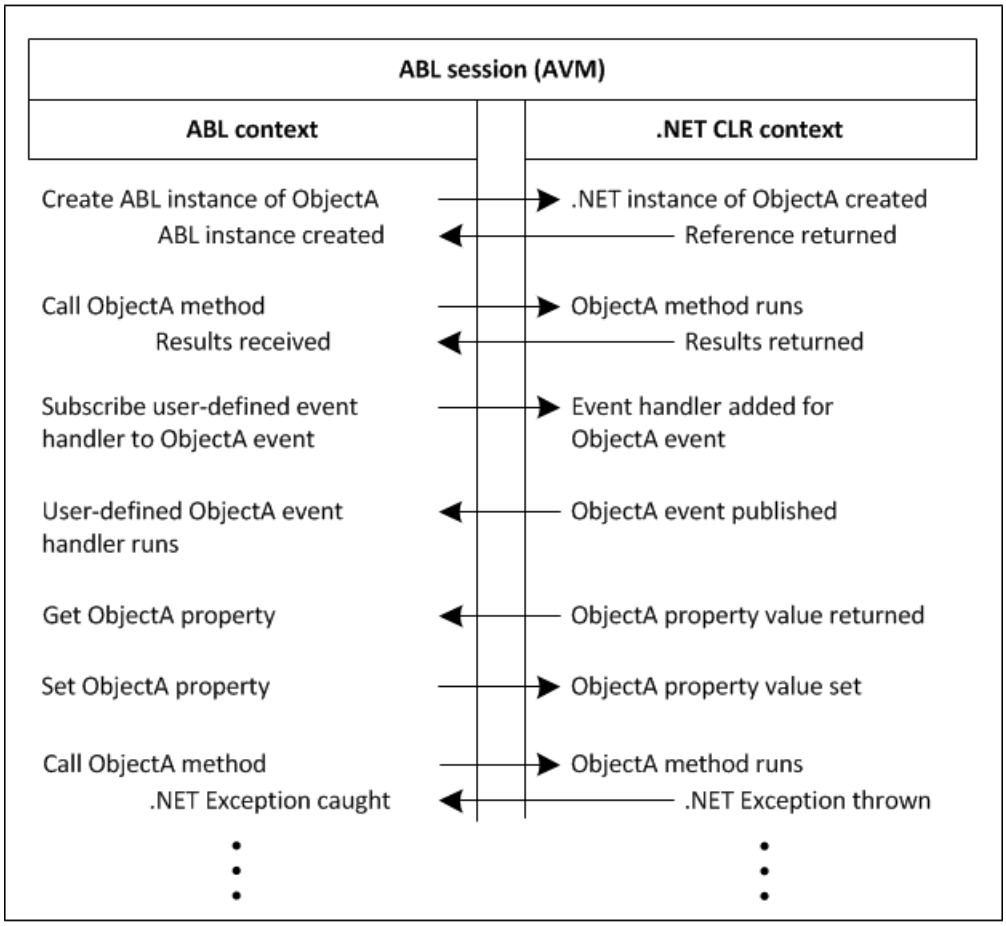
In order for ABL to identify any .NET class or other .NET type you reference, it has to locate the .NET assembly that contains the type definition. In .NET, an assembly is a kind of Windows dynamic link library (DLL) that is specially formatted to uniquely identify the .NET types that it defines. For some supported .NET types, ABL internally knows how to locate the assemblies that define them. For any additional .NET types you want to reference, you must identify the assemblies that define them to ABL using an *assembly references file*. This is an XML file that you can edit using an OpenEdge tool to record the name and identification information for each additional assembly that you require for your application. For more information on assembly references files and how to use them, see Identifying .NET assemblies to ABL on page 35.

Note: You must provide an appropriate assembly references file for both compilation and deployment of an ABL application that accesses .NET objects. For more information on deploying OpenEdge GUI for .NET applications, see OpenEdge Deployment: Managing ABL Applications.

GUI for .NET run-time architecture

The following figure shows the run-time architecture for accessing .NET objects using a sample ABL session instance to illustrate the run-time behavior.

Figure 2: .NET object run-time architecture



The entire session runs in a single ABL Virtual Machine (AVM)—a single `prowin32.exe` process in the Windows Task Manager. The AVM maintains two separate contexts:

- The ABL context, which manages instantiation and execution of ABL procedures and classes
- The .NET CLR context, which is an embedded CLR that manages instantiation and execution of all .NET classes on behalf of the ABL session

The ABL context functions like a client of the .NET CLR context. In other words, all actions initiate from a running ABL procedure or class. When an ABL action involves .NET, such as a reference to a .NET object type or any instantiation of and access to a .NET object and its members, the embedded CLR responds appropriately. When the .NET objects involved are .NET forms and controls, the ABL client then acts as a controller for the CLR-managed view.

So, in the above figure, the sample session starts out by instantiating the .NET class (ObjectA) which could be any accessible .NET object, such as a `System.Windows.Forms.Button`, or an ABL-derived .NET object, such as `Acme.Controls.CustomButton` (see Incorporation of the .NET object model on page 21). When this occurs, the CLR instantiates the working .NET instance and returns its object reference to ABL, which maintains it in an object reference structure. For an ABL-extended .NET class, the ABL object also contains the additional ABL code used to extend the .NET base class or implement a .NET interface. However, to an ABL application with an ObjectA that is an ABL-extended .NET class, both the ABL and .NET instances of ObjectA appear as one, which the ABL application accesses as if it were entirely managed by the ABL context.

ABL supports additional mechanisms required to handle many .NET features that are not supported for ABL classes, such as access to .NET inner classes. This allows an ABL session to appear and function similar to how it might if ABL were a fully CLS-compliant language. The difference consists in the .NET features that ABL does not support and the fact that the embedded CLR does not act on its own, except as initiated by the ABL session. This CLR has no direct knowledge of ABL context, nor can .NET classes interact with pure ABL objects as .NET objects.

Note, also, for any ABL-derived .NET class that overrides a method on the .NET base class, if .NET calls that method on the .NET instance using a super class object reference, the ABL-derived implementation of the method is executed with all results returned to .NET. For example, if ObjectA is an ABL-derived .NET class that extends `System.Windows.Forms.Button`, and ObjectA overrides the .NET `ToString( )` method, it can perform any defined ABL processing and return its own `CHARACTER` value. If some .NET object running in the CLR references the .NET instance of ObjectA as a `System.Windows.Forms.Button` and invokes `ToString( )` on the object, the ABL override of `ToString( )` runs, not the .NET implementation defined for `System.Windows.Forms.Button`. After any ABL processing defined for `ToString( )` completes, the resulting `CHARACTER` value of `ToString( )` is returned to the .NET caller as a `System.String` value.

A similar effect can occur for an ABL class that implements a .NET interface. In other words, if ObjectA is an ABL-extended .NET class that implements a .NET interface. Some .NET object running in the CLR can reference the .NET instance of ObjectA as the .NET interface type that ObjectA implements. When the CLR thus invokes an ABL-implemented .NET method or accesses an ABL-implemented .NET property, it returns the results to the CLR as though these members were implemented directly in the CLR context.

Performance note

When an application operation is running entirely within the ABL context or entirely within the .NET CLR context, performance is generally the same, respectively, as running the same operation in a pure ABL application or in a pure .NET application. However, any operation that exchanges data between the ABL context and the .NET CLR context (such as a property assignment) incurs some performance penalty over a similar operation running entirely within the ABL or .NET CLR context alone. This is because all such data exchanges involve the movement and conversion of data between one or more corresponding ABL and .NET data types.

Note that a typical GUI for .NET application requires many such data exchanges. However an exchange involving a simple primitive data type, such as an ABL `INTEGER` or `DECIMAL`, often incurs an insignificant penalty relative to the native I/O and other processes that drive either the .NET or ABL components of the application. Somewhat larger penalties result from exchanges of larger and more complex data types, such as large ABL `LONGCHAR` values and especially large arrays, temp-tables, and ProDataSets.

A key to minimizing performance penalties in a GUI for .NET application is to reduce the number and size of data exchanges between the ABL and .NET CLR contexts. In general, GUI for .NET applications perform best when you focus all high-volume data processing within one context and move only the final results to the other context—typically, for display in the .NET CLR context and for storage in the ABL context.

Finally, .NET provides certain class-based data structures (such as hash tables and multi-dimensional arrays) that are not natively supported in ABL but that might seem convenient to hold appropriate data originating in ABL. However, before using such .NET data structures for ABL data, consider the possible performance impact of exchanging that data between ABL and the potential .NET data structure, especially if the typical number of ABL data items or data exchanges is relatively large.

Simple example

The following sample procedure, `ShowDateTime.p`, is a simple ABL application that accesses .NET objects. It makes use of some of the basic ABL features that support access to .NET objects, including an OpenEdge .NET class and two from among a set in-the-box controls that OpenEdge installs to support the GUI for .NET.

```
USING Infragistics.Win.Misc.* FROM ASSEMBLY.                          /* 1 */
USING System.Windows.Forms.* FROM ASSEMBLY.

/* Define object reference and data variables */                      /* 2 */
DEFINE VARIABLE rDateForm AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rOKButton AS CLASS UltraButton NO-UNDO.
DEFINE VARIABLE rDateField AS CLASS UltraLabel NO-UNDO.

/* Create objects */                                                  /* 3 */
rDateForm = NEW Progress.Windows.Form( ).
rOKButton = NEW UltraButton( ).
rDateField = NEW UltraLabel( ).

/* Initialize OK button */                                            /* 4 */
rOKButton:Text = "OK".
rOKButton:Size = TextRenderer:MeasureText( "OK",
                                           rOKButton:Font ).
rOKButton:Height = rOKButton:Height + 12.
rOKButton:Width = rOKButton:Width + 12.
rOKButton:DialogResult = DialogResult:OK.
rOKButton:Top = rDateField:Top + rDateField:Height + 4.

/* Initialize current date/time field */                              /* 5 */
rDateField:Text = STRING(System.DateTime:Now).
rDateField:Size = TextRenderer:MeasureText( rDateField:Text,
                                            rDateField:Font ).
rDateField:Top = 2.

/* Initialize dialog with field and button */                         /* 6 */
rDateForm:Text = "Today's Date and Time".
rDateForm:MaximizeBox = FALSE.
rDateForm:MinimizeBox = FALSE.
rDateForm:FormBorderStyle = FormBorderStyle:FixedDialog.
rDateForm:Controls:Add( rDateField ).
rDateForm:Controls:Add( rOKButton ).
rDateForm:AcceptButton = rOKButton.

/* Adjust dialog size and controls for field and button */            /* 7 */
rDateForm:Width = rDateField:Width * 1.5.
rDateForm:Height = rOKButton:Top + rOKButton:Height +
                   ( 2 * rDateField:Height ) + 24.
rDateField:Left = ( rDateForm:Width - rDateField:Width ) / 2.
rOKButton:Left = ( rDateForm:Width - rOKButton:Width ) / 2.

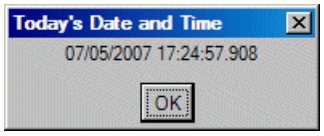
/* Show dialog and wait for button click */                           /* 8 */
WAIT-FOR rDateForm:ShowDialog( ).

rDateForm:Dispose( ).                                                 /* 9 */
```

For information on compiling and running this procedure, see Example procedures.

When you run `ShowDateTime.p`, it creates and displays a dialog box that shows the current date and time, as shown in the following figure.

Figure 3: Simple application accessing .NET objects



The following is a description of the ABL elements used to implement this procedure, numbered according to the numbered comments in the sample code:

1. The USING statements specify .NET namespaces (similar to ABL packages) in which some of the referenced .NET object types are defined. This allows you to reference the object type by its unqualified class or interface name. The `FROM ASSEMBLY` option tells ABL to search .NET assemblies (rather than ABL packages on PROPATH) for the definitions of types defined in these namespaces.

   Note: While sometimes named alike, .NET namespaces and assemblies are not the same thing. Assemblies are the physical files that contain actual type definitions, while namespaces are logical groups of types. Types in the same namespace can be defined in different assemblies, and a single assembly can define types in different namespaces.

2. This code defines object reference variables for the classes used in the procedure. The Progress.Windows.Form class is an OpenEdge .NET class that inherits from the Microsoft `System.Windows.Forms.Form` class and provides a .NET form object that is designed for use in an ABL session. The `UltraButton` and `UltraLabel` classes (from the `Infragistics.Win.Misc` namespace) are two of the Ultra Controls from Infragistics® installed with this OpenEdge release. The `UltraButton` provides a button similar to the Microsoft `System.Windows.Forms.Button` class and the `UltraLabel` provides a fill-in for displaying data, similar to the `System.Windows.Forms.Label` class.

   Note: For more information on `Progress.Windows.Form`, see Creating and Using Forms and Controls on page 89.

3. The NEW function (classes) instantiates these .NET classes just like an ABL user-defined class.
4. To initialize the button object, the procedure assigns ABL data type values to selected `UltraButton` class properties. However, the data type of the `DialogResult` property is the .NET `System.Windows.Forms.DialogResult` enumeration. Thus, the procedure sets this property to the `OK` member of the `DialogResult` enumeration class.

   Note: The `DialogResult` property has the same name as its data type, which is the `DialogResult` enumeration type. Although the names are the same, they do refer to two different things.

5. To initialize the date/time field object, the procedure assigns ABL data type values to selected `UltraLabel` class properties. It gets the current date and time from the static `Now` property of the `System.DateTime` class.

   Note: You can also obtain a similar result more efficiently using the NOW ABL built-in function, which also includes the time zone. However, this example uses the .NET property to demonstrate access to the .NET feature.

6. To initialize the dialog box object, the procedure assigns ABL data type values to selected Progress.Windows.Form class properties, which are inherited from `System.Windows.Forms.Form`. It also adds the `UltraLabel` and `UltraButton` controls to the form by invoking the `Add( )` method on the form `Controls` property. This property has the .NET type, `System.Windows.Forms.Control+ControlCollection`, which is an inner class of the `Control` class. Because the `Form` class inherits `Control`, the form object can use this property to contain a collection of controls that the form can display in its client area.
7. After the form is resized according to the control dimensions, the procedure repositions the controls to center them in the form client area.
8. Once the dialog box object is initialized, the procedure blocks for input using a .NET-specific form of the WAIT-FOR statement for input-blocking .NET methods. The syntax for this `WAIT-FOR` statement invokes the .NET form method, `ShowDialog( )`, which displays the dialog box as well as blocking for input. The statement then blocks until you click the **OK** button or press the **ENTER** key. This causes the button object to publish a `Click` event, during which .NET automatically sets the `DialogResult` property of the form to the enumeration value (`DialogResult:OK`) of the button's own `DialogResult` property. Anything that sets the `DialogResult` property on a form displayed as a dialog box automatically causes the dialog box to close and the `ShowDialog( )` method to return that same value, which also terminates the `WAIT-FOR` statement. If `ShowDateTime.p` needed to check the method return value, the .NET `WAIT-FOR` statement syntax also provides an option (`SET`) that conveniently makes this return value available to the procedure.
9. .NET garbage collects objects in much the same way that OpenEdge garbage collects ABL class instances. However, .NET dialog boxes can be closed in a way that prevents them from being garbage collected in a timely manner. Therefore, to ensure garbage collection of a .NET dialog box object and to avoid the application memory leaks that it can create, you must call the .NET `Dispose( )` method on the object when you no longer need it. Calling this method is unnecessary, however, for most other .NET objects.

The remaining chapters of this book describe all of the features shown here (and more) for working with .NET objects in an ABL session.

Note: You can locate the third-party documentation for the .NET objects referenced in this example as follows. For the OpenEdge .NET class, Progress.Windows.Form, see the reference entry for this class in OpenEdge Development: ABL Reference. For the OpenEdge Ultra Controls for .NET added to the .NET form, see OpenEdge Ultra Controls for .NET on page 202.

2

Accessing and Managing .NET Classes from ABL

ABL supports general syntax and features to access and manage .NET classes. This support for .NET object types is similar to native ABL support for user-defined types. There is also some additional ABL syntax and a set of custom OpenEdge .NET classes that together provide features that are unavailable with ABL user-defined types alone.

To understand how to use a .NET class, you must consult the appropriate .NET documentation. The Microsoft .NET class library documentation specifies the signature of every property and method using the syntax for each of the principle CLS-compliant languages that it supports. You can therefore identify many features of .NET classes using the syntax specified for specific .NET languages. For your general reference, this manual indicates ABL mappings to language-specific .NET features using C# syntax.

For details, see the following topics:

- Supported features and limitations
- Referencing and instantiating .NET classes
- Accessing .NET class members
- Handling .NET events
- Handling .NET exceptions
- Defining ABL-extended .NET objects

Supported features and limitations

ABL access to .NET classes is generally similar to accessing ABL classes (see Referencing and instantiating .NET classes on page 34). However, ABL supports new syntax to handle some features that are particular to .NET.

Support for .NET classes

ABL provides syntax and additional OpenEdge .NET classes to support the following general features of .NET classes and their access in an ABL session:

Note: For information on restrictions, see Limitations of support for .NET classes on page 31.

- **.NET class instances** — ABL allows you to directly instantiate supported .NET classes using the ABL NEW functions and statements. For more information, see Instantiating and managing .NET class instances on page 42.
- **.NET class members** — ABL supports access to both instance and static members of .NET classes, including fields (data members), properties, methods, and events (see Accessing .NET class members on page 45).
- **.NET data types** — ABL supports mappings to most .NET data types, with some limitations (see Using .NET data types in ABL on page 205). This includes support for accessing .NET object types, including the various kinds of class and interface types that .NET supports.
- **Handlers for .NET events** — In a GUI session (WebClient or full GUI client), ABL allows you to process .NET object events by subscribing an ABL class-based method or internal procedure as a handler for a given .NET event. You use the same mechanism for responding to .NET events as you use for responding to ABL class events (see Handling .NET events on page 57).
- **.NET exceptions** — ABL transparently handles .NET exceptions as ABL-raised ERROR conditions or as ABL-thrown error objects. So, .NET exceptions behave within the ABL application like any ABL error (see Handling .NET exceptions on page 65).
- **ABL-extended .NET classes** — ABL allows you to create ABL-extended .NET classes by defining ABL classes that inherit from .NET protected or public classes (including abstract classes) or by implementing .NET public interfaces, which ABL interfaces that you define can also inherit (see Defining ABL-extended .NET objects on page 70).
- **.NET forms and ABL data** — In a GUI session (WebClient or full GUI client), ABL provides extended support for creating, visualizing, and managing .NET forms (see Creating and Using Forms and Controls on page 89) and for binding ABL data to .NET controls on those forms (see Binding ABL Data to .NET Controls on page 131).
- **.NET forms and ABL windows** — ABL allows .NET forms and ABL windows to display and operate in the same ABL session, and also allows the client area of any ABL window to be embedded in the client areas of .NET forms in a manner that allows you to interact with the embedded ABL widgets using native ABL widget management features (see Using .NET Forms with ABL Windows on page 183).
- **.NET sets** — ABL provides support for a variety of .NET controls to use with .NET forms (see OpenEdge Installed .NET Controls on page 201)

Limitations of support for .NET classes

In ABL, you can use many features of .NET classes much as you would in a CLS-compliant language. However, ABL does have the following limitations on the classes you can access and the features you can use:

- **Accessing .NET classes and class members** — From ABL, you can access most .NET classes and class members. However, ABL either does not support certain .NET classes or provides its own syntax to support the features that they provide. The following table lists the specific restrictions on accessing .NET classes and methods and describes any ABL equivalent support. If you attempt to use one of the restricted .NET classes or methods listed in this table, the ABL Virtual Machine (AVM) raises a run-time error.

Restriction	ABL support
You cannot access a .NET type that is defined in the default namespace (with no namespace defined).	ABL can only access .NET types that have a namespace defined for them.
You cannot use an instance of `System.Threading.Thread` or any class derived from it.	Not supported—the AVM is single threaded.
You cannot use an instance of `System.Delegate` or any delegate type derived from it. However, when you implement a .NET abstract or interface event in ABL, you must make reference to a delegate type in order to specify the event signature.	A .NET delegate type (*delegate*) is a special type of class that defines an event handler method for any .NET event that requires it. ABL allows you to associate either a class-based ABL method or an internal procedure as an event handler for a given .NET event. You must reference the documentation for the associated .NET delegate to determine the signature required to define an ABL handler for the event or to publish any .NET abstract or interface event that you implement. For more information, see the .
You cannot call the static method, `System.Windows.Forms.Application:DoEvent( )`, anywhere.	In a .NET application, the `Application:DoEvent( )` method invokes event handlers for all currently published events that have not yet been handled. In ABL, you invoke the PROCESS EVENTS statement to handle currently published events. For more information, see Handling .NET events on page 57.
You cannot call the static method, `System.Windows.Forms.Application:Run( )`, outside of a `WAIT-FOR` statement.	In a .NET application, the `Application:Run( )` method blocks for non-modal form input. In ABL, you must invoke this method within a WAIT-FOR statement to block for non-modal form input. For more information, see Blocking on non-modal forms on page 100.

Restriction	ABL support
You cannot call the instance method, `System.Windows.Forms.Form:ShowDialog( )`, outside of a `WAIT-FOR` statement.	In a .NET application, the `ShowDialog( )` method blocks for input in the modal form instance (dialog box) on which you call it. In ABL, you must invoke this method within a WAIT-FOR statement to block for a dialog box. For more information, see Blocking on modal dialog boxes on page 103.
You cannot call a .NET generic method.	You can call non-generic methods on a .NET generic class or interface object. However, .NET generic methods use a separate mechanism that ABL does not support.

- **Extending .NET classes** — You cannot define an ABL class that inherits from a .NET class that is itself defined with any of the following .NET directives (as specified in C#):
  - `internal`
  - `private`
  - `sealed`
  - `static`

  You cannot define an ABL class that inherits from a .NET generic class.

  You cannot define an ABL class that can become an additional part for a .NET `partial` class.

  You cannot override the following methods of an inherited .NET class:

  - `DestroyHandle( )`
  - `Dispose( )`
  - `Finalize( )`
  - `GetHashCode( )`

  If your non-abstract ABL class inherits from a .NET abstract class, it must implement (override) all abstract properties, methods, and events of the inherited class. However, you **cannot** inherit from a .NET abstract class that defines an abstract indexed property (including a default indexed property), because ABL has no support for overriding and defining indexed properties. For more information on these restrictions, see Deriving .NET classes in ABL on page 72.

  An ABL class cannot inherit from the following .NET class types or any class types that are derived from them:

  - `System.Delegate`
  - `System.Enum`
  - `System.Threading.Thread`
  - `System.ValueType`

  For more information on these restrictions, see Features of ABL classes that implement .NET interfaces on page 85.

- **Implementing .NET interfaces** — You cannot implement a .NET interface that:
  - .NET defines as `internal`
  - .NET defines as generic or that, itself, specifies a generic method
  - Defines an indexed property (including a default indexed property)

  Also note that ABL has no support for explicit interface members. If you implement more than one .NET interface that defines a property or method with identical signatures, these identical properties or methods must all share the same implementation.

- **Blocking for events** — If you use any non-modal .NET forms in your application, you can simultaneously execute only one WAIT-FOR statement at a time that invokes a non-modal .NET input-blocking method (such as `System.Windows.Forms.Application:Run( )`) to process events on all simultaneously open non-modal .NET forms. You can also use this same `WAIT-FOR` statement to simultaneously process events for non-modal ABL windows and non-GUI features that are active in the ABL session, such as sockets and asynchronous remote procedure calls. For more information, see ABL support for managing .NET forms and controls on page 98.

  Note: You can use as many `WAIT-FOR` statements as required, anywhere in your application, to block on modal .NET or ABL dialog boxes.

- **.NET objects in non-GUI ABL applications** — While you can access .NET objects in non-GUI ABL sessions, you cannot visualize .NET forms and controls in a non-GUI session. You also cannot execute the .NET `WAIT-FOR` statement or otherwise block for and handle .NET object events in a non-GUI session. The ABL sessions with these limitations include:
  - Character-mode clients
  - Batch clients
  - AppServer agents
  - WebSpeed agents

  However in these sessions, you can use all other supported features of .NET objects.

- **Publishing .NET events** — You cannot directly publish (send or fire) .NET object events from ABL. Each .NET object is responsible for publishing its own events, which you can then handle in ABL using event handlers. However, if a .NET object provides protected or public methods for publishing events programmatically on behalf of the object, you can also invoke these methods from ABL to publish the specified events. The use of these .NET methods in ABL is analogous to using the APPLY statement to publish ABL events "to" a specified widget or handle. If you inherit and implement an abstract .NET event in ABL, you can publish the event directly in the implementing ABL class using the ABL Publish( ) event method.

- **.NET operator overloading** — In CLS-compliant languages, .NET classes support the ability to overload operators of the language, like plus (+), for a given class. This overloading allows a function to be defined and executed for the operator when the operator is used with instances of a given class, such as adding two objects together (`Obj1 + Obj2`). In ABL, for most .NET classes, you can only access the functionality of .NET overloaded operators using .NET reflection on a class. However, ABL does provide a helper class (Progress.Util.EnumHelper) that has static methods for performing standard operations on enumeration types. For more information on enumeration types, see Accessing and using .NET enumeration types on page 228.

  Note: ABL does not support operator overloading for user-defined classes.

- **Casting .NET data types** — As with ABL user-defined types, you can cast references to .NET object types, including .NET generic types, using the ABL built-in CAST function. However, unlike CLS-compliant languages, you cannot include .NET primitive types in this ABL type casting.
- **.NET classes and OpenEdge startup options** — .NET classes do not adhere to OpenEdge run-time startup parameters or `.ini` file settings. You must control such options using general Windows settings or in ABL through programmatic control of each .NET object that you create.
- **Accessing Progress.Lang.Class on .NET objects** — Although all .NET classes in ABL extend `Progress.Lang.Object`, you cannot access `Progress.Lang.Class` from a .NET object using the `GetClass( )` method inherited from `Progress.Lang.Object`. On a .NET object, any attempt to call the `Getclass( )` method on a .NET object returns the `Unknown value (?)`. To do reflection on a .NET object, use the .NET native reflection mechanism, in particular, the properties and methods of `System.Type`.

In addition, there are more specific limitations in the syntax and behavior of some of the supported features. This documentation notes these limitations for each feature where they apply.

Referencing and instantiating .NET classes

You can reference and instantiate supported .NET classes in exactly the same way as ABL user-defined classes. And you can reference .NET object types, generally, in the same way as ABL user-defined types. However, the mechanism for finding .NET object types is very different. ABL locates ABL user-defined types on `PROPATH` while .NET locates all .NET types in assemblies. Therefore, in order for ABL to locate the .NET types that you reference, you must sometimes provide additional information to ABL that identifies the assembly where a particular .NET type is located.

Thus, the following sections describe:

- How .NET organizes types for reference on page 34
- Identifying .NET assemblies to ABL on page 35
- Referencing .NET class and interface types on page 37
- Using case sensitivity with .NET objects on page 41
- Using unqualified .NET type names on page 41
- Instantiating and managing .NET class instances on page 42

How .NET organizes types for reference

ABL and .NET organize their class-based type hierarchies using different mechanisms. ABL organizes user-defined types into packages (which correspond to physical directory paths on `PROPATH`) and stores the types in class definition (`.cls`) files (which you compile into r-code (`.r`) files). Packages allow user-defined types to be both associated in a hierarchy and uniquely identified. To locate an ABL user-defined class or interface during compilation, ABL must find its class definition file within a specified package relative to `PROPATH`. And to access a compiled ABL class or interface at run time, ABL must find the corresponding r-code file in the same package as the class definition file from which it is compiled. For more information on how ABL organizes user-defined types using packages, see OpenEdge Development: Object-oriented Programming.

.NET, on the other hand, organizes its object types into namespaces and stores the types in assembly files (*assemblies*). An assembly can be a dynamic link library (DLL) or executable (EXE) file that is specially formatted to store one or more types associated with their individual namespaces. A .NET *namespace* is analogous to an ABL package in that it provides a means to organize and uniquely identify types, but instead of being associated with an actual directory structure where the type definition is stored, a namespace is a completely logical construct that is associated with each type that is stored in an assembly. Assemblies provide several features for .NET types, including versioning, security, and localization support, all of which are incorporated into the format of the name used to identify the assembly. Also, the assembly where a given .NET type is defined can reside in any one of several locations on your system, and you must have access to this assembly both to compile and run an ABL application that references that .NET type. For more information on .NET namespaces and assemblies, see the Microsoft .NET documentation.

From the viewpoint of ABL programming, you code ABL packages for ABL user-defined type references and namespaces for .NET type references using the same syntax, and they can appear in most of the same coding contexts (see Referencing .NET class and interface types on page 37). In addition, for .NET type references, you sometimes have to use an external tool to explicitly identify the assemblies where .NET types are stored in order to compile and run ABL code that references them.

Identifying .NET assemblies to ABL

To compile and run an ABL class or procedure that instantiates a .NET class or references any .NET object type, ABL must be able to identify and locate the assembly where the specified .NET object type is implemented. For any ABL application that accesses .NET objects, the AVM automatically loads certain .NET assemblies, including the appropriate versions of:

- **`Progress.NetUI.dll`** — The assembly where all custom OpenEdge .NET classes reside
- **`Mscorlib.dll`** — The assembly where all core Microsoft .NET classes reside
- **`System.Windows.Forms.dll`** — The assembly where all Microsoft form and control classes reside
- **`System.Drawing.dll`** — The assembly where all Microsoft graphics classes reside

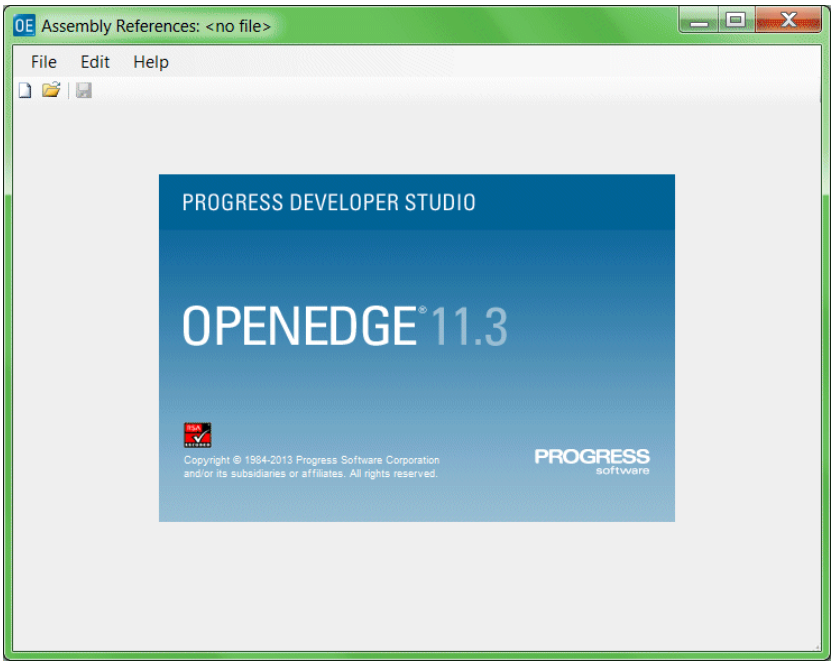
ABL can therefore locate any .NET type that you reference from these assemblies without any further work on your part.

However, for any .NET type that you reference that is not in one of these assemblies, you must explicitly identify the appropriate assembly to ABL using an *assembly references file*. This XML file, named `assemblies.xml`, is required to compile and run your application. This file must appear either in your working directory (the top-level directory for each project in Progress Developer Studio for OpenEdge) or in the directory you specify using the Assemblies (`-assemblies`) startup parameter. Progress Developer Studio for OpenEdge provides an **Assemblies** dialog in the project properties dialog box (click **Properties** > **Project**) and a separate Assembly References tool to create and edit assembly references files, as you require.

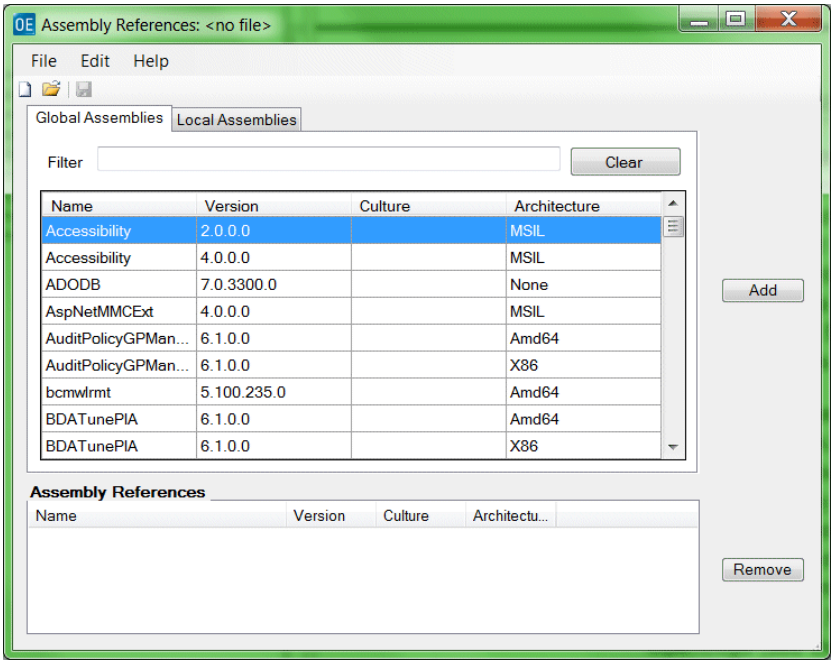
To create and edit an assembly references file using the Assembly References tool:

1. Run the Assembly References tool in one of the following ways:
   - In Progress Developer Studio for OpenEdge, click **Assembly References** on the **Tools** > **OpenEdge** menu.
   - In the Procedure Editor, click **Assembly References** on the **Tools** menu.
   - On a Proenv command line, enter the `proasmref` command.

   The utility opens, as shown:



2. To create a new assembly references file, select **File**→ **New** from the menu bar or click the New icon on the toolbar. An **Assembly References** screen appears:



A **Global Assemblies** tab displays a list box that you can filter, which shows all assemblies that are currently registered in the .NET Global Assembly Cache (GAC). All Microsoft .NET assemblies in the .NET Framework are installed in the GAC, as are most third-party assemblies that you might install, including the Infragistics assemblies that support the OpenEdge GUI for .NET. If you create or otherwise obtain a .NET assembly that is not registered in the GAC, you can access its DLL or EXE file using the **Local Assemblies** tab, which allows you to browse all the files on your system, with or without filtering.

3. On the **Global Assemblies** tab, you might first scroll to the **Infragistics4.Win.Misc.v13.1** assembly and add it by selecting the assembly and clicking the **Add** button. The version of Infragistics reads 13.1.20131.2015. The selected assembly (**Infragistics4.Win.Misc.v13.1**) appears in the **Assembly References** list box below the **Global Assemblies** tab. Note that there might be multiple versions of the assembly listed in the displayed GAC. For a .NET Framework assembly, you typically select the latest assembly version that is supported by the .NET Framework version, in turn, supported by your release of OpenEdge (see General capabilities and limitations on page 20). Otherwise, you must ensure that you are using the appropriate version of any assembly that you select for your application.

   Note: You can add multiple assemblies at one time by scrolling to and group-selecting the assemblies you want to add before clicking the **Add** button.

4. When you have added all the assemblies that you need, you can save the file to a location on your `PROPATH` by choosing the **File→ Save** or **File→ Save As** menu item. Once a new file is saved, the filename replaces **<no file>** in the title bar.

   Note: The Assembly References tool allows you to save an assembly references file by any filename you choose, and you might use different working filenames or store `assemblies.xml` files in different directories to manage assembly references required by different applications or application versions. However, ABL only recognizes and loads the `assemblies.xml` file that is in your current working directory or as in the directory specified by the `-assemblies` startup parameter.

   You can also open an existing assembly references file to view or edit by choosing the **File→ Open** menu item, or you can create another new file by choosing the **File→ New** menu item. In either case, the tool checks to make sure you have saved any unsaved changes to the current file before proceeding.

   When editing a file, you can remove assemblies in much the same way that you add them, by selecting them in the bottom panel and clicking **Remove**, or by double-clicking the assembly you want to remove. In this case, the selected assembly disappears from the bottom list box.

   Note: You can remove multiple assemblies at one time by scrolling to and group-selecting the assemblies you want to remove in the bottom panel before clicking the **Remove** button. In this case, all the selected assemblies disappear from the panel.

   For more information on the options of the Assembly References tool, see the tool's online help.

   As noted, you need an appropriate assembly references file for both compiling and running ABL code that references .NET types. This means that you must also deliver the assembly references file with any OpenEdge GUI for .NET application that you deploy. For more information on GUI for .NET application deployment, see OpenEdge Deployment: Managing ABL Applications.

Referencing .NET class and interface types

In ABL, you can reference .NET class and interface types (object types) using much the same syntax required for referencing ABL class and interface types (see the Type-name syntax reference entry in OpenEdge Development: ABL Reference). This includes references to unqualified type names with an appropriate USING statement (see Using unqualified .NET type names on page 41). You can reference several different supported kinds of .NET classes, including (but not limited to) structures, enumerations, and delegates.

ABL also provides additional syntax support for referencing .NET nested (*inner*) types, which typically include inner classes or interfaces, and it uses existing syntax to reference .NET array types and .NET generic types. The following sections describe how to reference ABL-supported .NET types that are unique to .NET (do not exist as native ABL object types). For more information on accessing and working with all .NET types supported in ABL, see Using .NET data types in ABL on page 205.

Referencing inner classes and interfaces

A .NET type can define one or more nested (*inner*) types as members, typically as inner classes or (more rarely) as inner interfaces. Inner classes are typically defined and often instantiated only within the context of the defining class, and inner interfaces are typically defined only within the context of a defining interface. However, inner classes and interfaces support the same features as a normal class or interface, and an ABL application can reference any .NET inner classes and interfaces that are public.

You can reference the full type name for an inner class or interface using the following syntax for combining the defining class or interface type name and a member inner class or interface name:

Syntax

```
type-name+inner-class-or-interface-name
```

The *type-name* is any supported .NET class or interface type and *inner-class-or-interface-name* represents the name of a given inner class or interface that is defined within its defining class or interface.

In .NET languages, the type name of a class and the type name of one of its inner classes are separated by a dot (`.`) instead of a plus (`+`), as in ABL. The same is true for interface type names and the names of their defined inner interfaces. When you look up an inner class or interface in the Microsoft .NET class library documentation, note that its name appears under the same namespace and follows the name of its defining class or interface. You can identify an inner class or interface in this list because it is specified using the *class-or-interface-name*`.`*inner-class-or-interface-name* syntax, where *class-or-interface-name* is the name of the **defining** class or interface and *inner-class-or-interface-name* is the name of the **defined** inner class or interface. Normally (for a non-inner class), no dot (`.`) appears in the class name.

Note that whether you can instantiate or only reference an existing instance of a public inner class depends on the semantics of the defining class. For more information, see the .NET documentation for nested classes of the defining class.

Referencing arrays of objects

In .NET, all arrays are object types that inherit from the `System.Array` class and can have any number of dimensions. Therefore, you can reference all .NET arrays as instances of either `System.Array` or the particular derived array object type. Note that whatever array type you use to access a .NET array, the class members for setting and getting the array elements access the elements as objects of type `System.Object` (the .NET root class). So, ABL also supports mechanisms to access the underlying type of these `System.Object` array elements.

For example, the following statement defines an object reference to a two-dimensional .NET array of `Button` objects:

```
DEFINE VARIABLE rButtonArray
  AS CLASS "System.Windows.Forms.Button[,]" NO-UNDO.
```

Note that in a .NET array type name, the double brackets (`[]`) specify the array dimensions. An empty pair of brackets specifies an array of one dimension, and you add as many embedded commas as necessary to specify the number of additional dimensions for the array type. Also, in ABL, you must use the surrounding double-quotes as part of the type name in order to allow the bracket characters in the name. Unlike an ABL array (which is not an object), a .NET array type name does **not** include the number of elements (the extent) in each dimension of the specified array. You specify .NET array extents when you create the array object at run time.

You can also assign directly between any one-dimensional .NET array and an ABL array (`EXTENT` variable) of compatible element type and extent. ABL thus supports the implicit mapping of array element types when you assign between compatible ABL and.NET arrays.

For more information on .NET arrays and working with them in ABL, see Accessing and using .NET arrays on page 235.

Referencing .NET generic types

.NET supports a concept of generic (or parameterized) types. A .NET *generic type* is any .NET class or interface type that uses *placeholders* for data type names in order to define one or more members or local variables as part of the generic class or interface definition. You determine the actual data types (primitive or object types) for these placeholders by how you reference the generic type name of the class or interface, which includes these placeholders as *type parameters*. Wherever you reference the generic type name, you substitute each type parameter in the name with an actual data type name. Thus, one generic class actually has multiple implementations based on the possible sets of substitution data types that you can provide for the type parameters in the generic class name. ABL supports references to .NET generic type names using this syntax:

Syntax

```
"namespace.object-name<Tparm [ , Tparm] . . .>"
```

The *namespace* is any required .NET namespace. The remaining construction specifies the .NET class or interface name for the generic type, where *object-name* is some identifier and *Tparm* is a type parameter, of which there can be more than one. The number of type parameters and the data types you can specify for each one depend on the generic type definition, which can include constraints for each type parameter. Note that ABL requires the surrounding quotes (`""`) to allow for the required angle brackets (`<>`) and any spaces in the type name. Note also that any number of spaces between type parameters, commas, and angle brackets are allowed but optional.

When you reference the generic type name for a given implementation of the type, you replace each defined *Tparm* in the name with an appropriate data type for the chosen implementation. This data type must explicitly identify the .NET type that the generic type requires for a given type parameter. Some .NET data types, which represent the primitive data types of any .NET language, map directly to ABL primitive data types, and are therefore referred to as *.NET mapped data types*. For example, a .NET `System.Int32` maps to an ABL `INTEGER`. For these .NET mapped types, you must specify an ABL data type to identify the corresponding .NET data type. For information on how ABL primitive types map to .NET types, see Implicit data type mappings on page 210.

Note that some ABL primitive types correspond to more than one .NET mapped type. To specify a particular .NET mapped type, ABL provides extended type keywords called *AS data types* that identify a particular .NET mapped type. For example, to identify a .NET `System.Int16`, which implicitly maps to an ABL `INTEGER`, you must substitute the `SHORT` AS data type for the corresponding *TParm* in a generic type name. For more information on making explicit references to .NET mapped types, see Explicit data type mappings on page 216.

Note: Explicit references to .NET mapped types are required when working with several .NET features in ABL.

For references to .NET types other than .NET mapped types (all other object types), you can directly substitute the .NET object type name (for example, `System.Drawing.Point` or `System.Windows.Forms.Button`) for the corresponding *`TParm`*. You can even substitute another .NET generic type for a *`TParm`*, if the specified generic type supports it.

For example, .NET supports a generic stack object that you can use to create a stack of any .NET type. If you want to define an object reference to a .NET stack of `System.Int16` objects, you can code the following ABL statement:

```
DEFINE VARIABLE rSHORTStack AS CLASS
  "System.Collections.Generic.Stack<SHORT>" NO-UNDO.
```

If you want to define an object reference to a .NET sorted list of key/value pairs (sorted by the key) consisting of `System.String` keys and `System.Int16` values, you can code the following ABL statement:

```
DEFINE VARIABLE rKeyValueList AS CLASS
  "System.Collections.Generic.SortedList<CHARACTER, SHORT>" NO-UNDO.
```

Again, the class type names for these .NET generic types in ABL are `"System.Collections.Generic.Stack<SHORT>"` and `"System.Collections.Generic.SortedList<CHARACTER, SHORT>"` (respectively), and any reference to them in ABL must be surrounded in quotes to allow angle brackets and spaces in the name. Like any type name, you can also make unqualified references to them with an appropriate USING statement, for example, `"SortedList<CHARACTER, SHORT>"`. Note that you can also use abbreviated ABL type names to specify the *`Tparm`* substitution types in the generic type name, for example, `"SortedList<CHAR, SHORT>"`.

Referencing a generic type like this is referred to as *constructing* the type. Therefore, a reference to a .NET generic type that specifies a particular set of data types for its type parameters is referred to as a reference to its *constructed type* name. The .NET documentation specifies the type parameters for a given generic type using a *generic type definition*, which .NET also refers to as an *open generic type*, or simply an *open type*. The open type name for a generic type indicates the number and order of its type parameters. The complete definition for an open type also indicates possible substitute data types (constraints) for its type parameters and other information, such as any interfaces that a generic class type implements.

Note that ABL does not support open types and has no support for defining generic types of its own. ABL supports references only to constructed type names for a .NET generic type. You can thus reference a given generic type in ABL for all uses of .NET types, **except** to:

- Inherit from a .NET generic class
- Implement a .NET interface
- Instantiate a generic class using the DYNAMIC-NEW statement or the New( ) method.
- Cast an object reference to a .NET generic type using the DYNAMIC-CAST function

For example, you cannot specify a .NET generic type for the `INHERIT` or the `IMPLEMENTS` options of the ABL CLASS statement or dynamically cast to or instantiate a generic class. However, you can cast an object reference to a generic type using the CAST function and instantiate a generic class using the NEW function. For more information on inheriting .NET classes and implementing .NET interfaces, see Defining ABL-extended .NET objects on page 70.

You can also define both .NET and ABL arrays of generic types, and .NET generic types can take .NET array objects as constructed type parameters. For example:

```
DEFINE VARIABLE rSHORTListArrayObj AS /* .NET array of a generic type */
  CLASS "System.Collections.Generic.List<SHORT>[]" NO-UNDO.
DEFINE VARIABLE rSHORTListArrayExt AS /* ABL array of a generic type */
  CLASS "System.Collections.Generic.List<SHORT>" EXTENT 3 NO-UNDO.
                                      /* .NET array as a constructed */
DEFINE VARIABLE rButtonArrayList AS CLASS /* type parameter              */
  "System.Collections.Generic.List<System.Windows.Forms.Button[]>" NO-UNDO.
```

In the .NET Framework, the `System.Collections.Generic` namespace defines many of the generic types in the Framework. However, generic types are also defined in several other .NET object namespaces.

For more information on referencing and working with .NET generic types in ABL, including how to identify the available data types you can substitute for generic type parameters from open types listed in .NET documentation, see Working with .NET generic types on page 232.

Using case sensitivity with .NET objects

.NET languages are generally case sensitive with respect to all names and identifiers used in the language. However, because ABL is generally **not** case sensitive, it minimizes the need to respect case sensitivity when programming with .NET objects. Thus, ABL requires that you specify all qualified and unqualified .NET type names using the correct letter case on the first reference only. After that, you can specify the type name using any letter case.

So, all statements that take a .NET type name, such as the following, must use the case sensitive name if they make first reference to that type name:

- `CAST (rObj, System.Windows.Forms.Panel).`
- `DEFINE VARIABLE rObj AS CLASS System.Windows.Forms.Panel NO-UNDO.`
- `DEFINE PARAMETER rObj AS CLASS System.Windows.Forms.Panel NO-UNDO.`
- `USING System.Windows.Forms.* FROM ASSEMBLY.`
- `rObj = NEW System.Windows.Forms.Panel( ).`

Unlike .NET type names, all ABL access to the members of a .NET type are case **insensitive**. If multiple non-overloaded members of a .NET type have the same name, differing only by letter case, ABL finds only the first one that it encounters in the type definition. For example, if a .NET class has two non-overloaded properties, `Foo` and `foo`, ABL always finds `Foo` or `foo`, but not both, depending on which one it encounters first. Typically, this does not present a problem for ABL applications, because Microsoft strongly recommends that no class should have members that differ only by the letter case of their names.

Using unqualified .NET type names

ABL provides the USING statement, which allows you to specify an ABL class or interface type by its unqualified type name, which is the class name or interface name, alone, specified without any defined package. This statement also allows you to specify unqualified .NET type names, which are .NET class or interface names specified without the defined namespace:

Syntax

```
USING type-spec [ FROM { ASSEMBLY | PROPATH } ]
```

The *type-spec* represents one of the following:

- A single fully qualified ABL user-defined or .NET type name that you want to be able to specify in your code using its unqualified type name.
- An ABL package name (appended with `.*`), all of whose defined types you want to be able to specify in your code using their unqualified type names. Note that this cannot be a partial package name representing only the first few of several package components in the package name. The specified components must fully represent the package that contains the user-defined types that you want to reference with unqualified names.
- A .NET namespace (appended with `.*`), all of whose defined types you want to be able to specify in your code using their unqualified type names. Note that, similar to ABL package names, you must specify a complete namespace that contains the object types whose unqualified names you want to reference. You cannot reference type names using nested namespaces within a given namespace as supported for CLS-compliant languages.

The *type-spec* can also represent a fully qualified .NET generic type or generic type namespace. Also you can use a namespace or fully qualified type name in one `USING` statement to resolve an unqualified type name specified for a type parameter of the generic type in a subsequent `USING` statement or other ABL statement.

The `FROM` option tells ABL where to search for the types that you reference according to this statement. If you specify `ASSEMBLY`, ABL treats *type-spec* as a .NET type or namespace reference, and searches for a given unqualified type name only in the available assemblies (see Identifying .NET assemblies to ABL on page 35). If you specify `PROPATH`, ABL treats *type-spec* as an ABL user-defined type or package reference, and searches for a given unqualified type name using only the specified package on `PROPATH`.

If you do not specify the `FROM` option in the `USING` statement, ABL resolves unqualified type names by combining them with the *type-spec* in the following search order until it finds a match with:

1. A built-in ABL class or interface type
2. An ABL user-defined type
3. A .NET type

For more information and examples of `USING` statements for qualifying .NET types, see the USING statement reference entry in OpenEdge Development: ABL Reference.

Instantiating and managing .NET class instances

As noted previously, ABL allows you to program with .NET classes much as you do with ABL user-defined classes. This section describes ABL support for these features:

- Instantiating and obtaining instances of .NET classes on page 43
- Casting .NET class and interface types on page 44
- .NET class instances and garbage collection on page 45

Instantiating and obtaining instances of .NET classes

For most .NET class types, you can instantiate the class exactly as you do an ABL user-defined class, by defining a variable or parameter as the given class type and using the NEW function (classes) to invoke a constructor of the class. For example:

```
DEFINE VARIABLE rList AS CLASS System.Collections.Generic.SortedList NO-UNDO.
rList = NEW System.Collections.Generic.SortedList( ).
```

As when creating any ABL user-defined class instance, ABL creates this .NET `SortedList` object using its default constructor and adds the instance to the session object chain, because in ABL, all ABL object instances, including .NET ones, are also instances of `Progress.Lang.Object` (see Incorporation of the .NET object model on page 21).

You can also obtain an instance of a class created by .NET and assign its object reference to an ABL data item, which also adds the instance to the ABL session object chain. This occurs in the following cases:

- When you access a .NET property or data member that returns a reference to an object created by .NET and assign this object reference to an ABL data item. For more information on accessing .NET properties and data members, see Accessing .NET class members on page 45.
- When you pass an ABL data item as an argument to a .NET output method parameter that returns (and "assigns" to the data item) the object reference of an object created by .NET. For more information on passing .NET method parameters, see Specifying .NET constructor and method parameters on page 50.
- When you access the event parameters from within an ABL handler for .NET events, which always passes an input `System.EventArgs` object reference that is created by .NET. There is also an input `System.Object` parameter, but you usually already have an object reference to this object, which doesn't need to be added to the session object chain. For more information on using ABL methods and procedures as .NET event handlers, see Handling .NET events on page 57.

For example, the `Location` property in this code fragment returns a reference to a .NET `System.Drawing.Point` object created by .NET for a button control and assigns this object reference to an ABL variable (`rLocation`) that is defined to hold that object reference:

```
DEFINE VARIABLE rLocation AS CLASS System.Drawing.Point      NO-UNDO.
DEFINE VARIABLE rButton AS CLASS System.Windows.Forms.Button NO-UNDO.

rButton = NEW System.Windows.Forms.Button( ).

/* Add button to a form */
...
rLocation = rButton:Location.
```

However, you can create or obtain a .NET object whose reference does not appear on the ABL session object chain. This happens when the object reference for the .NET object that you create or access is:

- Assigned directly to another .NET property or field (data member)
- Passed directly to an `INPUT` parameter of a .NET method
- Used directly in an ABL expression (as in a MESSAGE statement) or otherwise never stored in an ABL data element

Note: You cannot pass an object reference to a public .NET property or data member as an `OUTPUT` or `INPUT-OUTPUT` parameter to a .NET method, because ABL syntax does not allow the required colon (`:`) notation for passing parameters in these access modes. So, you cannot set a .NET property or data member object reference value in this way.

For example, this code fragment instantiates a .NET `System.Drawing.Size` object and assigns its object reference directly to the `Size` property of a .NET button object:

```
DEFINE VARIABLE rButton AS CLASS System.Windows.Forms.Button NO-UNDO.

ASSIGN
  rButton      = NEW System.Windows.Forms.Button( )
  rButton:Size = NEW System.Drawing.Size(80, 40).
```

In other words, the `System.Drawing.Size` object created by the code fragment is never stored as an ABL object reference. In all cases where you create or return existing .NET objects from .NET that you assign or pass directly back to .NET, .NET handles all garbage collection for such objects. (see .NET class instances and garbage collection on page 45).

Note that when you invoke a .NET class constructor or method that takes parameters defined as a .NET mapped data type, ABL requires that you pass arguments defined as the corresponding ABL primitive types. In other words, you cannot define and pass a .NET `System.Int32` to parameter defined as a `System.Int32`, but must pass an ABL `INTEGER` instead. Otherwise, if the parameter is defined as an unmapped .NET object type (such as `System.Drawing.Size`), you can pass an instance of that object type directly to the parameter.

For example, for the `SortedList` constructor that takes a .NET `System.Int32` parameter, you can instantiate the .NET class by passing an ABL `INTEGER` value:

```
DEFINE VARIABLE rList AS CLASS System.Collections.Generic.SortedList NO-UNDO.

rList = NEW System.Collections.Generic.SortedList(5).
```

For more information on passing .NET parameters, see the information on accessing .NET methods in Accessing .NET class members on page 45. For more information on working with .NET data types in ABL, see Using .NET data types in ABL on page 205.

To create one supported set of class types—array objects—you must use a completely different mechanism for creating class instances for them. All .NET array objects derive from the .NET base class, `System.Array`. To instantiate and perform basic operations on array objects, you must use members of the `System.Array` base class. For more information on the instantiation and management of array objects, see Accessing and using .NET arrays on page 235.

Casting .NET class and interface types

You can use the CAST or DYNAMIC-CAST function to cast .NET object types in exactly the same way that you cast ABL user-defined types.

.NET class instances and garbage collection

.NET handles garbage collection for any .NET objects involved in an ABL session. However, if you assign a .NET object reference to an ABL object reference variable (or other data element), this works like any native .NET reference to prevent .NET from garbage collecting the object. Once the ABL object reference no longer exists and no other .NET reference to the object exists, the object is again available for .NET garbage collection.

ABL references to .NET class instances require both ABL and .NET garbage collection. For pure .NET classes, the garbage collection is all in the .NET context because the ABL context maintains only an object reference to the .NET class. However, for ABL-derived .NET classes, the ABL context also maintains its own class instance, possibly including a destructor. Therefore, garbage collection is more complicated and can be much more significant because the ABL instance is not garbage collected in the ABL context until there are no more ABL references to it **and** the .NET context completes its own garbage collection for the class. For more information on ABL-derived .NET classes, see Defining ABL-extended .NET objects on page 70.

Caution: When a .NET modal form (dialog box) is open, it can be closed in a way that prevents automatic garbage collection on the form object. Thus, after a .NET dialog box is closed and no longer needed by your application, you must call the `Dispose( )` method on the form object to ensure that both the form and the .NET controls that it contains are garbage collected. For more information, see Blocking on modal dialog boxes on page 103.

Avoid using the DELETE OBJECT statement to explicitly delete the ABL component of an ABL-derived .NET object unless you are sure that there are no more references to this object within the .NET context. It is possible for the .NET context to have an active object reference to the ABL component of the object after no more ABL object references to the object exist. Prematurely deleting the ABL component of such an object can cause unpredictable behavior in the .NET context.

Note that in certain cases, ABL resources can be held by .NET and prevented from being removed from the ABL context. This can happen, for example, when a `Progress.Data.BindingSource` (ProBindingSource) object that was attached to a ProDataSet is awaiting .NET garbage collection. Until it is garbage collected, all internal buffers and queries for the ProDataSet remain in the ABL context.

You can force immediate removal of these excess buffers and queries by doing one of the following:

- Calling the .NET `Dispose( )` method on the ProBindingSource object reference (if still available
- Explicitly deleting the ProBindingSource object using a DELETE OBJECT statement
- Deleting the ProDataSet for which the buffers and queries were created.

However, before deleting such ABL resources, be certain that .NET is no longer using them. For more information on ProBindingSource objects, see Binding ABL Data to .NET Controls on page 131.

Accessing .NET class members

ABL supports access to public or protected members of a .NET class instance using much the same syntax for accessing public or protected members of an ABL class instance. This includes support for using built-in ABL data types that automatically map to specified .NET data types, for example, to get and set .NET property values or to pass .NET method parameters. For more information on data type mapping, see Using .NET data types in ABL on page 205.

Note: The Progress Developer Studio for OpenEdge provides a Class Browser view that you can use to list the members of a .NET class, showing the signature for each member using ABL syntax.

So, you can access the following members of a .NET class instance in exactly the same way as you access them for an ABL class instance:

- **.NET constructors** — Invoke a .NET class constructor by instantiating a .NET class exactly like an ABL class using the NEW function (classes). From the constructor of an ABL-derived .NET class, you can also access a constructor of the immediate super class using the SUPER statement. For more information, see Instantiating and managing .NET class instances on page 42, and Specifying .NET constructor and method parameters on page 50.
- **.NET fields** — Equivalent to ABL data members, read or write the values of .NET fields of a class using class-based data member access, as described in OpenEdge Development: ABL Reference.
- **.NET methods** — Invoke .NET methods of a class using class-based method calls, as described in OpenEdge Development: ABL Reference. Also, see Specifying .NET constructor and method parameters on page 50.
- **.NET properties** — Set or get the values of .NET properties of a class using class-based property access, as described in OpenEdge Development: ABL Reference. You can also access .NET properties dynamically using Accessing .NET indexed properties dynamically on page 52.
- **.NET object events** — Like ABL classes, .NET classes can define, publish, and handle events. .NET applications handle events by subscribing .NET methods as event handlers. ABL applications can handle .NET events by subscribing ABL methods and internal procedures as event handlers. These ABL event handlers respond to published .NET events in a manner similar to .NET event handlers in .NET applications. For more information, see Handling .NET events on page 57.

Note: In .NET, you separate the reference to a .NET field, method, or property from its associated object reference (for an instance member) or class type name (for a static member) using a period (`.`) instead of a colon (`:`), as in ABL.

ABL allows you to use native syntax to access class members in support of certain .NET features not supported in native ABL applications. These include:

- **.NET interface members** — .NET supports types of interface members that ABL interfaces do not support. For more information on accessing these interface members in ABL, see the .
- **.NET enumeration types** — Enumerations are special .NET classes that can represent a particular enumerated subset of values of a given .NET data type. Each value of that subset is represented as a member of a given enumeration type. For more information on enumeration types and how to use them in ABL, see Accessing and using .NET enumeration types on page 228.

ABL also provides extended syntax to access .NET class member features that are currently not supported for ABL class members or that require additional syntax to access them in the .NET context. The following sections thus describe the extended syntax for:

- **Static members of a .NET class** — .NET supports static class members, which are accessible in ABL only on the class type exactly like static members of ABL classes. The only difference for .NET static members in ABL is the additional syntax for accessing .NET inner class types (see Referencing .NET class and interface types on page 37). For general information on accessing static class members in ABL, see OpenEdge Development: Object-oriented Programming. For more information on accessing .NET static class members in ABL, see Accessing static members of a .NET class on page 49.
- **Explicit mappings to .NET data types** — In many cases, wherever a given .NET data type maps to an ABL primitive type, you can use the corresponding ABL primitive type to exchange data with the .NET type. However, some usage requires that you explicitly indicate the .NET data type that you intend to use. For example:
  - You must use appropriate mappings between ABL and .NET parameter options to pass ABL data to overloaded .NET constructor and method parameters. ABL extends its parameter-passing syntax to specify explicit mappings between ABL primitive types and the corresponding .NET mapped data types

of parameters. For more information on how and why ABL supports these mappings, see Specifying .NET constructor and method parameters on page 50.

- You must use explicit mappings for .NET mapped data types when you override .NET methods or .NET abstract properties in an ABL-derived .NET class. For more information, see Deriving .NET classes in ABL on page 72.
- You must use explicit mappings for .NET mapped data types when you reference the constructed type name of a .NET generic type. For more information, see Referencing .NET generic types on page 39.

- **.NET indexed properties and collections** — .NET supports properties that can provide access to a keyed group of values. .NET commonly uses these indexed properties to support object collections. For more information on using both .NET indexed properties and collections in ABL, see Accessing .NET indexed properties and collections on page 52.

Accessing members of .NET interfaces

Like ABL interfaces, .NET interfaces can define properties, methods, and events. Similar to an ABL interface type, an object reference defined as a .NET interface type allows you to access members of any .NET class instance that implements the interface, including all public properties, methods, and events defined by the interface. Thus, .NET interfaces support access to classes that implement them in the same manner as ABL interfaces.

You can also define an ABL class that implements the properties, methods, and events of a .NET interface in much the same way as it might implement an ABL interface. For more information, see Defining ABL-extended .NET objects on page 70.

Unlike ABL classes, .NET classes can implement interface members in a special manner that allows you to access that member **only** using an object reference defined as the interface type. .NET provides this feature to handle the implementation of multiple interfaces that define identical members, each of which is intended to be implemented for a different application. Thus, if the same .NET class implements more than one such interface, it can implement that identical member differently for each interface.

To enable you to tell the class which member implementation you want to access, .NET allows you to identify the member explicitly using an object reference defined with the associated interface type. .NET also requires the class to implement and identify that interface member as an *explicit interface member*, which you can only access using the interface type. If you try instead to access an explicit interface member using an object reference to its implementing class type, .NET raises an exception, even if the class implements the member for only one interface.

You can identify the explicit interface members of a class from the class documentation. The Microsoft .NET Framework documentation (see OpenEdge Installed .NET Controls on page 201) indicates any explicit interface members in a table of "Explicit Interface Implementations" that is shown in the list of members for each class.

For example, you might have a `Vehicle` class that implements a `Drive( )` method that is defined with an identical signature by a `Car` interface and a `Train` interface, both of which are implemented by the `Vehicle` class. However, the `Vehicle` class must implement the `Drive( )` method differently for a car than it does for a train, and a user of the `Vehicle` class must be able to access the implementation of the `Drive( )` method that is appropriate for their particular vehicle, whether it be a car or a train.

The following code fragment creates a three-element .NET array (`System.Array` class) of `System.Int32` elements, which ABL maps as `INTEGER`. It then defines object references for two interface types that `System.Array` implements and sets them to reference the array instance that is created using the `CreateInstance( )` method:

```
DEFINE VARIABLE rArray       AS CLASS System.Array                    NO-UNDO.
DEFINE VARIABLE rIList       AS CLASS System.Collections.IList        NO-UNDO.
DEFINE VARIABLE rICollection AS CLASS System.Collections.ICollection NO-UNDO.
ASSIGN
  rArray       = System.Array:CreateInstance
                   (System.Type:GetType("System.Int32"), 3)
  rIList       = rArray
  rICollection = rArray.

rArray:SetValue(0, 0).
rArray:SetValue(1, 1).
rArray:SetValue(2, 2).

MESSAGE "Array element 1 = " rIList[1] VIEW-AS ALERT-BOX INFORMATION.
MESSAGE "Array Count = " rICollection:Count VIEW-AS ALERT-BOX INFORMATION.

rIList:Clear( ).

MESSAGE "Cleared array element 1 = " rIList[1] VIEW-AS ALERT-BOX INFORMATION.
```

Note: For information on creating and using .NET arrays, see Accessing and using .NET arrays on page 235.

Finally, it initializes the array with the `INTEGER` values 0, 1, and 2 in their respective array elements, then accesses two explicit interface properties (the default indexed property and the `Count` property) and one explicit interface method (the `Clear( )` method) on their respective interface references.

In this case, MESSAGE statements display the initialized value at position 1, the count of the elements, and the cleared value at position 1 in the array.

Note: You can do all of these array operations using public properties and methods of `System.Array`. However, this code uses explicit interface methods and properties for demonstration.

On the other hand, if you attempt to invoke these explicit interface methods and properties on a reference to the `System.Array` class instance, ABL raises compile-time errors starting with the first explicit interface member reference that it encounters, as in the following code fragment:

```
DEFINE VARIABLE rArray AS CLASS System.Array NO-UNDO.
rArray = System.Array:CreateInstance
           (System.Type:GetType("System.Int32"), 3).

rArray:SetValue(0, 0).
rArray:SetValue(1, 1).
rArray:SetValue(2, 2).

/* All of the following bolded code results in compile-time errors */
MESSAGE "Array element 1 = " rArray[1] VIEW-AS ALERT-BOX.
MESSAGE "Array Count = " rArray:Count VIEW-AS ALERT-BOX.

rArray:Clear( ).

MESSAGE "Cleared array element 1 = " rArray[1]
  VIEW-AS ALERT-BOX INFORMATION.
```

Accessing static members of a .NET class

ABL supports access to the following kinds of .NET static class members:

- Fields (data members)
- Properties
- Methods
- Events

Microsoft .NET class library documentation indicates a static member in the list of members for a class using the following symbol:



ABL allows you to reference a static .NET member using the same general syntax as an ABL static member, as follows:

Syntax

```
[ [class-type-name] : ] class-member-reference
```

The *class-type-name* is the type name (qualified or unqualified, depending on the USING statements you specify) of the .NET class that defines the static member. Note that ABL supports additional syntax for referencing .NET inner class types (see Referencing .NET class and interface types on page 37). The *class-member-reference* is a read or a write access to a static data member or property, a call to a static method, or a reference to a static event to which you are subscribing or unsubscribing an event handler. For more information on accessing static members of a class in ABL, see the sections on using static members in OpenEdge Development: Object-oriented Programming.

Note: In .NET, you separate the *class-type-name* from the *class-member-reference* using a period (`.`) instead of a colon (`:`), as in ABL.

For example, the following procedure displays a static property on the `Progress.Windows.Form` class (`MousePosition`) without instantiating the class:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

MessageBox:Show("Mouse Position = " +
                Progress.Windows.Form:MousePosition:ToString( )).
```

Note: `Progress.Windows.Form` is an OpenEdge .NET form class that inherits from `System.Windows.Forms.Form`. For more information, see Creating and Using Forms and Controls on page 89.

This property contains the position of your mouse pointer at the moment you access the property. Note that this value is displayed using the static `Show( )` method on the Microsoft .NET `System.Windows.Forms.MessageBox` class.

Specifying .NET constructor and method parameters

ABL supports calls to .NET constructors and methods in much the same way as it does for ABL constructors and methods. However, you must identify the syntax for passing a .NET constructor or method parameter from one of the following sources:

- Interpret the ABL parameter passing syntax from the corresponding parameter as specified for the constructor or method signature in the .NET documentation for the method. The .NET documentation typically provides method signatures in two or more of the supported CLS-compliant languages.
- Interpret the ABL parameter passing syntax from the signature for the method displayed using the Class Browser view of the Progress Developer Studio for OpenEdge. This Class Browser allows you to view all .NET method signatures using an extended form of ABL parameter definition syntax. ABL documentation also uses this syntax to document the public methods provided by OpenEdge .NET classes. For more information, see the introduction to the Class Properties and Methods Reference (.NET Objects) section in OpenEdge Development: ABL Reference.

This section provides guidelines for identifying these parameter options. To help with this, ABL also provides extended syntax to disambiguate .NET parameter data types of overloaded method parameters, all of which map to the same ABL data type.

This is the general syntax for a parameter list that you pass to a class constructor that you invoke using the NEW function (classes) or that you pass to a method that you call:

Syntax

```
( parameter [ , parameter ] . . . )
```

For a .NET class constructor or method you can specify each *parameter* using the following parameter passing syntax:

Syntax

```
[ INPUT | OUTPUT | INPUT-OUTPUT ] parm [ AS data-type ]
```

Note: You can use the same syntax for passing .NET parameters that you define for ABL routines, including ABL methods, procedures, and user-defined functions.

A *parm* is the data that you pass as a parameter, which can take one of several possible forms—for example, a literal value, expression, or variable—any ABL element supported for parameter passing that provides or can hold a value. To identify the parameter mode (`INPUT`, `OUTPUT` or `INPUT-OUTPUT`) and the form of data that you can specify for *parm*, you have to know the .NET parameter mode and data type of the corresponding constructor or method parameter. For a Microsoft .NET constructor or method, you can locate this information for each constructor and method in the .NET Framework class library documentation, where the parameter mode, and often the data type, is indicated using language-specific syntax.

The following table shows how to determine the ABL parameter mode from the keyword used to specify the equivalent parameter mode in C#.

Table 1: C# syntax matching ABL parameter modes

ABL parameter mode	Corresponding C# syntax
`INPUT` *`parm`*	*`parm`*
`OUTPUT` *`parm`*	`out` *`parm`*
`INPUT-OUTPUT` *`parm`*	`ref` *`parm`*

Note that in C#, the default (no keyword) corresponds to the ABL `INPUT` mode. Therefore, you cannot pass a literal value or an expression to a parameter whose C# mode is specified by `out` or `ref`.

As for any ABL method parameter, you must pass *`parm`* as a data type that matches the data type of a given .NET constructor or method parameter. For all .NET object types, except for a small subset briefly described in the following paragraph, you must pass a compatible .NET object type, similar to how you pass an ABL user-defined class or interface type as an ABL method parameter.

A small subset of .NET object types correspond and map to .NET language primitive data types, such as the C# `int`, `float`, or `string`. Both the .NET primitive types and their corresponding object types are referred to, in ABL documentation, as *.NET mapped data types*. For each of these .NET mapped types (whether it is the primitive or object equivalent), you must pass a corresponding ABL built-in primitive type. ABL supports implicit mappings between ABL built-in primitive types and all .NET mapped data types. ABL also supports widening relationships that allow multiple ABL primitive types to be passed as certain .NET mapped data types.

Because ABL has fewer primitive types than .NET has mapped data types, the implicit mappings include a few ABL primitive types that match multiple .NET mapped data types. In order to support .NET constructor and method overloading, the parameter passing syntax includes the `AS` option where you can specify a keyword (*`data-type`*) that corresponds to a particular .NET mapped data type. In this way, you can allow ABL to identify the specific overloading when more than one .NET data type in the overloading for a parameter maps to the ABL primitive type of the argument you are passing.

Note: This parameter-passing `AS` option is essentially the same as the `AS` option for passing parameters to COM object methods. For more information, see the Accessing COM object properties and methods section in OpenEdge Development: ABL Reference.

.NET also supports constructors and methods with a variable number of parameters. The parameters in this variable list are always of the same .NET data type. In ABL, you can pass the variable parameters in a single-dimensional array as the final parameter (*`parm`*) to the constructor or method. You can typically call a method that takes variable parameters multiple times, varying the number of elements in the array with each call according to the number of variable parameters that you want to pass. In .NET documentation, the C# signature for a method that defines variable parameters specifies the keyword `param` at the position of the variable parameter (again, always as the last parameter), followed by the data type of the parameter array.

Note: In some .NET languages, you can provide the elements of the variable parameter array as individual parameters to the constructor or method call. However, in ABL you can only pass variable parameters in a single array parameter.

For more information on data types for parameter passing, including the implicit data type mappings, the available keywords to indicate explicit .NET data type mappings using the `AS` option, widening options, and using arrays with .NET, see Using .NET data types in ABL on page 205 and Passing ABL data types to .NET constructor and method parameters on page 218. For reference information on passing .NET constructor and method parameters, see the Parameter passing syntax reference entry in OpenEdge Development: ABL Reference.

Accessing .NET indexed properties and collections

ABL supports access to .NET indexed properties. In .NET, an indexed property has a group of values. Each of these property values is referenced by an *indexer* that can be defined with one or more keys. In ABL, you can only reference an indexed property whose indexer is defined with one key, as shown in the following syntax:

Syntax

```
[object-reference]:property-name[ key ]
```

The *object-reference* is a reference to an instance of .NET class that has an indexed property you want to reference. The *property-name* is the name of the indexed property, and *key* is a key value expression of a specified data type that identifies a particular property value. The range of valid values for *key* depends entirely on its definition, regardless of the data type. Like the syntax for accessing an ABL array, the square brackets of the indexer (`[ ]`) for an indexed property are part of the syntax.

Note: .NET documentation strongly recommends that any indexed property should only be created with one key for the indexer, and this is how all indexed properties are defined for the Microsoft .NET Framework. However, it is possible that other third-party .NET classes could define indexed properties with two or more keys for the indexer, in which case you cannot access those indexed properties in ABL.

Microsoft .NET class library documentation identifies indexed properties using the following indications: it typically is defined as a default property with the name, `Item` (see Default indexed properties on page 54), and the language-specific property signature always includes an index definition. For example, a C# indexed property definition might be shown as in the following example of some `Control` property, where `this` refers to the class in which the indexed property is defined and *index* identifies the data type of the single indexer key:

```
public virtual Control this [
         int index
] { get; }
```

.NET primarily uses indexed properties to access the items of *collections*, which are objects that allow you to manage collections of other objects. OpenEdge also provides indexed properties on its OpenEdge .NET class, Progress.Data.BindingSource (ProBindingSource). For more information on the ProBindingSource, see Binding ABL Data to .NET Controls on page 131.

Accessing .NET indexed properties dynamically

You can access or set .NET indexed properties dynamically using the `DYNAMIC-PROPERTY` function by providing a character expression for the property name, along with an object reference, the `THIS-OBJECT` system reference, or a class type. At runtime, the AVM determines the actual type, resolves the property name expression, validates the property name against the actual type, and validates and performs the assignment.

This is the syntax of the `DYNAMIC-PROPERTY` function:

Syntax

```
[ return-value = ] DYNAMIC-PROPERTY( { object-reference | class-type-name }
  , property-name [ , index ] )
```

This is the syntax of the DYNAMIC-PROPERTY statement:

Syntax

```
DYNAMIC-PROPERTY( { object-reference | class-type-name }
  , property-name [ , index ] ) = new-value
```

Element descriptions for these syntax diagrams follow:

return-value

Specifies a data element that is assigned the value returned when you execute the property's GET accessor. The appropriate CAST or ABL data type conversion function is applied to convert the property's value to the data type of *return-value*.

object-reference

Specifies a reference to a .NET class instance that exposes the specified property as an instance member. The compiler allows *object-reference* to be declared as any object type. At runtime, the object type must resolve to the type that exposes the property.

class-type-name

Specifies the name of a .NET class type that defines the specified property as a static member. This is a CHARACTER expression that the AVM evaluates to the type name of a class at runtime. It must be the name of a class: interfaces are not valid.

property-name

Specifies a CHARACTER expression that evaluates to the property's name at runtime. For non-dynamic access, you can refer to a .NET indexed property without using the property name. In dynamic access, however, you must use the property name.

index

Specifies an index of the specified element. For a .NET indexed property, *index* is an expression for the specified element. At compile time, index can be any data type that ABL supports for indexers. However, at runtime, it must resolve to a data type appropriate to the property type.

new-value

The value of any data type that can be assigned to the property.

Note: The value of new-value cannot be converted to the property's type.

ABL also provides the `GetPropertyValue( )` and `SetPropertyValue( )` methods with functionality similar to the `DYNAMIC-PROPERTY` function and statement, respectively. Use the `GetPropertyValue( )` and `SetPropertyValue( )` methods to access and set a property value at runtime, when you do not know the property's name or type at compile time. There is a difference between `GetPropertyValue( )` or `SetPropertyValue( )` and `DYNAMIC-PROPERTY`—using `GetPropertyValue( )` or `SetPropertyValue( )` requires an instance of `Progress.Lang.Class` in addition to the object reference when accessing instance properties.

Conversions between the property type and the source type of a `SET` or the target type of a `GET` succeed as long as the conversion between the data type of the source and the data type of the target of the assignment is supported. The AVM does the required conversions automatically as if there were a `CAST` or an ABL data type conversion function such as `STRING` or `INTEGER` present.

Default indexed properties

A .NET class can have one default indexed property. In fact, C# can **only** define default indexed properties. The default indexed property for a class typically has the name, `Item`. ABL allows you to access a default indexed property in two ways:

- Using the indexer on the property name, as described previously
- Using the indexer directly on the object reference to a class instance without the need to specify the property name, as in the following syntax:

```
[object-reference] [ key ]
```

Note: Much of the Microsoft .NET documentation, especially for C#, refers to an indexed property, which is always a default property, as an *indexer*. The ABL documentation, however, refers to an *indexer* only as the bracketed expression for accessing an indexed property (default or not). Also, what .NET documentation refers to as the *index* of an indexer, ABL documentation refers to as the *key* of the indexer. This is to distinguish a .NET property indexer key with an ABL array index.

For example, the following code fragment displays the same default indexed property value of a `System.Windows.Forms.Control+ControlCollection` object by using the indexer on the object reference and by using its property name:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE rForm AS Progress.Windows.Form          NO-UNDO.
DEFINE VARIABLE rCntrlColl AS Control+ControlCollection NO-UNDO.
ASSIGN
  rForm      = NEW Progress.Windows.Form( )
  rCntrlColl = rForm:Controls.
. . .
rCntrlColl:Add( ... ).
...
MESSAGE rCntrlColl[5] VIEW-AS ALERT-BOX.
MESSAGE rCntrlColl:Item[5] VIEW-AS ALERT-BOX.
```

Note: `Progress.Windows.Form` is an OpenEdge .NET form class that inherits from `System.Windows.Forms.Form`. For more information, see Creating and Using Forms and Controls on page 89.

The `Controls` property of a .NET form is defined as a `Control+ControlCollection` object and is an example of a .NET collection. All collections have a default indexed property, along with common properties and methods to access the objects that they contain. In this fragment, the `Add( )` method is used to add objects to the collection. For more information on using collections, see Working with collections on page 56.

Note that sometimes a property is defined as a class type, and this class type has a default indexed property as a member. For such properties, references to the default indexed property of the class type make the property, itself, appear to be an indexed property. For example, the `Controls` property on a `System.Windows.Forms.Form` class is defined as a `System.Windows.Forms.Form+ControlCollection`, whose control instances you can access:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE rControl AS Control NO-UNDO.

rControl = NEW Control( ).
...
rControl:Controls:Add( ... ).
...
MESSAGE STRING(rControl:Controls[5]) VIEW-AS ALERT-BOX.
```

So, when you reference the `Controls` property to access one of the control instances that the class it references contains, the `Controls` property reference, itself, appears to be a reference to a non-default indexed property on its class instance (`rControl`). Instead, you are actually using a non-indexed property (`Controls`) to reference the default indexed property on the class instance (`Form+ControlCollection`) that the `Controls` property, itself, references. In ABL, the two different references—a non-default indexed property reference and a non-indexed property reference to a default indexed property—appear syntactically the same, but they refer to two different things, which you typically do not have to recognize when using them.

Indexed property overloading

Unlike the case for an ABL array index, the data type of an indexed property's key does not have to be an `INTEGER`. Also, .NET allows overloaded indexed properties within a class that are distinguished by the data type of the index key, similar to parameter data types of overloaded methods. For example in ABL, the default indexed property of the `Control+ControlCollection` class can be indexed by either an `INTEGER` or a `CHARACTER` (or `LONGCHAR`) data type.

Note that some ABL data types map implicitly to more than one .NET data type. This affects how ABL identifies the indexed property to access. If a property indexer key is overloaded by multiple .NET data types that implicitly map to a single ABL data type, and you specify an indexer using that ABL data type, ABL uses the indexer key whose .NET data type is the default match for the ABL data type. For example, if an indexed property key is overloaded by the C# data types `short` and `int` (both of which map to `INTEGER` in ABL), ABL always accesses the property by the `int` key. However, if there is no key defined with the default matching data type, ABL chooses the first key that it encounters that is an implicit match for the specified ABL data type.

Note: This means that without a default-matching key defined for the property, you cannot be sure which key ABL will use among the keys that map implicitly to the specified ABL data type. However, it always uses the same one for a given indexed property.

You also cannot disambiguate overloaded indexed property keys using the `AS`*`data-type`* option that is available for method parameters (see Specifying .NET constructor and method parameters on page 50). Using this option on property index keys raises a compile-time error.

ABL also respects data type widening for the match. For more information on .NET data type mappings and the default matches for ABL data types, see Using .NET data types in ABL on page 205.

Indexed properties in chained references

You can use indexed properties in chained references, for example, as shown by the following syntax examples:

Syntax

```
variable-name = object-reference:object-reference:property-name[ key ].
```

```
object-reference:property-name[ key ]:method-name( ).
```

```
variable-name = object-reference:property-name[ key ]:property-name.
```

Working with collections

Most Microsoft .NET indexed properties are default indexed properties for collections. A collection is a class that implements the following interfaces:

- `System.Collections.ICollection`
- `System.Collections.IEnumerable`
- `System.Collections.IList`

Some commonly used methods and properties of a collection class include:

- **`Add( )` method** — For adding objects to a collection, as follows:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE myControls AS CLASS Control+ControlCollection NO-UNDO.
DEFINE VARIABLE myForm     AS CLASS Progress.Windows.Form     NO-UNDO.
DEFINE VARIABLE myButton   AS CLASS Button                    NO-UNDO.

ASSIGN
  myForm        = NEW Progress.Windows.Form( )
  myButton      = NEW Button( )
  myButton:Text = "Ok".

/* Controls property references a ControlCollection object */
myControls = myForm:Controls.
myControls:Add(myButton).
```

Note: `Progress.Windows.Form` is an OpenEdge .NET form class that inherits from `System.Windows.Forms.Form`. For more information, see Creating and Using Forms and Controls on page 89.

- **`Contains` property** — To determine if a particular object is in the collection, as follows:

```
IF myControls:Contains(myButton) THEN
  MESSAGE "myControls has a myButton object" VIEW-AS ALERT-BOX.
```

- **`Count` property** — To determine how many objects there are in the collection, as follows:

```
MESSAGE "myControls has " myControls:Count " objects." VIEW-AS ALERT-BOX.
```

- **`Item` property** — The default indexed property to access a particular object in the collection. Some collections overload this property. However, they all have one `Item` property that is indexed on a zero (0)-based `INTEGER` key, as follows:

```
MESSAGE "Control #0 Text: " myControls[0]:Text VIEW-AS ALERT-BOX.
```

- **`Remove( )` method** — For removing an object from a collection, as follows:

```
IF myControls.Contains (myButton) THEN
  myControls:Remove(myButton).

MESSAGE "myControls has " myControls:Count " objects." VIEW-AS ALERT-BOX.
```

For more information on collections, see the .NET Framework documentation for a particular .NET collection class and the `ICollection`, `IEnumerable`, and `IList` interfaces in the `System.Collections` namespace. For more information on locating this documentation, see OpenEdge Installed .NET Controls on page 201.

Handling .NET events

ABL supports several event models, including the use of triggers and call-backs for handle-based objects and the use of class events for class-based objects. .NET also supports an event model for classes. Like ABL class-based objects, .NET objects define events as members. Each object that defines an event is responsible for sending (*publishing*, in ABL terms) the event in response to some condition. Any other object can receive and respond to a given event using an *event handler* whose signature is defined as part of the event definition by a type of .NET class known as a *delegate*. The definitions for ABL class events typically define event signatures like an ABL method definition, but can also define event signatures with reference to a .NET delegate type. Again, like ABL class events, the events defined by .NET objects can be either instance or static events.

To respond to (*handle*) a .NET event in ABL, you must specify (*subscribe*) either an ABL method or an internal procedure as a handler for the event. (You cannot subscribe a .NET method as a handler for .NET events in ABL.)

Note: OpenEdge does not support the handling of .NET object events in **non-GUI** ABL sessions, such as a character-mode client or AppServer agent.

The .NET documentation for each event of an object specifies a delegate that defines the type of event handler that you must use to handle the event. Although ABL does allow (and sometimes requires) you to reference .NET delegates to define class event signatures, ABL does not refer directly to delegates to define event handlers, but instead provides its own syntax to do this, as described in this section. However, when you define a handler for a .NET event in ABL, you generally need to consult the .NET documentation for the delegate associated with a given event to identify the appropriate signature for the event handler that you define. This section also describes how to do this.

The following sections describe:

- Managing .NET events in ABL on page 58
- Identifying the events published by a .NET object on page 59
- Defining handlers for .NET events in ABL on page 59
- Specifying handler subscriptions for .NET events on page 61
- Managing .NET events from ABL-extended .NET classes on page 61
- Event handling example on page 62

Managing .NET events in ABL

The .NET event model functions as a call-back model, where the call-back is a method whose signature is defined by a specified .NET delegate. In ABL, the call-back for a .NET event can be an ABL method or an internal procedure whose signature is compatible with the signature defined for the .NET event. Like .NET, ABL uses the associated .NET delegate to validate any method or internal procedure subscribed as a handler for the event. However, ABL provides its own mechanism to subscribe the ABL routine as an event handler. The primary requirement for an ABL event handler is that the signature of the method or internal procedure must match the signature specified by the .NET delegate defined for the event. (For OpenEdge .NET events, the handler signature is included in the documentation for each event—see Defining handlers for .NET events in ABL on page 59.) The main difference between using methods and internal procedures as handlers for .NET events is that ABL verifies method event handler subscriptions at compile time (using strong typing), but verifies internal procedure event handler subscriptions only at run time.

In general, you can handle .NET events in ABL similar to how you handle ABL class events.

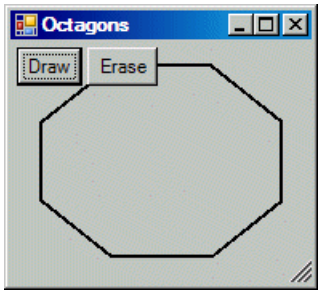
To handle .NET events in ABL:

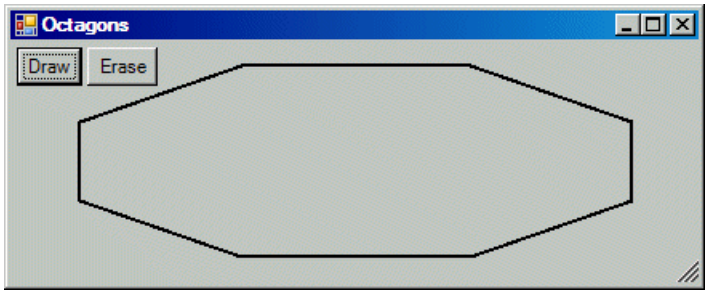
1. Identify the .NET object that defines and publishes the events, and identify an ABL routine signature that is compatible with the signature defined for each .NET event. For more information, see Identifying the events published by a .NET object on page 59.
2. Write an ABL method or internal procedure as part of each class or procedure definition that you want to receive an event. For more information, see Defining handlers for .NET events in ABL on page 59.
3. Where ever your application needs to prepare for receiving an event, subscribe an appropriate method or internal procedure as a handler for the event. For more information, see Specifying handler subscriptions for .NET events on page 61.
4. At any appropriate point in your application, block for input to allow the .NET objects you are using to publish events and allow your event handlers to respond to them. In ABL, these are typically GUI for .NET events that are published while blocking on a displayed .NET form. This chapter previews the use of .NET forms. For more information, see Creating and Using Forms and Controls on page 89.

You can also do more to manage .NET events from ABL-extended .NET classes. For more information, see Managing .NET events from ABL-extended .NET classes on page 61.

Identifying the events published by a .NET object

As with ABL class events, .NET events are members of any .NET class that defines and publishes them. You can therefore identify these events and the signatures they require in the documentation that describes the members of a given .NET class, as well as using the Class Browser of Progress Developer Studio for OpenEdge. For more information on the events supported by a given .NET object, see the Microsoft, the Infragistics, or other third-party documentation on the object. OpenEdge provides two classes that inherit .NET events from Microsoft .NET classes—Progress.Windows.Form, which inherits from `System.Windows.Forms.Form`, and Progress.Windows.UserControl, which inherits from `System.Windows.Forms.UserControl`. For more information on these classes, see Creating and Using Forms and Controls on page 89. In addition, OpenEdge provides a class, Progress.Lang.BindingSource (ProBindingSource), which defines its own .NET events and inherits others from the .NET `System.Windows.Forms.BindingSource` class. For more information on the ProBindingSource, see Binding ABL Data to .NET Controls on page 131.





The following section provides more information on finding the information required to handle these events.

Defining handlers for .NET events in ABL

You must define any ABL class-based method or internal procedure that you want to subscribe as a handler for .NET events using a signature that is required by the .NET event. In .NET, this signature is defined by a delegate type that is associated with each event definition.

As a .NET Framework convention, .NET event handler signatures have a void return type and consist of two input parameters, both of which are object references. The first parameter, identified as the *sender*, is always a reference to a `System.Object` instance, which represents the object that defines and publishes the event. The second parameter is typically a reference to an event arguments class instance that is generated by .NET each time it publishes the event. An *event arguments class* can be a `System.EventArgs` (the base class for all event arguments classes) or any derived class.

Note: .NET actually supports any signature for an event that is defined by its delegate. However, ABL only supports .NET event signatures defined by delegates that conform to this .NET Framework convention.

The difference between one .NET delegate and another is the specific event arguments class specified for the event handler signature that it defines. Thus, .NET events that are defined with the same signature are typically defined with reference to the same delegate.

Identifying the signature for an event handler on a third-party .NET object

As described in the previous section, you can identify the signature required for an event handler from the .NET documentation for the event.

To identify the signature required for any ABL method or procedure that you want to define as a handler for a .NET event on a third-party .NET object, such as a Microsoft or an Infragistics object:

1. In either the .NET Class Library or the API Reference documentation (see OpenEdge Installed .NET Controls on page 201), look up the object event that you want to handle—for example, the `FormClosing` event of `System.Windows.Forms.Form`. The documentation shows the event declaration using the syntax of several .NET languages. In C#, the `FormClosing` event declaration appears as follows:

```
public event FormClosingEventHandler FormClosing
```

2. The word (following `event`) that typically appears as a hypertext link in this event declaration is the name of the delegate class associated with the event. Click on this link. The documentation for the associated delegate class appears—in this case, it is for the `System.Windows.Forms.FormClosingEventHandler` delegate. The documentation shows the event handler prototype using the syntax of several .NET languages. In C#, the `FormClosingEventHandler` prototype appears as follows:

```
public delegate void FormClosingEventHandler
  (Object sender, FormClosingEventArgs e)
```

3. This signature indicates the return type (`void` by convention) and the parameters that you must define for your event handler—in this case, the typical first parameter, which is a `System.Object`, and the second parameter that for the `FormClosingEventHandler` delegate is a `System.Windows.Forms.FormClosingEventArgs` class. In this case, you might use the `Cancel` property of the `FormClosingEventArgs` class to cancel the `FormClosing` event, which prevents the form from being closed.

Note: For any ABL method or internal procedure defined as a .NET event handler, if you never intend to actually use the second parameter of its signature, or you prefer to cast the object reference to its specified event arguments subclass at some later point in the event handler block, you can always define this parameter using the common event arguments base class, `System.EventArgs`.

Identifying the signature for an event handler on an OpenEdge .NET object

To identify the signature required for any ABL method or procedure that you want to define as a handler for a .NET event on an OpenEdge .NET object:

1. Look up the object event that you want to handle in the OpenEdge documentation for the event (see the Class Events Reference section of OpenEdge Development: ABL Reference).
2. The syntax that introduces the event reference entry shows the required signature. For example, in the reference entry for the `CreateRow` event, this syntax appears:

```
EventHandlerName
   (INPUT sender AS CLASS System.Object,
    INPUT args AS CLASS Progress.Data.CreateRowEventArgs).
```

3. You can then define your event handler parameter accordingly.

Specifying handler subscriptions for .NET events

ABL allows you to subscribe or unsubscribe a class-based method or internal procedure as an event handler for a given .NET event using the same mechanism it uses for ABL class events.

This is an overview of the ABL syntax to manage event handler subscriptions to .NET events:

Syntax

```
[ publisher : ] event-name : { Subscribe | Unsubscribe }
  ( [ subscriber : ] handler-method |
    [ subscriber-handle , ] handler-procedure ) [ NO-ERROR ] .
```

You use the following ABL built-in event methods to manage event handler subscriptions:

- Subscribe( ) — Subscribes the specified method (*handler-method*) or internal procedure (*handler-procedure*) as a handler for the .NET event specified by *event-name*
- Unsubscribe( ) — Removes the specified method or internal procedure as a handler for the .NET event specified by *event-name*

When you call these event methods, the optional *publisher* identifies the .NET class or class instance that publishes the specified static or instance event, the optional *subscriber* is the ABL class or class instance that defines the specified static or instance method, and the optional *subscriber-handle* is a handle to an external procedure that defines the specified internal procedure. You need these options only if the class or procedure context where you invoke `Subscribe( )` or `Unsubscribe( )` does not resolve the specified element. Note that even though *handler-method* is for handling a .NET event, the handler cannot be a .NET method; it can only be an ABL class method.

Otherwise, this syntax works the same way for .NET events as it does for ABL class events. For more information, see the sections on subscribing to ABL class events in OpenEdge Development: Object-oriented Programming.

For a complete description of these event methods for managing both ABL class and .NET events, see the reference entries for the Subscribe( ) method and the Unsubscribe( ) method in OpenEdge Development: ABL Reference.

Managing .NET events from ABL-extended .NET classes

To more fully integrate .NET event management in an ABL application, you can also publish many .NET events in ABL-extended .NET classes, where you might:

- Publish some events inherited from a .NET super class using an inherited method that publishes each event.
- Implement and publish inherited .NET abstract events similar to implementing ABL abstract events
- Implement and publish events defined in .NET interfaces similar to implementing ABL interface events

You can also manage events published by private .NET controls by delegating the handling to public ABL events that you define and publish on their behalf. For example, you might create an ABL-derived .NET user control that privately contains other .NET controls. In such an ABL-derived user control, you can manage events on its privately contained .NET controls by handling all the events for these controls internally, and defining and publishing public ABL class events from your user control in their place. In this way, consumers of the ABL-derived .NET user control can receive just the events that the user control needs its consumers to handle rather than making its contained .NET controls public for its consumers to manage their events directly. For more information on .NET event management for ABL-extended .NET classes, see Defining ABL-extended .NET objects on page 70.

Event handling example

The following procedure, `EventHandlers.p`, displays a form and draws or erases an octagon defined by a series of points that can change their locations depending on the following conditions:

- When the form (**Octagons**) that displays the octagon first opens.
- Whenever you click the **Draw** button (using a visible graphical pen color).
- Whenever you click the **Erase** button (using an invisible graphical pen color, the background color).
- Whenever the **Octagons** form is moved.
- Whenever the **Octagons** form is painted on the screen. Painting occurs when the form is first opened, whenever the form is resized, and when other forms and windows are dragged across the top of the **Octagons** form. (Painting, however, does not occur in the same way when the form is moved, which is why the form draws the octagon explicitly when it is moved.)

For information on locating and running this sample, see Example procedures.

Thus, the octagon displayed by `EventHandlers.p` continues to appear when its window is moved or stretched.

The points used to draw the octagon are maintained in a .NET array whose values are recalculated as necessary to allow the octagon to fit within the dimensions of the form client area. Another procedure example, `PointerArray.p`, is a simpler version of the same application. It performs the same octagon calculations and array operations in order to demonstrate the use of .NET array objects in ABL. (The difference between this `EventHandlers.p` example and `PointerArray.p` is that `PointerArray.p` draws the octagon only when you click a **Draw** button.) Therefore, for more information on the point calculations and array operations used in `EventHandlers.p`, see Example: Accessing a .NET array on page 246.

`EventHandlers.p` draws the octagon for all of the listed conditions by handling events on either the form or its buttons. It thus responds to all of these conditions by drawing the octagon for each of the following events:

- `Click` event on the `rDrawBtn` object
- `Click` event on the `rEraseBtn` object
- `Move` event on the `rForm` object
- `Paint` event on the `rForm` object

This is the mainline of `EventHandlers.p`, showing the code for UI initialization, event handler subscription, UI launch (displaying the form and blocking to handle the events), and procedure cleanup.

```
USING System.Windows.Forms.*  FROM ASSEMBLY.
USING Infragistics.Win.Misc.* FROM ASSEMBLY.
DEFINE VARIABLE rPointArray AS CLASS "System.Drawing.Point[]" NO-UNDO.
DEFINE VARIABLE rArray      AS CLASS System.Array             NO-UNDO.
DEFINE VARIABLE rPoint      AS CLASS System.Drawing.Point     NO-UNDO.
DEFINE VARIABLE rDrawBtn    AS CLASS UltraButton              NO-UNDO.
DEFINE VARIABLE rEraseBtn   AS CLASS UltraButton              NO-UNDO.
DEFINE VARIABLE rForm       AS CLASS Progress.Windows.Form    NO-UNDO.
DEFINE VARIABLE enColor     AS CLASS System.Drawing.Color     NO-UNDO.
DEFINE VARIABLE sqrt2       AS DECIMAL                        NO-UNDO.
DEFINE VARIABLE idx         AS INTEGER                        NO-UNDO.

FUNCTION AdjustOctagon RETURNS "System.Drawing.Point[]" FORWARD.

/* Create .NET Point array and a Point object to set its element values */
ASSIGN
  rPoint      = NEW System.Drawing.Point(0, 0)
  rArray      = System.Array:CreateInstance(rPoint:GetType( ), 8)
  rPointArray = CAST(rArray, "System.Drawing.Point[]").

/* Prepare UI to draw octagons */
ASSIGN
 rDrawBtn             = NEW UltraButton( )
  rDrawBtn:AutoSize     = TRUE
  rDrawBtn:Text         = "Draw"
  rDrawBtn:DialogResult = DialogResult:None
  rDrawBtn:Top          = 4
  rDrawBtn:Left         = 4.

ASSIGN
 rEraseBtn              = NEW UltraButton( )
  rEraseBtn:AutoSize     = TRUE
  rEraseBtn:Text         = "Erase"
  rEraseBtn:DialogResult = DialogResult:None
  rEraseBtn:Top          = 4
  rEraseBtn:Left         = 10 + (rDrawBtn:Width / 2).

ASSIGN
 rForm        = NEW Progress.Windows.Form( )
  rForm:Height = 300
  rForm:Width  = 300
  rForm:Text   = "Octagons".

rForm:Controls:Add(rDrawBtn).
rForm:Controls:Add(rEraseBtn).

ASSIGN
  sqrt2   = SQRT( 2.0 ) /* "Constant" needed to draw octagons */
  enColor = System.Drawing.Color:Black. /* Initial pen color */

/* Event handlers that draw and redraw the octagon */
rEraseBtn:Click:Subscribe("FormErase").
rDrawBtn:Click:Subscribe("FormDraw").
rForm:Move:Subscribe("FormMoveAndPaint").
rForm:Paint:Subscribe("FormMoveAndPaint").

WAIT-FOR rForm:ShowDialog( ).

rForm:Dispose( ).
```

The WAIT-FOR statement invokes the `ShowDialog( )` method to display the form as a modal dialog box and block for all events, including the button `Click` event to draw or erase the octagon. Closing the form then terminates the `WAIT-FOR` statement. For more information on this syntax for the `WAIT-FOR` statement, see Creating and Using Forms and Controls on page 89.

The effect is that clicking the **Draw** button makes the octagon visible and clicking the **Erase** button makes the octagon invisible, and the octagon continues to be visible or invisible while it changes its size, shape, and screen position as you resize or move the form until you click the other button.

Note: This effect works best if you set your Windows display properties so that Windows shows the contents of all windows when they are dragged. On Windows XP, you can find this setting by clicking the **Effects** button on the **Appearance** tab shown for the **Display** settings applet in the **Control Panel**.

So, `EventHandlers.p` uses an **Erase** button to erase the octagon and a **Draw** button to draw the octagon. In addition, it adds event handlers to maintain display of the form every time you change the form size or position. The event handlers ensure that all octagons remain erased once the **Erase** button is clicked or continue to be drawn once the **Draw** button is clicked. The octagons start out in a drawn state by initializing a `System.Drawing.Color` enumeration variable (`enColor`) with the `Black` enumeration member.

The `Draw` internal procedure is similar to the `Draw` internal procedure for the `PointArray.p` example described in Example: Accessing a .NET array on page 246. The difference in the version coded for `EventHandlers.p` is that this `Draw` procedure is not, itself, an event handler and therefore does not have event handler parameters.

```
PROCEDURE Draw:
  /* Draw octagon from point array */
  DEFINE VARIABLE rGraphics AS CLASS System.Drawing.Graphics NO-UNDO.
  DEFINE VARIABLE rPen      AS CLASS System.Drawing.Pen      NO-UNDO.

  ASSIGN
    rGraphics  = rForm:CreateGraphics( )
    rPen       = NEW System.Drawing.Pen(enColor)
    rPen:Width = 2.

  /* Pass adjusted point array to .NET method to draw octagon */
  rGraphics:DrawPolygon( rPen, AdjustOctagon( ) ).
END PROCEDURE.
```

Instead, this `Draw` procedure is called by the different event handlers in order to draw the octagon using the current pen color specified by `enColor`, which determines whether the octagon is actually drawn or erased.

The procedure continues with the same `AdjustOctagon` user-defined function and `CalcOctagonSide` internal procedure that are defined for `PointArray.p`.

```
FUNCTION AdjustOctagon RETURNS "System.Drawing.Point[]":
  ...
END FUNCTION.

PROCEDURE CalcOctagonSide:
  ...
END PROCEDURE.
```

This function and internal procedure implement the algorithm for calculating the points required to draw the octagon given the current dimensions of the form client area. For more information, see the description of these routines in Example: Accessing a .NET array on page 246.

Finally, EventHandlers.p defines its event handlers.

```
PROCEDURE FormErase:
  DEFINE INPUT PARAMETER sender AS CLASS System.Object    NO-UNDO.
  DEFINE INPUT PARAMETER e      AS CLASS System.EventArgs NO-UNDO.

  enColor = rForm:BackColor.
  RUN Draw.
END PROCEDURE.

PROCEDURE FormDraw:
  DEFINE INPUT PARAMETER sender AS CLASS System.Object    NO-UNDO.
  DEFINE INPUT PARAMETER e      AS CLASS System.EventArgs NO-UNDO.

  enColor = System.Drawing.Color:Black.  RUN Draw.
END PROCEDURE.

PROCEDURE FormMoveAndPaint:
  DEFINE INPUT PARAMETER sender AS CLASS System.Object    NO-UNDO.
  DEFINE INPUT PARAMETER e      AS CLASS System.EventArgs NO-UNDO.

  RUN Draw.
END PROCEDURE.
```

The difference between FormErase and FormDraw event handlers is in the pen color setting. To "erase" octagons, they are simply drawn with the color of the form background, and to "draw" them, they are drawn with the color black. The FormMoveAndPaint event handler responds to both the Move and Paint events by redrawing the octagon uses the most recent setting of enColor.

Note that the delegates for the Move and Paint events specify different event handler signatures. You can see this from the C# signatures for the different delegates shown in the Microsoft .NET Framework documentation. For example, this is the declaration for the EventHandler delegate used to define the Move event:

```
public delegate void EventHandler (Object sender, EventArgs e)
```

And this is the declaration for the PaintEventHandler delegate used to define the Paint event:

```
public delegate void PaintEventHandler (Object sender, PaintEventArgs e)
```

However, because the event handler required by EventHandlers.p for both events calls the Draw procedure in exactly the same way, and does not require access to its event arguments parameter, EventHandlers.p can define a single event handler for both events using the same signature based on the common base class for all event arguments (System.EventArgs).

Handling .NET exceptions

.NET error handling is based on a structured error handling model that encapsulates each error by an object type referred to as an exception. The base class for all .NET exceptions is System.Exception, and ABL allows you to trap System.Exception and all of its derived .NET Exception objects in much the same way as you trap ABL error objects.

In order to trap and handle .NET `Exception` objects in common with ABL error objects, OpenEdge has enhanced the .NET `System.Exception` class to implement the Progress.Lang.Error interface. Implementing this interface allows you to handle all .NET exceptions that are raised in the ABL context by both traditional and structured ABL error handling constructs.

If you use traditional error handling, you can trap a .NET exception raised from accessing .NET objects in a given block, and raise that exception to the next enclosing block, as with any ABL error.

If you use structured error handling, you can catch and throw a .NET `Exception` object and consult the same properties and methods as for a Progress.Lang.ProError object. However, these extended OpenEdge properties and methods function a little differently for a .NET exception than for an ABL error.

For more information on:

- The properties and methods available on .NET `Exception` objects, see Using properties and methods on .NET Exception objects on page 68.
- Differences in handling .NET exceptions compared to ABL errors, see Unique scenarios when handling errors with .NET objects on page 69.

Using the ABL error trapping constructs

Like ABL errors and error objects, you can handle .NET `Exception` objects by any of the ABL error trapping constructs:

- ON ERROR phrase
- `NO-ERROR` option on supported statements
- CATCH statement

This means that you can catch all errors with one `CATCH` block. The following code catches ABL system errors (Progress.Lang.SysError), ABL application errors (Progress.Lang.AppError), and .NET exceptions (System.Exception):

```
CATCH Progress.Lang.Error eError:
  ...
END CATCH.
```

You also can write `CATCH` blocks to handle .NET errors separately.

The following example shows an ABL class-based method that calls the .NET static `OpenRead( )` method to open a specified file for reading and return a .NET `System.IO.FileStream` object for it, which the ABL method then returns. The method contains three `CATCH` blocks to catch any errors, starting with the .NET `System.IO.FileNotFoundException`, followed by any other .NET exceptions, and finally by any ABL system errors. The AVM executes the first `CATCH` block matching the error type that is raised by the method, as shown:

```
METHOD PUBLIC CLASS System.IO.FileStream OpenReadFile
  (INPUT cPathname AS CHARACTER, INPUT bThrow AS LOGICAL):

  DEFINE VARIABLE rFileStream AS CLASS System.IO.FileStream NO-UNDO.

  rFileStream = System.IO.File:OpenRead( cPathname ).
  RETURN rFileStream.

  CATCH System.IO.FileNotFoundException eFileNotFound:
    ...
    IF bThrow THEN UNDO, THROW eFileNotFound.
  END CATCH.

  CATCH System.Exception eException:
    ...
    IF bThrow THEN UNDO, THROW eException.
  END CATCH.

  CATCH Progress.Lang.SysError eSys:
    ...
    IF bThrow THEN UNDO, THROW eSys.
  END CATCH.
END METHOD.
```

These `CATCH` blocks are organized in the standard fashion, where the most specialized blocks come first. (Otherwise, they can never be executed because any `CATCH` block for a super class before them always catches the specialized exception first.) In this case, the most likely .NET exception (`System.IO.FileNotFoundException`) occurs when the specified file does not exist. Note also that instead of ignoring the `Exception` objects after they are referenced, they can be re-thrown, depending on a condition passed into the ABL method (`bThrow`). Often, the decision whether to re-throw or consume an error object that is caught derives from the code in the `CATCH` block itself rather than being passed in as a `LOGICAL` parameter. In this case, the method assumes that the caller knows best whether the method should re-throw any error objects or consume then entirely within the method using garbage collection.

If you do not handle the .NET Exception object with either a CATCH block or with the `NO-ERROR` option, and the exception is not otherwise re-thrown by the containing ABL class or procedure using the ROUTINE-LEVEL ON ERROR UNDO, THROW statement, the AVM displays all the messages it contains to the current output device, including those in any `InnerException` objects. When you specify the Debug Alert (`-debugalert`) startup parameter or `SESSION:DEBUG-ALERT` is `TRUE`, the AVM adds the .NET stack trace to the Debug Alert information. The .NET stack trace is added both in the Debug Alert Help dialog box and in the client log (when the Client Log (`-clientlog`) startup parameter is specified). The top of the stack (most recent call) is displayed at the top of the trace listing.

If you handle the exception with the `NO-ERROR` option, as with all objects that implement Progress.Lang.Error, the messages populate the `ERROR-STATUS` system handle and the exception object is not accessible. These messages each represent the value of the `Message` property on the outer `Exception` object and on each `InnerException` object that has generated a message for the exception. These messages are identical to those retrieved using the GetMessage( ) method provided by `Progress.Lang.Error` and implemented by System.Exception, except that in addition to the text of the `Message` property for an `Exception` object, each message returned by `GetMessage( )` also indicates the type name of the .NET object from which the message originated. For more information, see the information on `GetMessage( )` in the following section.

Using properties and methods on .NET Exception objects

System.Exception supports a common set of Microsoft .NET properties and methods that are inherited by all .NET `Exception` objects. In addition, each derived `Exception` object can have its own unique set of properties and methods. In ABL, you can access all of these Microsoft properties and methods.

In addition, with implementation of the Progress.Lang.Error interface provided by OpenEdge, you can also access .NET `Exception` objects in a manner generally consistent with accessing ABL error objects, including Progress.Lang.ProError and all of its subclasses. This is particularly true of accessing .NET exception messages.

.NET `Exception` objects all support a `Message` property that returns the human-readable message for the exception. In addition, each `Exception` object supports an `InnerException` property that can reference another `Exception` object that has directly lead to throwing the current Exception object. Thus, this `InnerException` property can reference an exception chain with any number of `Exception` objects, each of which returns a message from its own `Message` property. Using the OpenEdge-extended `GetMessage( )` method, you can access every message from this entire chain of `Exception` objects without having to walk the `InnerException` object chain.

The following table shows a summary of all the OpenEdge-extended properties and methods, and how they work with .NET `Exception` objects.

Property or method	Description
`CallStack` property	Lists the ABL procedure and class call stack at the time a .NET method was called or a .NET data member or property was accessed that caused the exception. **Note:** The .NET `StackTrace` property is entirely different and lists only the stack of .NET calls within the CLR where the .NET exception originated.
`GetMessage ( n )` method	Returns the *n*th message in a message list. For a .NET `Exception` object, each message in addition to the first one in the list represents a separate inner exception message. Also, each such message indicates both the type name of the `Exception` object that generated the message and the text of its `Message` property. For example, if a System.ArgumentOutOfRangeException object is caught with its `Message` property set to `"Index 2 is out of range"`, this method returns a corresponding message with the following value: `"System.ArgumentOutOfRangeException: Index 2 is out of range"`
`GetMessageNum ( n )` method	Non-functional (always returns 0).
`NumMessages` property	Returns the number of messages listed for the `Exception` object, including all of its inner exceptions.
`Severity` property	Non-functional (always returns 0).

As noted at the start of this section, different types of .NET `Exception` objects can have their own unique set of properties and methods. For example, the `FileNotFoundException`, shown in the previous code fragment, has a `FileName` property that identifies the name of the file that cannot be found. You can only access such custom properties and methods directly on the .NET `Exception` object itself, and the information they provide is not available using OpenEdge-extended properties or methods. So, for an inner exception, you need to walk the exception chain to locate a given `Exception` object in order to access its custom properties and methods.

Unique scenarios when handling errors with .NET objects

ABL responds with some unique behavior when handling .NET exceptions that result from interactions with .NET user-interface events:

- Raising errors from ABL handlers for .NET events on page 69
- Handling .NET exceptions raised during display of a .NET form on page 69
- Raising errors from overridden .NET methods or implemented .NET interface members on page 70

Raising errors from ABL handlers for .NET events

When a .NET event is published to which you have subscribed ABL handlers (see Handling .NET events on page 57), if an error condition is raised from an ABL handler for the .NET event, the AVM does not throw an `Exception` to the .NET Common Language Runtime (CLR), but displays a message to the default output device and processing continues as if no error has occurred. Thus, if any handlers for the event have not yet run when a handler raises an error, all remaining handlers continue to run.

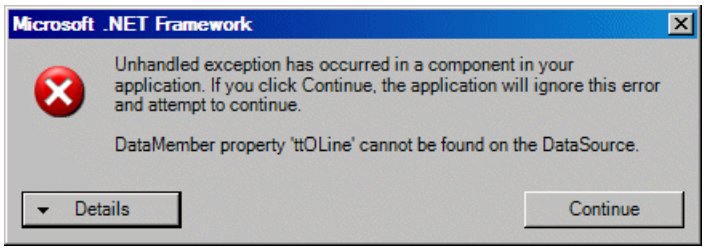
Note: This is different from the AVM response to an error condition raised from a handler for an ABL class event. In this case, the AVM raises the error on the statement that invoked the ABL Publish( ) method on the event, and any handlers that have not yet run for the event when the error is raised do not run.

Handling .NET exceptions raised during display of a .NET form

In an ABL application, .NET exceptions can occur when .NET user-interface components first display or when users interact with a .NET user interface. From the ABL viewpoint, all these exceptions occur within the context of a .NET `Show( )` method, WAIT-FOR, or PROCESS EVENTS statement that is currently running in an ABL session.

.NET typically traps most such exceptions within the CLR, before they have a chance to reach the ABL context. In this case, you might see a .NET alert box displayed like the one in the following figure.

Figure 4: Alert box from the Microsoft .NET Framework



This alert box displays a message that tells you how you might fix a bug that is typically in code that initializes .NET forms, controls, or related objects.

To handle the display of a .NET alert box in the ABL context:

1. Review the message to identify the problem that caused its display.
2. Click **Continue** to allow your ABL application and session to continue and see what you might do to fix the problem. Also note that sometimes the .NET alert box also provides a **Quit** button. Do **not** click **Quit** unless you want to immediately terminate your ABL session.

   However, some kinds of exceptions, such as a `System.AccesssViolationException`, cause `System.Windows.Forms.Application:Run( )` to terminate and raise a `STOP` condition on the executing `WAIT-FOR` or `PROCESS EVENTS` statement. You can trap this condition using the `ON STOP` phrase of an enclosing block. However, because all displayed forms are closed when `Application:Run( )` exits, the most that you can do is to clean-up and attempt a graceful shutdown of your application.

Raising errors from overridden .NET methods or implemented .NET interface members

If you raise an ABL error from within an overridden .NET method or an implemented .NET interface member, if the method or property is referenced from the .NET context, the AVM converts the error to a .NET `System.ApplicationException` according to how you raise the error. For more information, see Error handling for ABL-extended .NET classes on page 88.

Defining ABL-extended .NET objects

You can extend a .NET class with an ABL class much as you can extend another ABL class—by inheriting a .NET class or by implementing a .NET interface in the ABL class definition. As such, an *ABL-derived .NET class* is an ABL user-defined class that inherits from a .NET class. An *ABL-extended .NET class* is an ABL user-defined class that either inherits from a .NET class, implements a .NET interface, or both. Thus, the possible set of ABL-extended .NET classes represent a proper super set of the possible set of ABL-derived .NET classes. No matter how you define an ABL-extended .NET class, any instance of that class is accessible from both the ABL context and the .NET context of a single ABL session (see GUI for .NET run-time architecture on page 24).

The following sections describe how to define and use ABL-extended .NET classes, both by inheriting .NET classes and implementing .NET interfaces:

- Features of ABL-derived .NET classes on page 70
- Deriving .NET classes in ABL on page 72
- Managing events for ABL-derived .NET classes on page 81
- Features of ABL classes that implement .NET interfaces on page 85
- Implementing .NET interfaces in ABL on page 85
- Error handling for ABL-extended .NET classes on page 88

Features of ABL-derived .NET classes

ABL allows you to use an ABL class to derive (create a subclass of) many .NET classes, including `abstract` classes, with some restrictions. For more information on these restrictions, see Limitations of support for .NET classes on page 31.

As described previously (see Incorporation of the .NET object model on page 21), when an ABL class inherits from a .NET class, it appears at the bottom of the .NET class hierarchy exactly like any .NET class that inherits from the same hierarchy. This means that an ABL-derived .NET class inherits the same protected and public members in the .NET class hierarchy that a .NET derived class does (see Accessing .NET class members on page 45), including:

- Fields (data members)
- Methods
- Properties
- Events
- Nested (inner) types
- Static fields (data members)
- Static properties
- Static methods
- Static events

When an ABL class derives a .NET class, it can modify the data and behavior of the base class by:

- Overriding non-abstract instance methods inherited from the .NET class hierarchy with its own ABL `OVERRIDE` instance methods. For more information, see Overriding .NET methods on page 73.
- Overriding and implementing abstract properties, methods, and events of a .NET abstract base class. For more information, see Overriding .NET methods on page 73, Overriding .NET abstract properties on page 76, and Overriding .NET abstract events on page 77.
- Publishing events inherited from the .NET base class by calling methods that are typically defined for this purpose and inherited along with each event. For more information, see Managing events for ABL-derived .NET classes on page 81.
- Extending the .NET base class with its own ABL data members, properties, methods, and class events, including the overloading of inherited .NET methods. Such additional ABL class members can be added in exactly the same way as in a pure ABL class. However, these extended ABL members are only available to the ABL context and are not visible to the .NET context.

An ABL class cannot implement the following .NET modifications as part of deriving a .NET class:

- **Overriding non-abstract properties and events** — In .NET, a derived class can override non-abstract properties, methods, and events inherited from a super class. However, an ABL-derived .NET class can only override non-abstract methods inherited from a .NET super class (see Overriding .NET methods on page 73).
- **Hiding non-abstract members** — *Hiding* is a concept in .NET where a subclass can redefine a member of a super class without overriding it, by essentially replacing the inherited definition with a new one (using the `new` keyword in C#). ABL does not support the hiding of super class members, whether or not they are inherited from ABL or .NET. Any attempt to define a new member with the same name as non-abstract super class member results in a compile-time error, unless it is to explicitly override a method.

So, when an ABL application instantiates an ABL-derived .NET class, it can access all the supported public .NET class members on the ABL-derived instance, including subscribing to inherited .NET events. It can also access any `PUBLIC` ABL data members, properties, methods, and events defined by the ABL-derived class, including any non-abstract .NET methods that are overridden or overloaded by these ABL methods and any abstract .NET properties, methods, and events that are implemented by corresponding ABL properties, methods, and events.

ABL also supports polymorphic access to ABL-overridden .NET methods and ABL-implemented .NET abstract properties, methods, and events from both the ABL and the .NET run-time contexts. While .NET does not know about any ABL extensions, it can still use the ABL-extended class instance as it does the .NET super class or interface. If either .NET or ABL calls an ABL-overridden method on a super class reference to the ABL-derived instance, ABL ensures that the most derived method in the hierarchy (in this case, the overriding ABL method) is called. The same is true if either .NET or ABL calls an ABL-implemented abstract property method, or event. For more information on how the ABL and .NET run-time contexts interact, see GUI for .NET run-time architecture on page 24.

Deriving .NET classes in ABL

The minimum you need to do to derive a .NET class (like deriving any ABL class) is to inherit the class type in your ABL class definition:

```
CLASS Acme.Forms.CustomForm INHERITS Progress.Windows.Form:
  ...
END CLASS.
```

You can then use standard ABL class definition syntax to extend the .NET class with additional ABL data members, properties, methods, and events, including ABL methods that overload .NET methods. With the help of special ABL syntax (when necessary), you can also override .NET methods. As with inheriting from an ABL abstract class, if you inherit from a .NET abstract class, you can and must implement its abstract properties, methods, and events by overriding each abstract member if your inheriting ABL class is not also abstract.

You can also publish inherited .NET events, either by calling inherited .NET methods defined for this purpose or by invoking the ABL Publish( ) event method directly on inherited .NET abstract events that you implement in ABL. Finally, you can define ABL class events to delegate the handling of .NET events on privately contained .NET objects. This can be especially useful for encapsulating event management for .NET controls privately contained by an ABL-derived .NET user control. For more information, see Managing events for ABL-derived .NET classes on page 81.

Accessing inherited members of a .NET base class

Similar to accessing inherited members of an ABL base class, you can access inherited members of a .NET base class by referencing the member name and any other required member syntax as described in the following reference entries from OpenEdge Development: ABL Reference:

- Class-based data member access
- Class-based method call
- Class-based property access
- `Publish( )` event method (to publish abstract .NET events that you implement in ABL)
- `Subscribe( )` event method
- `Unsubscribe( )` event method

Caution: You cannot reliably access .NET super class members from a destructor of your ABL-derived class, because when the destructor executes .NET garbage collection might already have deleted the .NET components of your class.

Although you can make a naked reference to any inherited member name, prefixing an instance member name with the THIS-OBJECT system reference or prefixing a static member name with the type name of the defining class (static type-name reference) can help both readability and in distinguishing references to inherited and local class members from references to local constructor and method data elements of the same name. Also, as with inherited ABL class members, if an inherited .NET class member happens to have a name that is identical with an ABL reserved keyword, you must appropriately prefix the member name with either the `THIS-OBJECT` system reference or a static type-name reference when you reference the instance or static .NET member, respectively.

Defining new ABL members in a class that inherits from a .NET class

Note that when you define new methods in the definition of an ABL-derived .NET class, you might not be able to use the following method names, depending on their definition in the .NET class hierarchy:

- **`Get_`*`property-name`*`( )`** — Where *`property-name`* is the name of a property (including any default indexed property) defined by the .NET super class

  Note: For default indexed properties, *`property-name`* is usually `Item`.

- **`Set_`*`property-name`*`( )`** — Where *`property-name`* is the name of a property (including any default indexed property) defined by the .NET super class
- **`Add_`*`event-name`*`( )`** — Where *`event-name`* is the name of an event defined by the .NET super class
- **`Remove_`*`event-name`*`( )`** — Where *`event-name`* is the name of an event defined by the .NET super class

Similarly, as with any pure ABL class, you cannot define ABL data members, properties, or events with the same names as data members, properties, or events already defined in the inherited .NET class hierarchy.

Overriding .NET methods

In ABL, you can override any ABL method inherited by an ABL class using the `OVERRIDE` option of the METHOD statement. You use a similar ABL mechanism to override a .NET method. However in .NET, you can only override inherited methods that are defined to be overrideable. Similarly in ABL, you can only override an inherited .NET method that has been defined (in C#) as `virtual` or `abstract`.

Note: In .NET documentation, some methods that you can override might be declared with the C# `override` keyword instead of the `virtual` or `abstract` keyword. This simply means that the .NET method is itself an override of a `virtual` or `abstract` super class method higher up in the class hierarchy.

In addition, the method must be defined with one of the following .NET access level modifiers, which correspond to access modes in ABL:

- `public`
- `protected`
- `protected internal`

The overriding ABL method must have an ABL access mode that is equivalent to or less restrictive than the access level of the overridden .NET method. So, the ABL method access mode can correspond to the .NET access level modifier as follows:

- You can only override a `public` .NET method with a `PUBLIC` ABL method.

- You can override a `protected` or a `protected internal` .NET method with either a `PROTECTED` or a `PUBLIC` ABL method.

You cannot override a .NET method defined as any of the following:

- `static`
- `sealed` (`FINAL`)
- `DestroyHandle( )` method
- `Dispose( )` method
- `Finalize( )` method
- `GetHashCode( )` method

The listed methods are all defined as overrideable in .NET. However, ABL defines these methods as `FINAL` for an ABL session.

Caution: Although you can override the `WndProc( )` method, doing so can be risky. The method will be called for every event, like `WM_MOUSEMOVE` and `WM_PAINT`, so it can be called thousands of times for a single form, causing performance issues. It can also be called at unexpected times and cause unexpected behavior. Overriding `WndProc( )` when using ABL embedded windows with .NET forms can be especially problematic. For more information, see Using .NET Forms with ABL Windows on page 183.

As with overriding an inherited ABL method, the name, return type, and the signature of the overriding ABL method must match the overridden .NET method with respect to the number of parameters, the corresponding parameter modes, and the corresponding data types. For information on identifying .NET parameter modes, see Specifying .NET constructor and method parameters on page 50.

You can also define the overriding ABL method with or without the `ABSTRACT` option, whether or not the overridden .NET method is abstract. Note that to use the `ABSTRACT` option the inheriting ABL class must also be abstract. However, as for an inherited ABL abstract method, the first non-abstract ABL subclass that inherits a .NET abstract method must implement the inherited .NET abstract method unless an abstract ABL subclass implements the method higher in the class hierarchy.

As described previously, when you call an overloaded .NET method, you must explicitly indicate the .NET parameter data types that correspond to the particular method overloading you want to call (see Passing ABL data types to .NET constructor and method parameters on page 218). When overriding a .NET method, you must follow similar rules for how to define the parameters and return type in ABL based on how the parameters are defined in .NET. If the data type of a method element is a .NET mapped data type, and this mapped type **is** the default match for a given ABL primitive data type, you must specify the corresponding ABL primitive data type. However, if a .NET mapped type is **not** the default match of any ABL primitive data type, you must specify the data type using an `AS` data type keyword that represents the required .NET mapped type. See Table 11: Explicit mappings from ABL primitive to .NET data types on page 217 for information on both the .NET mapped types you must specify as default mappings and those you must specify using a corresponding `AS` data type.

For example, this is the C# declaration for a .NET method you can override:

```
/* C# overridable method declaration */
public virtual double Calculate( byte arg )
```

This is the ABL method prototype required to override this .NET method:

```
/* ABL overriding method */
METHOD PUBLIC OVERRIDE DOUBLE Calculate(piArg AS UNSIGNED-BYTE):
  ...
END.
```

Note that at run time, ABL interprets these parameter and return values as the corresponding ABL data types that map to their .NET equivalents. So, for example, ABL interprets `piArg` as an `INTEGER` and the return type as a `DECIMAL` value. So, to call this method in ABL, you might use the following code, where `derivedInstance` is a reference to an ABL class instance that defines the ABL override of `Calculate( )`:

```
/* ABL call to overriding method */
DEFINE VARIABLE dVar AS DECIMAL NO-UNDO.
DEFINE VARIABLE iVar AS INTEGER NO-UNDO.
...
ASSIGN
  iVar = 42  dVar = derivedInstance:Calculate( iVar ).
...
```

To override .NET methods that pass and return .NET object types, including .NET array types, you must define the parameters and return types using the equivalent .NET object types.

For example, this is the declaration for a .NET method you can override:

```
/* C# overridable method declaration */
public virtual System.IAsyncResult BeginReceiveFrom (
  byte[] buffer,
  int offset,
  int size,
  System.Net.Sockets.SocketFlags socketFlags,
  ref System.Net.EndPoint remoteEP,
  System.AsyncCallback callback,
  System.Object state
)
```

You might override this `BeginReceiveFrom( )` method in ABL as follows:

```
/* ABL overriding method */
METHOD PUBLIC OVERRIDE System.IAsyncResult BeginReceiveFrom
 (INPUT  prByteBuffer     AS "System.Byte[]",
  INPUT  piOffset         AS INTEGER,
  INPUT  piSize           AS INTEGER,
  INPUT  prSocketFlags    AS System.Net.Sockets.SocketFlags,
  INPUT-OUTPUT prRemoteEP AS System.Net.EndPoint,
  INPUT  prCallback       AS System.AsyncCallback,
  INPUT  prState          AS System.Object):
  ...
END.
```

Note: The Microsoft .NET Framework provides a `BeginReceiveFrom( )` method with a similar prototype that is not declared as `virtual`. Its prototype is borrowed and modified here in order to illustrate overriding.

Note that most of the .NET object type names for the `BeginReceiveFrom( )` method override transfer exactly. However, the C# primitive array type, `byte[]`, maps to the .NET array object type with the type name, `"System.Byte[]"`. You also **cannot** specify this .NET array type in ABL using the corresponding `AS` data type, such as in this case `UNSIGNED-BYTE EXTENT`. The C# primitive type, `int`, maps to the .NET `System.Int32`. However, because the default match for the ABL primitive type, `INTEGER`, is `System.Int32`, you simply use `INTEGER` to define the parameters with that .NET type. Finally, the .NET `ref` directive declares a parameter access level that corresponds to the ABL access mode, `INPUT-OUTPUT`. (In this example, the remaining parameter definitions also have the optional `INPUT` access mode specified for readability.)

The following table summarizes the rules for defining each .NET method parameter and return type for an ABL method that overrides a .NET method.

Table 2: Defining parameters and return types to override .NET methods[1]

To define this .NET element . . .	Follow this rule . . .
.NET mapped data type	Use either the default-matched ABL data type or the `AS` data type that maps to the corresponding .NET type, as specified in Table 11: Explicit mappings from ABL primitive to .NET data types on page 217.
.NET object type that is not mapped	Use the corresponding .NET object type. For .NET arrays, you cannot use the ABL `EXTENT` option—you must specify the equivalent .NET array object type surrounded in quotes (for example, use `"System.Byte[]"` or `"System.Drawing.Point[]"`, **not** `UNSIGNED-BYTE EXTENT` or `System.Drawing.Point EXTENT`). However, if the corresponding .NET array is specified as a `System.Array` object, you must also use `System.Array`.
.NET parameter mode	Use the corresponding ABL parameter mode specified in Table 1: C# syntax matching ABL parameter modes on page 51.

1 - You apply these same rules to define properties and the parameters and return types of methods that you implement in a .NET interface. For more information, see Implementing .NET interfaces in ABL on page 85.

Overriding .NET abstract properties

You can override a .NET abstract property (defined with the C# `abstract` option) inherited by an ABL class as long as the .NET abstract property is not an indexed property. The rules for overriding a .NET abstract property are very similar to overriding an ABL abstract property using the `OVERRIDE` option of the DEFINE PROPERTY statement.

For example, the overriding ABL property must have an ABL access mode that is equivalent to, or less restrictive than, the access level of the overridden .NET abstract property. The overriding ABL property must define the same accessor specified for the .NET abstract property, but can add the missing accessor if necessary. Thus, if the .NET abstract property has a C# `get`, the overriding ABL property must have a `GET`, and if the .NET property prototype has a C# `set`, the overriding ABL property must have a `SET`.

The overriding ABL property can also be defined as abstract (using the `ABSTRACT` option) as long as the inheriting ABL class is abstract. However, as for any ABL abstract property, the first non-abstract ABL subclass must implement the inherited .NET abstract property unless an abstract ABL subclass implements the property higher in the class hierarchy.

The primary difference in overriding a .NET abstract property is how to match the data type in the overriding ABL property definition when the .NET property has a .NET mapped data type. If the data type of the .NET abstract property is a .NET mapped data type, you must define the ABL property data type using the rules for explicitly mapping .NET data types. For more information, see Explicit data type mappings on page 216.

For example, you might have an ABL class implement an `Area` .NET abstract property from a .NET abstract super class, `Shape`, that is defined like this:

```
public abstractdoubleArea { get; }
```

In this example, you define an ABL `Rectangle` class to inherit the .NET `Shape` class and implement its abstract `Area` property:

```
CLASS Rectangle INHERITS Shape:

  DEFINE PUBLIC PROPERTY Width AS DECIMAL NO-UNDO
    GET.
    SET.
  DEFINE PUBLIC PROPERTY Height AS DECIMAL NO-UNDO
    GET.
    SET.

  DEFINE PUBLIC PROPERTY OVERRIDE Area AS DOUBLE NO-UNDO
    GET:
      /* Given the Width and Height, return the area of a rectangle */
      RETURN Width * Height.
    END.

...

END CLASS.
```

Overriding .NET abstract events

You can override a .NET abstract event (defined with the C# `abstract` option) inherited by an ABL class. The rules for overriding a .NET abstract event are very similar to overriding an ABL abstract event using the `OVERRIDE` option of the DEFINE EVENT statement. For example, the overriding ABL event must have an ABL access mode that is equivalent to, or less restrictive than, the access level of the overridden .NET abstract event. The overriding ABL event can also be defined as abstract using the `ABSTRACT` option as long as the inheriting ABL class is also abstract. However, as for any ABL abstract event, the first non-abstract ABL subclass must implement the inherited .NET abstract event unless an abstract ABL subclass implements the event higher in the class hierarchy.

To define the same signature as the .NET abstract event, you must use the `DELEGATE` option of the ABL `DEFINE EVENT` statement to specify the same .NET delegate type that is used to define the .NET abstract event. For more information on .NET delegate types, see Handling .NET events on page 57.

When you implement a .NET abstract event in an ABL class, you can directly publish the implemented event like any non-abstract ABL event defined in the overriding ABL class. For more information, see Managing events for ABL-derived .NET classes on page 81.

For example, you might have an ABL class implement the .NET abstract event, `DataEvent`, from the .NET abstract super class, `CheckScanner`, which is in the `Microsoft.PointOfService` namespace provided by Microsoft Point of Service for .NET as described on MSDN:

```
public abstract event DataEventHandler DataEvent
```

Among other abstract members, you have to implement the `RetrieveImage( )` method to get a check image from a particular check scanning device and the `ImageData` property to hold the scanned check image, the prototypes for which are defined in `CheckScanner` as follows:

```
public abstract void RetrieveImage ( int cropAreaId )

public abstract Bitmap ImageData { get; }
```

The `DataEvent` event is raised when a scanned check image is made available to the application. So, you might inherit the .NET `CheckScanner` class and implement this event (along with its other abstract members) in an ABL class defined as in the following `CheckProcessing.cls` class file fragment.

Note that one reason for .NET providing an abstract class, in this case, is to accommodate any possible scanning device that a given check scanning application needs to support.

This fragment starts out showing the `CheckProcessing` class inheriting the .NET `CheckScanner` class, defining some variable data members to hold the event arguments and a scanning status indicator, and implementing the abstract `DataEvent` event (with reference to the `DataEventHandler` delegate), the abstract `ImageData` property, and the abstract `RetrieveImage( )` method for getting and storing check images in the `ImageData` property.

The public `RetrieveImage( )` method enters a loop to retrieve scanned check images and publish `DataEvent` for each scan using the private `ScanCheck( )` method, possibly with the help of other members of the .NET abstract `CheckScanner` class (not shown). Note that `DataEvent` is published by `ScanCheck( )` regardless if check scanning input has ended or if a valid check image is available, leaving it to any subscribed event handler to process the results.

```
USING Microsoft.PointOfService.* FROM ASSEMBLY.
USING System.Drawing.* FROM ASSEMBLY.

CLASS CheckProcessing INHERITS CheckScanner:

  /* Variables for event arguments and status */
  DEFINE PRIVATE VARIABLE rEventArgs AS CLASS DataEventArgs NO-UNDO.
  DEFINE PRIVATE VARIABLE iStat AS INTEGER NO-UNDO.

  /* Implement abstract members from CheckScanner abstract class */
  DEFINE PUBLICOVERRIDE EVENT DataEvent DELEGATE CLASS DataEventHandler.

  DEFINE PUBLIC PROPERTY OVERRIDE ImageData AS Bitmap NO-UNDO
    GET...
    SET...

  METHOD PUBLIC OVERRIDE VOID RetrieveImage ( INPUT piCropAreaID AS INTEGER ):

    /* Get first check image and publish DataEvent */
    THIS-OBJECT:ScanCheck( piCropAreaID ).

    /* Loop to get more scanned check images and publish
       DataEvent while device is scanning checks */
    iStat = rEventArgs.Status. /* Status from DataEvent arguments */
```

```
    DO WHILE iStat = 0: /* Check scanner input is still available */
      ...
      THIS-OBJECT:ScanCheck( piCropAreaID ).
      iStat = rEventArgs.Status.
    END.

  END METHOD.
  /* Try to retrieve a check image and publish DataEvent with the result */
  METHOD PRIVATE OVERRIDE VOID ScanCheck ( INPUT piCropAreaID AS INTEGER ):

    IF /* device scanning check input */ THEN DO:
      rEventArgs = NEW DataEventArgs( 0 ). /* Device is scanning checks */
      IF /* check image available */ THEN DO:
        THIS-OBJECT:ImageData = /* Get check image from scanner */.
        /* May be do something with piCropAreaID */
        ...
      END.
      ELSE
        THIS-OBJECT:ImageData = ?. /* No image available */
    END.
    ELSE
      ASSIGN
        rEventArgs = NEW DataEventArgs( 1 ) /* Check scanning has ended */
        THIS-OBJECT:ImageData  = ?.

    THIS-OBJECT:DataEvent:Publish( THIS-OBJECT, rEventArgs ).

  END METHOD.
...

END CLASS.
```

When executed, the private `ScanCheck( )` method tests the check scanning device input status and instantiates the event arguments class (`DataEventArgs`) accordingly, passing the status value to the constructor and assigning its object reference to `rEventArgs`. It then gets the check image, if one is available, and assigns it to the `ImageData` property, or sets `ImageData` to the `Unknown value (?)` if a check image is not available. Finally, the method publishes `DataEvent`, passing parameters with the results according to the `DataEventHandler` delegate.

The following `ProcessCheckImages.p` is a procedure fragment that demonstrates how an ABL application might use the `CheckProcessing` class and its `DataEvent` event to process check images. The procedure first creates an instance of `CheckProcessing`, assigning its object reference to `rScanner`. The procedure then subscribes the internal procedure, `DataEvent_CheckHandler`, as a handler for the `DataEvent` on `rScanner`. Finally, it then calls `RetrieveImage( )` on `rScanner` to retrieve the image and publish the `DataEvent` for each scanned check.

The DataEvent_CheckHandler procedure is defined with a signature that is compatible with the signature specified by the DataEventHandler delegate, including the input System.Object (prSender), which is the instance of CheckProcessing that published the event, and the input DataEventArgs (prArgs), which is the specified event arguments class instance.

```
USING Microsoft.PointOfService.* FROM ASSEMBLY.
USING System.Drawing.* FROM ASSEMBLY.

DEFINE VARIABLE rCheck AS CLASS Bitmap NO-UNDO.
DEFINE VARIABLE rScanner AS CLASS CheckProcessing NO-UNDO.

rScanner = NEW CheckProcessing( ).
rScanner:DataEvent:Subscribe( DataEvent_CheckHandler ).

rScanner:RetreiveImage( 0 ). /* 0 CropAreaID = no image cropping */

  /* DataEvent handler with a DataEventHandler delegate signature */
  PROCEDURE DataEvent_CheckHandler:

    DEFINE INPUT PARAMETER prSender AS CLASS System.Object NO-UNDO.
    DEFINE INPUT PARAMETER prArgs AS CLASS DataEventArgs NO-UNDO.

    DEFINE VARIABLE rScanInstance AS CLASS CheckProcessing NO-UNDO.

    rScanInstance = CAST( prSender, CheckProcessing ).

    /* Process any valid check image returned by the ImageData property */
    iStat = prArgs:Status.
    IF iStat = 0 THEN /* Device is still scanning checks */
      IF VALID-OBJECT( rScanInstance:ImageData ) THEN DO:
        rCheck = rScanInstance:ImageData. /* Process check image */
        ...
      END. /* Otherwise, ignore this event without a check image */
    END.
    ELSE DO: /* Report scanning has ended and exit the procedure */
      MESSAGE "Check scanning has ended with a status of" iStat
        VIEW-AS ALERT-BOX.
    END.
  END PROCEDURE.
```

When a DataEvent is published on rScanner (during execution of the RetrieveImage( ) method), the DataEvent_CheckHandler procedure executes, first casting prSender to a CheckProcessing object reference (rScanInstance) in order to access CheckProcessing public members for the current scan result, especially the ImageData property, which holds any check image data. The event handler then checks the Status property on prArgs for the scanner input status and the ImageData property on rScanInstance to see if it holds a valid check image.

If a check image is available, the event handler processes the image. If the device is still scanning checks, but no check image is available for this event, the handler ignores the event. In either of these cases, the procedure continues with the next DataEvent to be published. If the scanner has scanned its last check, or otherwise stopped scanning, the handler displays a status message and RetrieveImage( ) also returns along with ProcessCheckImages.p.

Getting type information (reflection)

A call to the GetClass( ) method on an ABL-derived .NET class returns a Progress.Lang.Class instance. However, as the super class of the ABL class is a .NET class, the SuperClass property of Progress.Lang.Class returns the Unknown value (?).

An ABL-derived .NET class also inherits the .NET method, `GetType( )`, from `System.Object`. However, this inherited `GetType( )` method returns incomplete information that does not contain information about any new members that are part of the ABL class extension.

Managing events for ABL-derived .NET classes

Because an ABL-derived .NET class inherits the events of its .NET base class, you can subscribe to these events on an ABL-derived class instance in the same way you subscribe to the events of any .NET class. For more information, see Handling .NET events on page 57. You can also publish an inherited .NET event from within an ABL-derived class using any inherited .NET method provided to publish the event, and you can directly publish any inherited .NET abstract events that you implement using the ABL built-in Publish( ) event method.

Finally, you can define ABL events to delegate the management of events for .NET objects that you privately define in an ABL-derived .NET container class. For example, if you are implementing an ABL-derived .NET user control that contains a set of private .NET controls, this allows you to make existing .NET control events available outside an inherited control container without making the contained .NET controls public.

Publishing inherited .NET events

You can programmatically publish an event that an ABL-derived class inherits from a .NET object by calling an inherited method that the .NET base class defines to publish the event. Such inherited .NET methods for publishing events are typically defined as `protected` and, by convention, have the following general method calling sequence:

Syntax

```
On EventName( [ EventArgs ] )
```

Element descriptions for this syntax diagram follow:

EventName

Specifies the .NET name of the event.

eArgs

Passes an `INPUT` object reference argument defined as a `System.EventArgs` (or a derived class, depending on the method). This is the `System.EventArgs` argument that is passed as the second parameter to any handler method for this event (see Defining handlers for .NET events in ABL on page 59).

So, for example, the .NET method to publish the `FormClosing` event on a Progress.Windows.Form is the `OnFormClosing( )` method.

An ABL-derived class that inherits from a .NET class might need to both publish an inherited event and perform another action at the same time. For example, you might define an ABL `BlueButton` class that inherits from .NET's `System.Windows.Forms.Button`. For many events, such as the `Click` event, the .NET super class defines the method to fire the event as `virtual` (or `override`). You can thus override this event method like any other `virtual` method to provide additional event behavior.

However, if you override a method that .NET defines to publish an event, you must always invoke SUPER:On*EventName*( ) to publish the event on the base class to ensure that all subscribers receive the event. You can code your custom behavior before or after this `SUPER` method call, as necessary.

Publishing inherited .NET abstract events

If you override and implement an inherited .NET abstract event, you can use the ABL Publish( ) event method to publish the implemented event like any non-abstract ABL class event defined in the ABL derived class. The class must contain the non-abstract definition of the event. Like any inherited ABL abstract event, you cannot publish an overridden .NET abstract event in a class that defines it with the `ABSTRACT` and `OVERRIDE` options.

Handling events on controls contained by ABL-derived .NET classes

When an ABL-derived class is a container of other .NET controls, such as an ABL extension of Progress.Windows.UserControl (see Creating custom .NET forms and controls on page 110), the events on these contained controls might be of interest to the class containing the ABL-derived user control, such as an ABL extension of `Progress.Windows.Form`.

You can use two models to expose events on a .NET control contained by an ABL-derived user control to the ABL-derived form that contains the user control:

- **Make the contained controls public** — Make the object reference for the contained .NET controls public, allowing the ABL-derived form to subscribe directly to events on these controls.
- **Delegate the event publishing for privately contained controls to ABL class events** — If you do not want the contained .NET control to be public, publish public ABL class events on behalf of events on the private control. In other words, subscribe to a given event on the privately contained control and republish it as an ABL class event of the control container. The ABL form object can then respond to the public ABL event without having direct access to the private .NET control whose published event causes the ABL event to be published.

The following contrived class examples, `UserControl1` and `Form1`, show how you might handle events on public controls.

The ABL-derived user control (`UserControl1`) contains a `System.Windows.Forms.Button` control that is referenced by a `PUBLIC Button1` property.

```
CLASS UserControl1 INHERITS Progress.Windows.UserControl:
  DEFINE PUBLIC PROPERTY Button1 AS System.Windows.Forms.Button NO-UNDO
    GET.
    PRIVATE SET.

  CONSTRUCTOR UserControl1( ):
    Button1 = NEW System.Windows.Forms.Button( ).
    THIS-OBJECT:Controls:Add(Button1). /* Add button to user control */
    /* Set any other control properties */
    ...
  END CONSTRUCTOR.
  ...
END CLASS.
```

The ABL-derived form (`Form1`) subscribes to the `Click` event directly on the `Button1` property, like this.

```
CLASS Form1 INHERITS Progress.Windows.Form:
  DEFINE PRIVATE VARIABLE rUControl AS UserControl1 NO-UNDO.
  ...
  CONSTRUCTOR Form1( ):
    rUControl = NEW UserControl1(THIS-OBJECT).
    rUControl:Button1:Click:Subscribe(THIS_OBJECT:Button1_Click).
    THIS-OBJECT:Controls:Add(rUControl). /* Add user control to form */
    /* Set any other control properties */
    ...
  END CONSTRUCTOR.

  METHOD VOID Button1_Click(o AS System.Object, e AS System.EventArgs):
    /* Process the o and e parameters of the .NET click event */
    ...
  END METHOD.
END CLASS.
```

If you want an ABL-derived user control to keep all of its contained controls private, you can instead:

- Define an ABL class event that corresponds to each .NET control event whose handling you want to delegate. It could have the same or a different signature from the .NET control event, depending on how you need the client form to handle it.
- Define and subscribe an ABL method as a handler for each .NET control event. The primary action of this method is to publish the ABL event that corresponds to the control event it is handling. Thus, the form containing the ABL-derived user control can handle the user control event as a delegate for the privately contained .NET control event.

The code for the following contrived ABL-derived .NET container classes (`UserControl2` and `Form2`) demonstrates how to use an ABL event to delegate the handling of an event on a control that is contained privately and not directly accessible from outside its control container.

```
USING System.Windows.Forms.* FROM ASSEMBLY.
CLASS UserControl2 INHERITS Progress.Windows.UserControl:
  DEFINE PRIVATE VARIABLE rButton2 AS System.Windows.Forms.Button NO-UNDO.
  DEFINE PUBLIC EVENT UserControl2Click
    SIGNATURE VOID ( pcInfo AS CHARACTER, pArgs AS System.EventArgs ).
  ...
  CONSTRUCTOR UserControl2( ):
    rButton2 = NEW System.Windows.Forms.Button( ).
    /* Subscribe UserControl2 to the rButton2:Click event */
    rButton2:Click:Subscribe(THIS-OBJECT:Button2_Click).
    THIS-OBJECT:Controls:Add(rButton2). /* Add button to user control */
    /* Set any other control properties */
    ...
  END CONSTRUCTOR.

  METHOD VOID Button2_Click( sender AS System.Object,
                             e AS System.EventArgs ):
    DEFINE VARIABLE cInfo AS CHARACTER NO-UNDO.
    /* Set cInfo to the Text property of the control to identify it */
    cInfo = sender:Text.
    UserControl2Click:Publish (cInfo, e).
  END METHOD.
  ...
END CLASS.
```

The ABL-derived `UserControl2` defines a public ABL event (`UserControl2Click`) to delegate handling of the .NET `Click` event on a privately contained `Button` control, `rButton2`. To accomplish this, the ABL event handler that the user control defines and subscribes to this `Click` event (`Button2_Click( )`) publishes `UserControl2Click`. The container for the user control can then access the associated event behavior of the private button control by handling the public event that the user control defines for it.

Note that the parameter list defined for the `UserControl2Click` event includes an application-specified context string (`pcInfo`) and the `System.EventArgs` object (`e`) that is passed in the `Button2_Click( )` event handler parameter list. The purpose of `pcInfo`, in this case, is to pass a string to the subscribed event handler for `UserControl2Click` that identifies the private .NET control that has published the `Click` event while keeping the control object reference private. Thus, the ABL event parameter list ignores (and hides) the `sender` parameter, which is the object reference to the private control instance typically passed into a .NET control's event handler.

Note: Depending on the application, the delegating ABL event might also pass temp-tables and other application data with which a given control event is associated.

The following ABL-derived `Form2` then subscribes its own event handler (`UserControl2Click_Handler( )`) to the public `UserControl2Click` event on its contained ABL-derived `UserControl2` instance in order to respond to the `Click` event on the button that the user control privately contains.

```
CLASS Form2 INHERITS Progress.Windows.Form:
  DEFINE PRIVATE VARIABLE rUControl AS UserControl2 NO-UNDO.
  ...
  CONSTRUCTOR Form2( ):
    rUControl = NEW UserControl2( ).
    rUControl:UserControl2Click:Subscribe UserControl2Click_Handler.
    THIS-OBJECT:Controls:Add(rUControl). /* Add user control to form */
    /* Set any other control properties */
    ...
  END CONSTRUCTOR.

  METHOD VOID UserControl2Click_Handler
              ( pcInfo AS CHARACTER, e AS System.EventArgs ):
    /* Process the pcInfo and e parameters passed by the user control */
    ...
  END METHOD.
END CLASS.
```

In summary, an ABL event used to delegate the handling of privately contained control events provides the following features:

- The subscribing ABL-derived form does not determine what events it can handle on .NET controls contained by its user control. Only the ABL-derived user control determines the events that its client form can handle.
- The delegating ABL event defined by the ABL-derived user control passes all information about the private control that fires a .NET event to any subscribed handler for the delegating ABL event, allowing the defining user control to effectively prevent any direct reference to the private control from its client form.
- Note that this approach might be practical only for simple controls and a few number of events. In some cases, it is impossible to properly handle a control event without an object reference to the original publisher. For example, if the privately contained control is a grid, there are too many events associated with the rows and columns of a grid to delegate in this way, and any practical response to some of these events requires direct access to the grid reference.

Features of ABL classes that implement .NET interfaces

ABL allows you to implement .NET interfaces with some restrictions. For more information on these restrictions, see Limitations of support for .NET classes on page 31. The primary reason to implement a .NET interface is to define an ABL class with data and behavior that is accessible from the .NET context using the standard contract defined by the interface.

The restrictions on implementing .NET interfaces in ABL basically mean that you can implement .NET interfaces that define only the types of class members that are supported by ABL interfaces—properties, methods, and events (but no indexed properties). Attempting to implement a .NET interface that defines other types of class members raises an ABL compile-time error.

As with implementing ABL interfaces, you must implement all of the properties, methods, and events that a .NET interface defines. Note that like ABL interfaces, .NET interfaces can inherit member prototypes from other interfaces. In addition, you can define ABL interfaces that inherit .NET member prototypes from .NET interfaces. This means that ABL must support any inherited interfaces, and you must implement all the properties, methods, and events specified in the entire ABL and .NET interface hierarchy.

For example, the `System.Collections.ICollection` interface inherits from the `System.Collections.IEnumerable` interface. Therefore, if you implement `ICollection` you must also provide an implementation for the single `GetEnumerator( )` method defined by `IEnumerable`. Note, in this case, that `GetEnumerator( )` returns an object defined as a `System.Collections.IEnumerator` interface type. So you must also implement the `IEnumerator` interface, as well, typically in a separate class definition that provides the member implementations in the object that `GetEnumerator( )` returns.

Implementing .NET interfaces in ABL

As with overriding methods of an inherited .NET class (see Overriding .NET methods on page 73), when you implement a method of a .NET interface, you must define the method in ABL exactly as specified by the .NET method prototype. This means that the name, return type, and the signature of the implementing ABL method must match the .NET method prototype with respect to the number of parameters, the corresponding parameter modes, and the corresponding data types. Define the method parameters exactly as you do for the return values and parameters of ABL-overridden .NET methods (see Overriding .NET methods on page 73). See Table 2: Defining parameters and return types to override .NET methods1 on page 76 for a summary of the same rules for defining data types and parameter modes for ABL-overridden .NET methods.

When you implement a property of a .NET interface, you must define the ABL property with an implementation that is compatible with the specified .NET property prototype. So, your ABL property must define the same property name, the same .NET data type, and a compatible pattern of `GET` and `SET` accessors. If the .NET property prototype has a C# `get`, the implementing ABL property must have a `GET` accessor, and if the .NET property prototype has a C# `set`, the implementing ABL property must have a `SET` accessor. However, if the .NET property prototype specifies only one accessor, the implementing ABL property can add an implementation for the missing one. If the data type of the interface property prototype is a .NET mapped data type, you must define the ABL property data type using the rules for explicitly mapping .NET data types. For more information, see Explicit data type mappings on page 216.

When you implement an event of a .NET interface, you must define the event exactly as specified by the .NET event prototype, defining the same event name and signature. To define the same signature as a .NET event prototype, you must use the `DELEGATE` option of the ABL DEFINE EVENT statement to specify the same .NET delegate type that is used to define the event prototype. For more information on .NET delegate types, see Handling .NET events on page 57.

As with ABL, all .NET interface members are `PUBLIC`.

For example, if you wanted to implement the .NET `System.Collections.ICollection` interface, this is the C# declaration for it:

```
public interface System.Collections.ICollection :
   System.Collections.IEnumerable /* Inherited interface */
{
   int Count { get; }                /* Property */
   bool IsSynchronized { get; }      /* Property */
   System.Object SyncRoot { get; }   /* Property */

   void CopyTo (                     /* Method */
     System.Array array,
     int index
   )
}
```

This is the C# declaration for the inherited `System.Collections.IEnumerable` interface:

```
public interface System.Collections.IEnumerable
{
   System.Collections.IEnumerator GetEnumerator ( )    /* Method */
}
```

Finally, this is the C# declaration for the `System.Collections.IEnumerator` interface, the object type returned by the `GetEnumerator( )` method:

```
public interface System.Collections.IEnumerator
{
   System.Object Current { get; }  /* Property */

   bool MoveNext ( )               /* Method */
   void Reset ( )                  /* Method */
}
```

The following ABL `CustNameCollection` class then implements the `ICollection` and `IEnumerable` interfaces:

```
USING Progress.Lang.* FROM PROPATH.
USING System.Collections.* FROM ASSEMBLY.
ROUTINE-LEVEL ON ERROR UNDO, THROW.

CLASS CustNameCollection IMPLEMENTS ICollection, IEnumerable :
  DEFINE PUBLIC PROPERTY Count AS INTEGER NO-UNDO
    GET( ):
      RETURN THIS-OBJECT:Count.
    END.
    PRIVATE SET.

  DEFINE PUBLIC PROPERTY IsSynchronized AS LOGICAL NO-UNDO
    GET( ):
      UNDO, THROW NEW System.NotImplementedException( ).
    END.

  DEFINE PUBLIC PROPERTY SyncRoot AS System.Object NO-UNDO
    GET( ):
      UNDO, THROW NEW System.NotImplementedException( ).
    END.
```

```
  METHOD PUBLIC VOID CopyTo(
    pArray AS System.Array,
    pIndex AS INTEGER ):
      UNDO, THROW NEW System.NotImplementedException( ).
  END METHOD.

  METHOD PUBLIC IEnumerator GetEnumerator( ):
    RETURN NEW CustNameEnumerator(OUTPUT Count).
  END METHOD.
END CLASS.
```

In this case, while all the properties and methods of these interfaces are implemented, only the `Count` property of the `ICollection` interface and the `GetEnumerator( )` method of the `IEnumerable` interface are implemented with any functional behavior. The remaining members of `CustNameCollection` throw the .NET `System.NotImplementedException` object when accessed, because this is the standard exception to throw in .NET when you do not implement functionality for a member of an interface or in an overridden method.

The `GetEnumerator( )` method returns an `IEnumerator` object implemented by the ABL `CustNameEnumerator` class, which creates an ABL query on the `Customer` table of the Sports2000 database. The constructor returns the number of records in the opened query as an `OUTPUT` parameter and the `GetEnumerator( )` method stores the result in the `Count` property.

Note: While implementing these interfaces does allow you to access ABL data through a standard .NET mechanism, you typically use data binding to associate data with a class in .NET. For more information on using data binding in ABL see Binding ABL Data to .NET Controls on page 131.

The following ABL `CustNameEnumerator` class implements the `IEnumerator` interface:

```
USING System.Collections.* FROM ASSEMBLY.

CLASS CustNameEnumerator IMPLEMENTS IEnumerator :
  DEFINE PRIVATE QUERY qCust FOR Customer SCROLLING.

  DEFINE PUBLIC PROPERTY Current AS System.Object NO-UNDO
  GET( ):
    GET CURRENT qCust NO-LOCK.
    THIS-OBJECT:Current = IF AVAILABLE(Customer) THEN Customer.Name ELSE ?.
    RETURN THIS-OBJECT:Current.
  END.
  PRIVATE SET.

  METHOD PUBLIC LOGICAL MoveNext( ):
    GET NEXT qCust NO-LOCK.
  END METHOD.

  METHOD PUBLIC VOID Reset( ):
    GET FIRST qCust NO-LOCK.
    GET PREV qCust NO-LOCK.
  END METHOD.

  CONSTRUCTOR CustNameEnumerator(OUTPUT piCount AS INTEGER):
    OPEN QUERY qCust PRESELECT EACH Customer.
    piCount = QUERY qCust:HANDLE:NUM-RESULTS.
  END CONSTRUCTOR.
END CLASS.
```

This class implements all the members of the `IEnumerator` interface, which allow you to get the value of the `Name` field returned by a private query data member from a given record of the `Customer` table. The `Current` property returns the current `Name` value or the `Unknown value (?)` if the query has just been opened, is at the end, or has been reset to the beginning by the `Reset( )` method, and the `MoveNext( )` method gets the next record in the query. Note that the class constructor uses the `NUM-RESULTS` attribute on the query handle to return the value for the `Count` property of `ICollection`. Thus, all the implemented .NET properties and methods function and interpret ABL data in a manner that is understandable in the .NET context.

Error handling for ABL-extended .NET classes

As noted previously, an ABL method that overrides a .NET method, or that implements a method defined in a .NET interface, can be called from the .NET context. Similarly, an ABL property that implements a property defined in a .NET interface can be accessed from the .NET context. When you raise `ERROR` from within such an ABL method or property accessor that is invoked from the .NET context, and the specified ABL error options raise the error condition out of the method or accessor block, ABL returns a .NET `System.ApplicationException` to the caller.

If you raise the error condition by executing a `RETURN ERROR` with the optional error string, the `Message` property of the `System.ApplicationException` is available to the ABL context using the `RETURN-VALUE` function. If you raise the error condition by executing a `RETURN ERROR` with an optional ABL error object, or by executing an `UNDO, THROW` of the ABL error object, the `Message` property of the `System.ApplicationException` includes any messages from the ABL error object.

Note that ABL responds to errors raised from ABL handlers that you subscribe to ABL-inherited .NET events in exactly the same way as errors raised from ABL handlers of .NET instance or static events published by pure .NET classes. For more information, see Handling .NET exceptions raised during display of a .NET form on page 69.

3

Creating and Using Forms and Controls

The primary advantage of accessing .NET objects from ABL is to build a .NET user interface (UI) for your application using the OpenEdge GUI for .NET. In addition to the general ABL support for accessing and managing .NET classes (see Accessing and Managing .NET Classes from ABL on page 29), OpenEdge supports the GUI for .NET with a custom set of .NET classes and interfaces that allow the .NET forms of the GUI for .NET to work more naturally together in the same ABL session with ABL windows of the traditional OpenEdge GUI. This set of OpenEdge .NET classes also provides support for binding data from OpenEdge data sources to .NET controls. In addition, the Visual Designer of Progress Developer Studio for OpenEdge provides built-in support for a set .NET controls that you can use as components to design GUI for .NET user interfaces. These design components include a basic set of Microsoft .NET UI Controls and a more advanced set of OpenEdge Ultra Controls for .NET from Infragistics. You can also add third-party controls to enhance the installed design palette of the Visual Designer.

This chapter provides an introduction to the OpenEdge .NET classes and interfaces, the featured OpenEdge Ultra Controls, and how to use them to start building an OpenEdge GUI for .NET for your ABL application. For more information on binding OpenEdge data to .NET controls, see Binding ABL Data to .NET Controls on page 131 For more information on using .NET forms from the GUI for .NET with ABL windows from the traditional OpenEdge GUI, see Using .NET Forms with ABL Windows on page 183

For details, see the following topics:

- OpenEdge .NET form and control objects
- ABL support for managing .NET forms and controls
- Creating custom .NET forms and controls

OpenEdge .NET form and control objects

Previous chapters include several sample ABL procedures and code fragments for describing basic ABL support for accessing .NET objects. These samples use .NET forms and controls in very basic ways to demonstrate this ABL support for .NET. This section describes all the .NET form and control objects for which OpenEdge provides installed support for the OpenEdge GUI for .NET, and it briefly describes some of the more common .NET methods, properties, and events that you might use on many of these objects:

- Progress.Windows.Form **class** — `Progress.Windows.Form` inherits directly from the Microsoft .NET general form class, `System.Windows.Forms.Form`. As such, it provides the foundation for building the three basic types of general-purpose forms available to an OpenEdge GUI for .NET application:
  - **Non-modal forms** — Similar to ABL non-modal windows
  - **Modal forms (dialog boxes)** — Similar to ABL modal dialog boxes
  - **Multiple document interface (MDI) forms** — Not supported by the traditional OpenEdge GUI

  As a .NET derived class, `Progress.Windows.Form` provides all of the .NET methods, properties, and events that the Microsoft `System.Windows.Forms.Form` class provides. Thus, the type of form that you can create with a `Progress.Windows.Form` instance depends on how you use its properties, methods, and events, and you can use them in almost exactly the same way as you would use them to implement these form types using a .NET language, such as C# or Visual Basic. The differences lie in the uniquely ABL syntax that you use to interact with forms and the ABL restrictions on access to .NET features in general (see Supported features and limitations on page 30). This chapter provides more information on using the `Progress.Windows.Form` class to create and manage these basic form types.

  In addition, `Progress.Windows.Form` supports properties that are specifically used to manage .NET forms together with ABL windows in an ABL session. For more information, see Using .NET Forms with ABL Windows on page 183. Some of these properties can also be used to manage .NET forms when no ABL windows are present. For more information, see ABL support for managing .NET forms and controls on page 98.

- Progress.Windows.FormProxy **class** — `Progress.Windows.FormProxy` inherits directly from the Microsoft .NET root class, `System.Object`. Whenever you use .NET forms in an ABL session, ABL creates and associates a separate instance of this class with every ABL window that you create in the session. This class association provides a means to access .NET forms and ABL windows from one object chain. Its constructor is private. In addition to the members of `System.Object`, this class supports a set of OpenEdge properties in common with `Progress.Windows.Form`. For more information on the `Progress.Windows.FormProxy` class, and how it relates to both .NET forms and ABL windows, see Using .NET Forms with ABL Windows on page 183.

- Progress.Windows.IForm **interface** — `Progress.Windows.IForm` is a .NET interface that is implemented by both `Progress.Windows.Form` and `Progress.Windows.FormProxy`. It provides the common definition for the OpenEdge properties supported by these classes. As an object type, this interface allows you to reference both .NET forms and ABL windows together in a common manner. For more information on using this interface for common access to .NET forms and ABL windows, see Using .NET Forms with ABL Windows on page 183.

- **`Progress.Windows.MDIChildForm` class** — `Progress.Windows.MDIChildForm` inherits directly from `Progress.Windows.Form`. It provides an OpenEdge built-in form designed specifically for use as an MDI child form whose client area embeds the client area of a single ABL window. This allows you to display the client-area widgets of an ABL window in the MDI child form and interact with them as native ABL widgets using triggers instead of as controls of a .NET form using handlers on .NET events. For more information on using this class to create MDI child forms and to embed the client area of an ABL window, see Using .NET Forms with ABL Windows on page 183.

- Progress.Windows.UserControl **class** — `Progress.Windows.UserControl` inherits directly from the Microsoft .NET class, `System.Windows.Forms.UserControl`. This is a control container class that provides a container for user-defined control sets. For more information on using this class, see Sample ABL-derived .NET user control on page 126.
- **`Progress.Windows.WindowContainer` class** — `Progress.Windows.WindowContainer` inherits directly from the Microsoft .NET class, `System.Windows.Forms.UserControl`. This is a control container class used specifically to embed the client area of an ABL window. This allows you to display the client-area widgets of an ABL window in the client area of any .NET form and interact with them as native ABL widgets using triggers instead of as controls of the .NET form using handlers on .NET events. Using multiple instances of this control container, you can embed the client areas of multiple ABL windows in a single .NET form. For more information on using this class, see Using .NET Forms with ABL Windows on page 183.
- **Microsoft .NET UI Controls** — A subset of Microsoft .NET Framework controls that OpenEdge installs for access as design components using the Visual Designer of Progress Developer Studio for OpenEdge. These are from the same set of .NET Framework controls that you can access in ABL code that you write using any code editor. For a descriptive list of these controls, see Microsoft .NET UI Controls on page 202.
- **OpenEdge Ultra Controls for .NET** — Controls provided with Infragistics NetAdvantage for NET 2013, Volume 1 (CLR 4.0) that OpenEdge installs for access as design components using the Visual Designer of Progress Developer Studio for OpenEdge. You can also access these controls in ABL code that you write using any code editor. These controls provide a different combination of features than are provided by similar controls in the Microsoft .NET Framework, and they also include additional controls with features not available in the .NET Framework. For a descriptive list of these controls, see OpenEdge Ultra Controls for .NET on page 202.

The sections that follow briefly describe some of the more common methods, properties, and events that you might use on an OpenEdge form or control. For a more complete list of commonly-used .NET class members, see OpenEdge Development: GUI for .NET Mapping Reference.

Commonly-used .NET public form methods, properties, and events

The tables that follow describe some of the more commonly-used .NET public methods, properties, and events supported by Progress.Windows.Form. These tables contain brief descriptions for your orientation only. .NET supports many more public form methods, properties, and events than are described here. For detailed information on each `Form` class member, see the .NET Framework class library documentation on the `System.Windows.Forms.Form` class. For on-line access to this documentation, see Microsoft .NET UI Controls on page 202.

Common .NET public form methods

The following table shows some of the more common .NET public form methods.

Table 3: Common .NET public form methods

Method	Description
`Activate( )`	Brings the form to the front and publishes the `Activated` event.
`Close( )`	Closes the open form, publishing the `FormClosing` event followed by the `FormClosed` event. This method also calls the `Dispose( )` method on the form.
`Dispose( )`	See Table 6: Common .NET public control methods on page 95.
`Hide( )`	Conceals the form from the user by setting its `Visible` property to `FALSE`.

Method	Description
`ResumeLayout( )`	See Table 6: Common .NET public control methods on page 95.
`Show( )`	Initializes the form for display or reveals the form to the user by setting its `Visible` property to `TRUE`. You do not always need to call this method to initially display the form, depending on how you block for input on the form. For more information, see ABL support for managing .NET forms and controls on page 98.
`ShowDialog( )`	Displays and blocks for input on the form as a modal dialog box. You can **only** call this method using the `WAIT-FOR` statement for .NET forms (.NET WAIT-FOR statement). For more information see ABL support for managing .NET forms and controls on page 98. **Note:** To block for input on the form as a non-modal window (whether for a single form or for an MDI form), you must call the static `Run( )` method on the `System.Windows.Forms.Application` class from within a .NET `WAIT-FOR` statement.
`SuspendLayout( )`	See Table 6: Common .NET public control methods on page 95.

Other common methods for controls might also apply to forms. For more information, see Table 6: Common .NET public control methods on page 95.

Common .NET public form properties

The following table shows some of the more common .NET public form properties.

Table 4: Common .NET public form properties

Property	Description
`AcceptButton`	Gets or sets the object reference to the button control on the form whose `Click` event is published when the user presses the **ENTER** key.
`Bounds`	See Table 7: Common .NET public control properties on page 96.
`CancelButton`	Gets or sets an object reference to the button control on the form whose `Click` event is published when the user presses the **ESC** key.
`ClientSize`	See Table 7: Common .NET public control properties on page 96.
`ControlBox`	Gets or sets an ABL `LOGICAL` value that indicates if the control box icon is displayed in the caption (title) bar of the form. The control box is a system control that provides access to the system menu for the form. The icon used to represent it is specified by the `Icon` property.
`Controls`	See Table 7: Common .NET public control properties on page 96.

Property	Description
DialogResult	Gets or sets a System.Windows.Forms.DialogResult enumeration value that indicates the result of a form displayed as a dialog box. For more information, see Sample ABL-derived .NET modal dialog box on page 115.
FormBorderStyle	Gets or sets a System.Windows.Forms.FormBorderStyle enumeration value that indicates whether the form can be resized by the user and how the graphics of the frame border appear and function.
Icon	Gets or sets the object reference to a System.Drawing.Icon class instance that represents the icon displayed for the form on the taskbar and also the icon for the control box enabled by the ControlBox property.
Location	Gets or sets a System.Drawing.Point structure that specifies the pixel screen coordinates of the upper-left-hand corner of a form.
MainMenuStrip	Gets or sets the object reference to a System.Windows.Forms.MenuStrip class instance that represents the primary menu container for the form.
MaximizeBox	Gets or sets an ABL LOGICAL value indicating whether the **Maximize** button is displayed in the title bar of the form.
MinimizeBox	Gets or sets an ABL LOGICAL value indicating whether the **Minimize** button is displayed in the title bar of the form.
Modal	Gets an ABL LOGICAL value indicating if the form is displayed modally (as a dialog box).
Owner	Gets or sets the form that owns this form. OpenEdge also allows .NET forms to own ABL windows and for ABL windows to own .NET forms using the ABL PARENT attribute. For more information, see Using .NET Forms with ABL Windows on page 183.
Size	Gets or sets a System.Drawing.Size structure that specifies the width and height of the form in pixels.
Text	Gets or sets an ABL CHARACTER value that represents the title displayed in the title (caption) bar of the form.
Visible	Gets or sets an ABL LOGICAL value that indicates whether the form is visible or can be displayed on the screen. The form Show( ) method also sets this property to TRUE. You do not always need to initialize this property to TRUE in order to initially display the form, depending on how you block for input on the form. For more information, see ABL support for managing .NET forms and controls on page 98.

Other common properties for controls might also apply to forms. For more information, see Table 7: Common .NET public control properties on page 96.

Common .NET public form events

The following table shows some of the more common .NET public form events. Other common events for controls might also apply to forms. For more information, see Table 7: Common .NET public control properties on page 96.

Table 5: Common .NET public form events

Event	Description
`Activated`	Raised when the form receives focus. The form also publishes this event when you call the form `Activate( )` method.
`Deactivate`	Raised when the form loses focus.
`FormClosed`	Raised when the form is closed by the user, by the form `Close( )` method, or by the static `Exit( )` method on the `System.Windows.Forms.Application` class.
`FormClosing`	Raised as the form is closing, but before the `FormClosed` event is raised. You can set the `Cancel` property on the `FormClosingEventArgs` parameter when handling this event to cancel the close of the form and prevent the `FormClosed` event from being raised.
`Load`	Raised before the form is displayed for the first time, allowing you to allocate resources for and further configure the form before it actually becomes visible.
`Shown`	Raised immediately after the form is displayed for the first time.

Commonly-used .NET public control methods, properties, and events

The tables that follow describe some of the more commonly-used .NET public methods, properties, and events supported by `System.Windows.Forms.Control`. This is the base class for all .NET controls and control containers, including forms. These tables contain brief descriptions of the more common class members. .NET supports many more public methods, properties, and events on `System.Windows.Forms.Control` than are described, here. For detailed information on each `Control` class member and the unique members of the different derived controls, see the .NET Framework class library documentation on this class. For on-line access to this documentation, see Microsoft .NET UI Controls on page 202.

Common .NET public control methods

The following table shows some of the more common .NET public control methods.

Table 6: Common .NET public control methods

Method	Description
`Dispose( )`	Releases all resources held by the control or control container in preparation for garbage collection. You cannot override this method, as it is defined as `FINAL` in ABL. Note that the object on which you call this method is not garbage collected immediately and the object reference continues to work until .NET garbage collects the object. **Note:** Typically, you never have to call this method directly in ABL. If you execute the ABL DELETE OBJECT statement on the form or control object reference, this sufficiently prepares the .NET object for garbage collection wherever you might otherwise call the `Dispose( )` method in .NET. However, for .NET modal forms (dialog boxes, including ABL-derived forms), you must call this method to ensure that the form object is garbage collected. For more information, see Blocking on modal dialog boxes on page 103. Before executing `DELETE OBJECT`, you should also call this method on any instances of `Progress.Data.BindingSource` that you use to bind a ProDataSet to a .NET control. For more information, see Using the Dispose( ) method on page 156.
`Focus( )`	Sets input focus on the control.
`Hide( )`	Conceals the control or control container from the user by setting its `Visible` property to `FALSE`.
`ResumeLayout( )`	Resumes layout logic for the control or control container after it was suspended using the `SuspendLayout( )` method. Typically invoked on a form or other control container with a single `LOGICAL` argument set to `FALSE` in order to suppress the immediate resumption of pending layout logic until after all initialization of the control container is complete. This represents a performance improvement, especially for initializing control containers that have large numbers of child controls added.
`Show( )`	Displays the control or control container to the user by setting its `Visible` property to `TRUE`.
`SuspendLayout( )`	Temporarily suspends layout of the control, form, or other control container in order to prevent `Control.Layout` events from firing unnecessarily while completing control initialization. After control container initialization completes, you invoke the `ResumeLayout( )` method in order to allow the control initialization settings to take effect.

Common .NET public control properties

The following table shows some of the more common .NET public control properties.

Table 7: Common .NET public control properties

Property	Description
`Anchor`	Gets or sets a `System.Windows.Forms.AnchorStyles` enumeration value that specifies the edges of the container to which a control is bound and determines how a control is resized with its parent container. **Note:** This property is mutually exclusive with and manages somewhat different behavior than the `Dock` property. The last property that you set takes precedence over the other.
`Bottom`	Gets or sets an ABL `INTEGER` value that specifies the pixel distance between the bottom edge of the control and the top edge of the client area of its control container.
`Bounds`	Gets or sets a `System.Drawing.Rectangle` structure that specifies the location and size of the control or control container within the client area of its control container. For a form, the container is the screen area.
`ClientSize`	Gets or sets a `System.Drawing.Size` structure that specifies the client area of the control or control container. The client area is everything within the bounds of the control or control container except non-client elements, such as the title bar of a form.
`Controls`	Gets the object reference to the `System.Windows.Forms.ControlCollection` class instance for the form or other control container. Every .NET form or control container has a control collection where you can add or remove all the application controls that the form or control container contains, such as button, text, grid, and menu controls that you create and initialize for the form or control container. For more information on working with control collections, see Working with collections on page 56.
`Dock`	Gets or sets a `System.Windows.Forms.DockStyles` enumeration value that specifies which borders of the control are docked to its parent container and determines how a control is resized with its parent. **Note:** This property is mutually exclusive with and manages somewhat different behavior than the `Anchor` property. The last property that you set takes precedence over the other.
`Enabled`	Gets or sets an ABL `LOGICAL` value indicating whether the control can respond to user interaction.
`Font`	Gets or sets the font for the control or control container.
`Height`	Gets or sets an ABL `INTEGER` value that specifies the pixel height of the control.
`Left`	Gets or sets an ABL `INTEGER` value that specifies the pixel distance between the left edge of the control and the left edge of the client area of its control container.

Property	Description
Location	Gets or sets a System.Drawing.Point structure that specifies the pixel coordinates of the upper-left-hand corner of the control within the client area of its parent control container.
Parent	Gets or sets the object reference to a System.Windows.Forms.Control instance that is the parent form or control container for the control.
Right	Gets or sets an ABL INTEGER value that specifies the pixel distance between the right edge of the control and the left edge of the client area of its control container.
Size	Gets or sets a System.Drawing.Size structure that specifies the width and height of the control or control container in pixels.
TabIndex	Gets or sets an ABL INTEGER value that specifies the zero(0)-based tab order of the control within its control container. TabIndex values do not have to be contiguous. .NET simply observes the INTEGER order of the specified values.
TabStop	Gets or sets an ABL LOGICAL value indicating if the control participates in the tab order. **Note:** This property is always TRUE for an instance of any System.Windows.Form class (including Progress.Windows.Form).
Text	Gets or sets an ABL CHARACTER value that represents a string value displayed for the control or control container, as defined for the particular control or control container.
Top	Gets or sets an ABL INTEGER value that specifies the pixel distance between the top edge of the control and the top edge of the client area of its control container.
Visible	Gets or sets an ABL LOGICAL value that indicates whether the control or control container is visible to the user. The control or control container Show( ) method also sets this property to TRUE.
Width	Gets or sets an ABL INTEGER value that specifies the pixel width of the control.

Common .NET public control events

The following table shows some of the more common .NET public control events.

Table 8: Common .NET public control events

Event	Description
Click	Raised when the control or control container is clicked. **Note:** This is a higher-level event than a MouseClick event and is raised for other actions, such as pressing the **Enter** key when the control has focus.

Event	Description
DoubleClick	Raised when the control or control container is double-clicked.
EnabledChanged	Raised when the Enabled property value of the control or control container has changed (see Table 7: Common .NET public control properties on page 96).
Enter	Raised when the control or control container receives focus from any of several different actions. **Note:** This is the first of several events raised in a particular order, depending on how focus is change to the control.
Move	Raised when the control or control container is moved.
Paint	Raised when the control or control container is redrawn. This occurs for a number of different actions, such as resizing and moving other controls over this control. This event is especially useful for maintaining graphics that you draw for a control as its position and screen environment changes. **Note:** Use cautiously, and only when necessary, as it can impact performance.
TextChanged	Raised when the Text property of the control or control container changes value.
VisibleChanged	Raised when the Visible property of the control or control container changes value.

ABL support for managing .NET forms and controls

The general algorithm for creating and managing .NET forms is conceptually similar to creating and managing ABL windows. However, instead of creating widget objects and working with their handle attributes, methods, and events, you create .NET form and control objects and work with their object properties, methods, and events.

To block on all .NET non-modal forms in your application, you use a single occurrence of the WAIT-FOR statement (the .NET WAIT-FOR statement) that calls the static .NET input-blocking method, System.Windows.Forms.Application:Run( ). This single WAIT-FOR statement allows all non-modal .NET forms and ABL windows to be displayed and processes events for all non-modal ABL and .NET components in your application. However, to block on .NET modal forms, you use a separate .NET WAIT-FOR statement for each .NET modal form (dialog box), similar to displaying an ABL modal dialog box. This version of the .NET WAIT-FOR statement (for modal forms) calls a different input-blocking method (ShowDialog( )) on a given instance of the modal form. As noted previously (see the Limitations of support for .NET classes on page 31), you cannot call .NET input-blocking methods directly in an ABL application. Instead, you must use the .NET variation of the ABL WAIT-FOR statement to make these calls. This WAIT-FOR statement then blocks until the .NET input-blocking method returns.

Note: You cannot execute the .NET WAIT-FOR statement in a **non-GUI** ABL session, such as a character-mode client or AppServer agent.

.NET already provides the `System.Windows.Forms.Form` class to create forms. However, if you use the OpenEdge derived class (recommended), Progress.Windows.Form, to create all your .NET forms, ABL maintains the resulting instances on a single session form chain that you can access using SESSION system handle attributes. (This section describes how to access .NET forms on the session form chain.) You can then work with .NET forms and ABL windows in the same ABL session in a consistent fashion. For example, you can parent .NET forms and ABL windows to each other, creating window families that consist of both .NET forms and ABL windows. For more information on using .NET forms and windows together, see Using .NET Forms with ABL Windows on page 183

Finally, .NET allows you to create an XML resource (`.resx`) file associated with a given ABL class that stores resource sets (such as images) that some form controls use. To access this resource file from ABL, OpenEdge provides a .NET utility class, Progress.Util.ResourceHelper. This section describes how to use this class.

The following subsections describe:

- Initializing and blocking on .NET forms on page 99
- Blocking on non-modal forms on page 100
- Blocking on modal dialog boxes on page 103
- Accessing .NET forms using the SESSION system handle on page 107
- Accessing resource files for .NET forms on page 109

Initializing and blocking on .NET forms

There is a general pattern that applications follow to create and manage a .NET form.

To create and manage a .NET form:

1. Instantiate the form and its controls using the NEW function (classes).
2. Initialize the form, together with its controls and any additional control containers.
3. Subscribe handlers to appropriate .NET events, depending on your form and application.
4. Execute a .NET WAIT-FOR statement, which calls an appropriate .NET input-blocking method for the form.
5. After the form closes and the `WAIT-FOR` statement returns, do any post-form-closing tasks, such as resource clean-up, that your application might require.

Note: For multiple non-modal .NET forms, steps 1 through 3 can occur after step 4, in other words, in an event handler or trigger within the context of a single .NET `WAIT-FOR` statement.

For a given form, you can encapsulate most of these steps in a user-defined ABL form class that extends `Progress.Windows.Form`. For simplicity, this section primarily describes how to prepare and use the .NET `WAIT-FOR` statement for blocking on different types of .NET forms created directly from the `Progress.Windows.Form` class. For information on defining and using ABL-derived form classes to implement the basic types of .NET forms (non-modal, modal standalone, and non-modal MDI), see Creating custom .NET forms and controls on page 110. The basic principles for working with pure .NET form objects and ABL-derived .NET form objects are essentially the same.

Preparing to block on .NET forms

Depending on the application and type of form you might also have to explicitly initialize the form for display by setting its `Visible` property to `TRUE` or by calling its `Show( )` method. In any case, forms do not initially appear until you execute an appropriate .NET `WAIT-FOR` statement. After that point, you can display any other non-modal forms using their `Visible` properties or `Show( )` methods alone.

Blocking on and processing events for .NET forms

ABL supports the following general .NET `WAIT-FOR` statement syntax to block for input on .NET forms (see also the .NET WAIT-FOR statement syntax in OpenEdge Development: ABL Reference):

Syntax

```
WAIT-FOR dotNET-input-blocking-method-call .
```

For **all** non-modal .NET forms (and **all** non-modal ABL windows and other features that generate events, such as sockets), you use a single instance of this statement that calls the static `System.Windows.Forms.Application:Run( )` method. This statement blocks on one or more forms and processes all their events, as well as events for all non-modal ABL components. The difference between this .NET `WAIT-FOR` statement and a `WAIT-FOR` statement that processes only ABL events is that the .NET `WAIT-FOR` statement unblocks and terminates execution after the `Application:Run( )` method returns. This method termination automatically closes all instantiated .NET forms. However, the ABL-only `WAIT-FOR` statement for ABL events terminates only when a specified ABL event is raised, and you must manually close all non-modal ABL windows that you associate with this statement (see the reference entry for the ABL-only WAIT-FOR statement in OpenEdge Development: ABL Reference).

Note: You can also use the PROCESS EVENTS statement in an event handler or trigger to process all pending .NET and ABL events.

For **each** modal form, you use a single instance of the .NET `WAIT-FOR` statement that calls the `ShowDialog( )` method on the associated form object. This statement displays the form as a dialog box and processes events only for that form until the form is closed, much like the ABL-only `WAIT-FOR` statement that displays and processes events for an ABL dialog box.

For more information on using the non-modal .NET `WAIT-FOR` statement, see Blocking on non-modal forms on page 100. For more information on using the modal .NET `WAIT-FOR` statement, see Blocking on modal dialog boxes on page 103.

Blocking on non-modal forms

You can create non-modal forms from one or more instances of `System.Windows.Forms.Form` or any derived class. OpenEdge provides the .NET derived class, Progress.Windows.Form, which is specifically designed to work within an ABL session, allowing you either to create non-modal forms directly from this class or to create non-modal forms using ABL classes that you derive from it. To block on one or more non-modal forms, you call one of two overloadings of the static `Run( )` method on the .NET `System.Windows.Forms.Application` class. You can use this method to block on any combination of non-modal .NET forms and ABL windows, and to process all non-GUI ABL events in a session.

This is the syntax for calling the `Application:Run( )` method in a .NET WAIT-FOR statement:

Syntax

```
WAIT-FOR System.Windows.Forms.Application:Run( [ form-object-ref ] ) .
```

The *form-object-ref* is the object reference to one of several non-modal forms that you want to serve as the main form for your application. You can therefore block on non-modal forms by specifying a main form or block without specifying a main form. In both cases, this statement displays and blocks for input on all non-modal forms that you have initialized for display using their respective `Show( )` methods or `Visible` properties (see Preparing to block on .NET forms on page 100).

If you specify a main form using *form-object-ref*, you do not have to set this form to be visible prior to executing the `WAIT-FOR` statement. In this case, the `Application:Run(`*form-object-ref*`)` method automatically displays the form specified by *form-object-ref* (sets its `Visible` property to `TRUE`). In addition, the `Application:Run( )` method returns when this main form is closed by the user. Otherwise, you can programmatically cause the `Run( )` method to return by call the `Close( )` method on *form-object-ref*. Calling `Close( )` also automatically calls the `Dispose( )` method, which allows the form object and all the .NET controls that it contains to be to be garbage collected. For more information on the `Close( )` and `Dispose( )` methods, see Common .NET public control methods on page 94.

Caution: If you block using a main form (*form-object-ref*), and you have a trigger or event handler containing a loop that processes an ABL `READKEY` statement, if the user clicks the Close (**x**) button on the main form during execution of this loop, the session will shut down unconditionally. Therefore, avoid using `READKEY` statements in any applications that access .NET forms.

For example, this code fragment initializes four form objects and displays them:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE rForm1 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm2 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm3 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm4 AS CLASS Progress.Windows.Form NO-UNDO.

rForm1 = NEW Progress.Windows.Form( ).
rForm2 = NEW Progress.Windows.Form( ).
rForm3 = NEW Progress.Windows.Form( ).
rForm4 = NEW Progress.Windows.Form( ).

rForm1:Text = "Form1".
rForm1:Size = NEW System.Drawing.Size(300, 300).
rForm2:Text = "Form2".
rForm2:Size = NEW System.Drawing.Size(300, 300).
rForm3:Text = "Form3".
rForm3:Size = NEW System.Drawing.Size(300, 300).
rForm4:Text = "Form4".
rForm4:Size = NEW System.Drawing.Size (300, 300).

rForm2:Show( ).
rForm3:Show( ).

WAIT-FOR Application:Run( rForm1 ).

MESSAGE "End of Application" VIEW-AS ALERT-BOX INFORMATION.
```

When the WAIT-FOR statement executes, it displays the forms referenced by `rForm1`, `rForm2`, and `rForm3`, but not the form referenced by `rForm4` because this form has not been made visible.

When you close the main form referenced by rForm1, all other forms displayed for the blocking WAIT-FOR statement also close. In this example, when you click the Close (**x**) button on the form displayed for rForm1, this calls the Close( ) method on rForm1, and also calls the Close( ) method on rForm2 and rForm3. This is indicated by the fact that the MESSAGE statement displays its message box after all the forms have closed. You can also call the Close( ) method on rForm1 from an event handler to accomplish the same result.

The following example extends the previous example by allowing you to display another form, rForm4, using an event handler for the Click event on rForm1:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE rForm1 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm2 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm3 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm4 AS CLASS Progress.Windows.Form NO-UNDO.

rForm1 = NEW Progress.Windows.Form( ).
rForm2 = NEW Progress.Windows.Form( ).
rForm3 = NEW Progress.Windows.Form( ).

rForm1:Text = "Form1".
rForm1:Size = NEW System.Drawing.Size(300, 300).
rForm2:Text = "Form2".
rForm2:Size = NEW System.Drawing.Size(300, 300).
rForm3:Text = "Form3".
rForm3:Size = NEW System.Drawing.Size(300, 300).

rForm1:Click:Subscribe( "Form1_Click" ).

rForm2:Show( ).
rForm3:Show( ).

WAIT-FOR Application:Run( rForm1 ).

MESSAGE "End of Application" VIEW-AS ALERT-BOX INFORMATION.

PROCEDURE Form1_Click:

  DEFINE INPUT PARAMETER sender AS CLASS System.Object NO-UNDO.
  DEFINE INPUT PARAMETER e AS CLASS System.EventArgs NO-UNDO.

  rForm4 = NEW Progress.Windows.Form( ).
  rForm4:Text = "Form4".
  rForm4:Size = NEW System.Drawing.Size (300, 300).
  rForm4:Show( ).

END PROCEDURE.
```

Thus, when you click anywhere in the client area of the form for rForm1 after it displays, rForm4 displays also.

Finally, when you call the Application:Run( ) method without specifying a main form, the WAIT-FOR statement also displays and blocks on all properly initialized non-modal forms. However without a main form, the only way to return from the Application:Run( ) method is to call the Application:Exit( ) method from some event handler or trigger. Note that if you close all the displayed forms, you no longer have a UI affordance with which to terminate the application.

For example, in a variation of the previous example, a handler for the `Click` event on `rForm1` forces a return from the `Application:Run( )` method, which also closes any other displayed non-modal forms:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE rForm1 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm2 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm3 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm4 AS CLASS Progress.Windows.Form NO-UNDO.

rForm1 = NEW Progress.Windows.Form( ).
rForm2 = NEW Progress.Windows.Form( ).
rForm3 = NEW Progress.Windows.Form( ).
rForm4 = NEW Progress.Windows.Form( ).

rForm1:Text = "Form1".
rForm1:Size = NEW System.Drawing.Size(300, 300).
rForm2:Text = "Form2".
rForm2:Size = NEW System.Drawing.Size(300, 300).
rForm3:Text = "Form3".
rForm3:Size = NEW System.Drawing.Size(300, 300).
rForm4:Text = "Form4".
rForm4:Size = NEW System.Drawing.Size (300, 300).

rForm1:Click:Subscribe( "Form1_Click" ).
rForm1:Show( ).
rForm2:Show( ).
rForm3:Show( ).
rForm4:Show( ).

WAIT-FOR Application:Run( ).

MESSAGE "End of Application" VIEW-AS ALERT-BOX INFORMATION.

PROCEDURE Form1_Click:

  DEFINE INPUT PARAMETER sender AS CLASS System.Object NO-UNDO.
  DEFINE INPUT PARAMETER e AS CLASS System.EventArgs NO-UNDO.

  Application:Exit( ).

END PROCEDURE.
```

Note again in this example, if you close the form displayed for `rForm1` while the WAIT-FOR statement is blocking, there is no remaining UI affordance with which to exit the application (unless you had similar event handlers on `rForm2`, `rForm3`, and `rForm4`). The only way to terminate the application is to press `CTRL+BREAK`. So, if you call the `Application:Run( )` method without specifying a main form, you need to ensure that your application always calls `Application:Exit( )` based on an appropriate user action. Thus, if your application has a main form (for example, an MDI parent form), using the `Application:Run(` *`form-object-ref`* `)` method overloading to specify the main form makes multi-form management much easier.

Blocking on modal dialog boxes

When you display a form as a modal dialog box, the user cannot give focus to any other form in your application until they somehow close the dialog box. You can create modal forms from one of the following dialog box classes:

- **`System.Windows.Forms.Form` (or a derived class), especially `Progress.Windows.Form`)** — OpenEdge provides the derived .NET class, Progress.Windows.Form, which is specifically designed to work within an ABL session, allowing you either to create modal dialog boxes directly from it or to create

modal dialog boxes using ABL classes that you further derive from it. This class provides the same features for forms displayed as modal dialog boxes that are provided for forms displayed as non-modal forms. In addition, it provides features especially for use by dialog box forms.

- **.NET forms derived from the abstract class, `System.Windows.Forms.CommonDialog` —** `System.Windows.Forms.CommonDialog` is the base class for a set of special-purpose dialog boxes provided by the .NET Framework, including among others, the `System.Windows.Forms.FileDialog` class to implement a file selection dialog box, and the `System.Windows.Forms.ColorDialog` class to implement a color selection dialog box. These classes have features oriented around a specialized use as compared to the richer set of general-purpose features available with the `System.Windows.Forms.Form` class and its subclasses.

To display and block on a modal dialog box, execute a .NET `WAIT-FOR` statement that calls the `ShowDialog( )` method on the dialog box class instance. This is the syntax for using the .NET WAIT-FOR statement to call the `ShowDialog( )` method:

Syntax

```
WAIT-FOR dialog-object-ref:ShowDialog( [owning-form-ref] )
  [ SET return-value] .
```

The *dialog-object-ref* is an object reference to the form class instance you want displayed as a modal dialog box. The *owning-form-ref* is an object reference to a form that owns the dialog box (that is, a form over which the dialog box displays). Note that you cannot set the *dialog-object-ref* to be visible by calling its `Show( )` method prior to executing this statement, as you might with a non-modal form. If you do, .NET raises an error when you execute this `WAIT-FOR` statement. The `ShowDialog( )` method available on any *dialog-object-ref* returns a value of type `System.Windows.Forms.DialogResult`. This is an enumeration type that indicates the result from closing the dialog box. If you want to check this result, you can use the `SET` option, which sets a variable (*return-value*) to the value returned by the method. On some dialog form classes, you can also check the `DialogResult` property (with the same name as its type), which contains the same value.

For `Progress.Windows.Form` objects, you can set the `DialogResult` property in an event handler or rely on .NET to set the value from one or more button controls that you add to the dialog box. Every button control also has a `DialogResult` property, and .NET automatically sets the dialog box property to the value of the `DialogResult` property of the button that the user clicks in the dialog box. Either way, setting the value of `DialogResult` on the dialog box form object causes .NET to close the dialog box. You can then check the property value after the dialog box closes following termination of the `WAIT-FOR` statement.

Caution: Unlike for non-modal forms, when the user clicks the Close (**x**) button on a dialog box, or when you set the value of the *dialog-object-ref*`:DialogResult` property, the .NET Framework does not automatically call the `Close( )` method on *dialog-object-ref*, and therefore does not also call the `Dispose( )` method on the form. Instead, .NET hides the form so it can be shown again without having to create a new instance of the dialog box. Thus, when a modal form is no longer needed by your application, you must explicitly call the `Dispose( )` method on *dialog-object-ref* (and call it before **any** invocation of the DELETE OBJECT statement on *dialog-object-ref*) to ensure that the form and all the .NET controls that it contains are garbage collected. Otherwise, your .NET dialog boxes can create memory leaks in your application.

If you use a `Progress.Windows.Form` (or another `System.Windows.Forms.Form` class) instance to create a dialog box, you have access to the full range of methods, properties, and events that are available for this class, including the `DialogResult` property. This property is not available on `System.Windows.Forms.CommonDialog` objects. However, you can check the `DialogResult` value for dialog boxes displayed for a `CommonDialog` form by using the `SET` option.

Note: Unlike the single `WAIT-FOR` statement for displaying all non-modal .NET forms and ABL windows, you can execute a `WAIT-FOR` statement to display a dialog box (.NET or ABL) anywhere in your application, and in any order that you choose. For more information, see Using .NET Forms with ABL Windows on page 183.

The following `MixedForms.p` procedure defines two non-modal forms (`rForm1` and `rForm2`) and one modal dialog box (`rDialog1`) that displays when you click on either one of the forms. For information on locating and running this sample, see Example procedures.

```
USING System.Windows.Forms.* FROM ASSEMBLY.
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE rForm1 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rForm2 AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rLabel1 AS CLASS Label NO-UNDO.
DEFINE VARIABLE rLabel2 AS CLASS Label NO-UNDO.

rForm1 = NEW Progress.Windows.Form( ).
rForm2 = NEW Progress.Windows.Form( ).
rLabel1 = NEW Label( ).
rLabel2 = NEW Label( ).

rLabel1:Text = "Click in form to display dialog box...".
rLabel1:Size = TextRenderer:MeasureText( INPUT rLabel1:Text,
                                         INPUT rLabel1:Font ).
rLabel1:Width = rLabel1:Width * 1.1.
rLabel1:Top = 12.
rLabel2:Text = rLabel1:Text.
rLabel2:Size = rLabel1:Size.
rLabel2:Width = rLabel1:Width.
rLabel2:Top = rLabel1:Top.

rForm1:Text = "Form1".
rForm1:Size = NEW System.Drawing.Size( INPUT 300, INPUT 300 ).
rForm1:Controls:Add( INPUT rLabel1 ).
rLabel1:Left = INTEGER((rForm1:ClientSize:Width - rLabel1:Width) / 2).

rForm2:Text = "Form2".
rForm2:Size = rForm1:Size.
rForm2:Controls:Add( INPUT rLabel2 ).
rLabel2:Left = rLabel1:Left.

rForm1:Click:Subscribe("Form_Click").
rForm2:Click:Subscribe("Form_Click").

rForm2:Show( ).

WAIT-FOR Application:Run( rForm1 ).

MESSAGE "End of Application" VIEW-AS ALERT-BOX INFORMATION.

PROCEDURE Form_Click:

  DEFINE INPUT PARAMETER sender AS CLASS System.Object NO-UNDO.
  DEFINE INPUT PARAMETER e AS CLASS System.EventArgs NO-UNDO.

  DEFINE VARIABLE rDialog1 AS CLASS Progress.Windows.Form NO-UNDO.
  DEFINE VARIABLE rCloseButton AS CLASS Button NO-UNDO.
```

```
  DEFINE VARIABLE rCancelButton AS CLASS Button NO-UNDO.
  DEFINE VARIABLE enDialogResult AS CLASS DialogResult NO-UNDO.

  rDialog1 = NEW Progress.Windows.Form( ).
  rCloseButton = NEW Button( ).
  rCancelButton = NEW Button( ).

  rCloseButton:Text = "Close Form1".
  rCloseButton:Size = TextRenderer:MeasureText
                      ( INPUT "Close Form1", INPUT rCloseButton:Font ).
  rCloseButton:Width = INTEGER(rCloseButton:Width * 1.5).
  rCloseButton:Height = INTEGER(rCloseButton:Height * 1.5).
  rCloseButton:DialogResult = DialogResult:OK.

  rCancelButton:Text = "Cancel".
  rCancelButton:Size = TextRenderer:MeasureText
                       ( INPUT "Cancel", INPUT rCancelButton:Font ).
  rCancelButton:Width = INTEGER(rCancelButton:Width * 1.5).
  rCancelButton:Height = INTEGER(rCancelButton:Height * 1.5).
  rCancelButton:DialogResult = DialogResult:Cancel.

  rDialog1:Text = "Dialog1".
  rDialog1:FormBorderStyle = FormBorderStyle:FixedDialog.
  rDialog1:ClientSize = NEW System.Drawing.Size
    ( INPUT (rCloseButton:Width + rCancelButton:Width) * 2,
      INPUT (rCloseButton:Height + rCancelButton:Height) * 2 ).
  rDialog1:Controls:Add( INPUT rCloseButton ).
  rDialog1:Controls:Add( INPUT rCancelButton ).

  rCloseButton:Location = NEW System.Drawing.Point
    ( INPUT INTEGER( ( rDialog1:ClientSize:Width
                       - (rCloseButton:Width + rCancelButton:Width + 4) )
                     / 2 ),
      INPUT INTEGER( (rDialog1:ClientSize:Height - rCloseButton:Height)
                     / 2 ) ).
  rCancelButton:Location = NEW System.Drawing.Point
    ( INPUT rCloseButton:Location:X + rCloseButton:Width + 4,
      INPUT rCloseButton:Location:Y ).

  WAIT-FOR rDialog1:ShowDialog( ) SET enDialogResult.
  IF EnumHelper:AreEqual( INPUT enDialogResult,
                          INPUT DialogResult:OK )
  THEN rForm1:Close( ).

  rDialog1:Dispose( ).

END PROCEDURE.
```

In `MixedForms.p`, the `Form_Click` event handler defines and manages the dialog box and its `System.Windows.Forms.Button` controls (`rCloseButton` and `rCancelButton`). The handler sets the `DialogResult` property of each button to a different `DialogResult` enumeration value. It also sizes the client area (`ClientSize` property) of the dialog box based on the sizes of these buttons. It then adds and centers the buttons in the dialog box client area based on these dimensions. Note also the use of the `FormBorderStyle` property to specify a fixed dialog style from the `System.Windows.Forms.FormBorderStyle` enumeration.

The `ShowDialog( )` method called in the WAIT-FOR statement then blocks and displays the dialog box. If the user clicks the `rCloseButton` control, the `rCancelButton` control, or the Close (**x**) button on the title bar, the dialog box closes with the `DialogResult` property on `rDialog1` set to the clicked button's `DialogResult` property value or to `DialogResult:Cancel` for the Close (**x**) button.

After ShowDialog( ) returns, the WAIT-FOR statement terminates and sets enDialogResult with the method return value (which is also the value of the DialogResult property). The event handler then checks the value of enDialogResult using the static AreEqual( ) method of the Progress.Util.EnumHelper class. If the value is DialogResult:OK, the handler calls the Close( ) method on Form1. This causes the Application:Run( ) method to return from the non-modal WAIT-FOR statement in the main block, terminating the procedure. If the value is other than DialogResult:OK, the event handler assumes that the dialog box was cancelled, and closes, allowing the non-modal forms to continue being displayed. Note also that the handler calls Dispose( ) on rDialog1 to ensure that the dialog box is garbage collected.

For the System.Windows.Forms.CommonDialog classes, you can use the ShowDialog( ) method in much the same way as for Progress.Windows.Form. However, each CommonDialog class also provides its own set of class members built on an inheritance hierarchy based on CommonDialog. These dialog boxes do not provide DialogResult and many other properties provided by System.Windows.Forms.Form, including a Controls property to add controls. Befitting their application orientation, these dialog boxes are standardized for special-purpose use and manage all of the controls that they use as a built-in feature of each CommonDialog class.

Because there is no DialogResult property on these dialog boxes, to test the result of a CommonDialog object, you must use one of the following mechanisms:

- Use the SET option on the WAIT-FOR statement, as noted previously, to return the DialogResult value from ShowDialog( ).
- Query a property on the given CommonDialog object that changes based on the result.
- Use an event handler on a suitable subclass event, such as the FileOk event on the OpenFileDialog subclass, to handle the results.

Accessing .NET forms using the SESSION system handle

For GUI clients, ABL maintains a .NET form chain on the SESSION system handle that is analogous to the ABL window chain. This form chain links all .NET forms that you create in a session that are based on the OpenEdge form class, Progress.Windows.Form. The form chain is anchored to the SESSION handle at each end using the FIRST-FORM and LAST-FORM attributes. You can then use the NextForm and PrevForm properties on the Progress.Windows.Form class to walk the form chain similar to how you use the NEXT-SIBLING and PREV-SIBLING attributes to walk the ABL window chain.

However, note that the FIRST-FORM and LAST-FORM attributes actually have the Progress.Windows.IForm interface type, which Progress.Windows.Form implements. This interface defines the NextForm and PrevForm properties used to walk this chain so you can more naturally use .NET forms and ABL windows together. Thus, you can use this rIForm interface reference in the following example to walk the form chain:

```
DEFINE VARIABLE rForm AS CLASS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE rIForm AS CLASS Progress.Windows.IForm NO-UNDO.
DEFINE VARIABLE rDelForm AS CLASS Progress.Windows.IForm NO-UNDO.

rForm = NEW Progress.Windows.Form( ).
rForm:Text = "Market Quote 1".
rForm = NEW Progress.Windows.Form( ).
rForm:Text = "Market Search".
rForm = NEW Progress.Windows.Form( ).
rForm:Text = "Market Quote 2".

rIForm = SESSION:FIRST-FORM.
DO WHILE VALID-OBJECT( rIForm ):
  rForm = CAST(rIForm, Progress.Windows.Form) NO-ERROR.
  IF rForm <> ?
  THEN
    IF rForm:Text BEGINS "Market Quote"
    THEN
      MESSAGE rForm:Text VIEW-AS ALERT-BOX INFORMATION.
  rIForm = rIForm:NextForm.
END.

MESSAGE "End of Form Chain" VIEW-AS ALERT-BOX INFORMATION.
```

The iterative DO block in this example walks the chain looking for instances of Progress.Windows.Form. If this procedure is part of an application that also uses ABL windows, there can be a different type of form object on this chain (Progress.Windows.FormProxy) that is used to access these ABL windows. Thus, for each valid rIForm reference in the chain, if it cannot be cast to a Progress.Windows.Form, the ABL built-in CAST function returns the Unknown value (?). Otherwise, the cast succeeds, allowing the MESSAGE statement to display the Text property of an appropriate form object. For more information on using .NET forms with ABL windows, see Using .NET Forms with ABL Windows on page 183

Note: This example tests the result of the CAST function instead of the TYPE-OF function, because it is more efficient when used with .NET objects.

Note also that .NET forms, like all classes, appear on the session object chain anchored by the FIRST-OBJECT and LAST-OBJECT attributes of the SESSION system handle. So, you can also navigate them together with other class-based objects in an ABL session.

Note: Because System.Windows.Forms.Form does not implement Progress.Windows.IForm, if you use System.Windows.Forms.Form to create .NET forms in an ABL session, the form objects appear **only** on the session object chain. You also cannot use these native Microsoft form objects as a parent or child of an ABL window. For more information, see Using .NET Forms with ABL Windows on page 183 for more information.

Accessing resource files for .NET forms

.NET supports a file type for storing resources, such as graphics and images, that a form uses. These *resource files* are XML files with the `.resx` filename extension. In .NET, you typically create one resource file for a given form class and give it a filename that is the same as the name of the given form class, and store it with the source file for the class. In ABL, you might give it the same filename as an ABL-derived .NET form class for which the resources are designed, and store it with the ABL class file. The Visual Designer in Progress Developer Studio for OpenEdge automatically creates this file for any ABL-derived form to which you add resources. For more information on creating such resource files yourself, see the MSDN documentation at the following location:

http://msdn.microsoft.com/en-us/library/ekyft91f(VS.85).aspx

In order to retrieve resources from a given resource file, OpenEdge provides the .NET Progress.Util.ResourceHelper class. This class provides a single Load( ) method that allows you to retrieve the complete set of resources from a resource file. To call the method, you specify two `INPUT` parameters as `CHARACTER` expressions:

- A pathname that includes the filename (and extension) of the resource file. This can be an absolute or relative path, depending on the second parameter.
- Either a non-empty comma-separated list of pathnames (typically PROPATH) or an empty string (`""`). If it is non-empty, the first parameter must specify a file relative to a path on the list. Otherwise, the first parameter must specify a file either as an absolute pathname or as a pathname relative to the current working directory.

The `Load( )` method returns an object reference to a `System.Resources.ResXResourceSet`. This is a .NET Framework object that stores the resources of a resource file as objects that you can access by name. This class provides a `GetObject( )` method that you can use to return a specific resource object given its name.

The following code fragment loads a resource file, `form1.resx`, from PROPATH into a resources object (`rResources`). It then returns an image object named `"open.Image"` from `rResources` using the `GetObject( )` method:

```
DEFINE VARIABLE rResources AS System.Resources.ResXResourceSet NO-UNDO.
DEFINE VARIABLE rImage AS System.Drawing.Image NO-UNDO.

rResources = Progress.Util.ResourceHelper:Load( INPUT "form1.resx",
                                                INPUT PROPATH ).
rImage = CAST( rResources:GetObject("open.Image"), System.Drawing.Image ).
```

For more information on the `System.Resources.ResXResourceSet` class, see the .NET Framework Class Library documentation referenced by Microsoft .NET UI Controls on page 202.

Creating custom .NET forms and controls

Depending on your application and development environment, it can be useful to extend and work with .NET forms and control containers using ABL-derived .NET classes. The Visual Designer in Progress Developer Studio for OpenEdge allows you to visually design .NET forms and control containers from a palette of .NET controls (see OpenEdge Installed .NET Controls on page 201). It then generates ABL code for these forms and control containers as ABL-derived .NET form and control container objects. For more information on the Visual Designer, see OpenEdge Getting Started: Introducing the Progress Developer Studio for OpenEdge Visual Designer.

You can also write your own ABL-derived .NET classes to extend .NET forms and control containers using a similar design pattern. In general, using this design pattern, you interact with inherited protected and public members and add additional public members to these extended forms and control containers, as necessary, to allow your application to interact with application-specific elements of these objects. Otherwise, you can interact with these ABL-derived objects exactly like their base .NET object types using the inherited public members. (For more information on defining and using ABL-derived .NET classes, see Defining ABL-extended .NET objects on page 70). Using the same basic design pattern, you can implement extended versions of the basic .NET control and control container types:

- **Non-modal forms** — Windows that, once displayed, the user can enter, leave, and close in any order and manner that the user chooses according to the application design.
- **Modal dialog boxes** — Windows that, once displayed, the user must enter and close before entering any other window in the application.
- **MDI forms** — A non-modal window that contains other (child) non-modal windows. The MDI parent form provides a client area in which the user can display and manage child non-modal windows according to the application design. The user thus views the entire MDI form as a kind of desktop within a desktop.
- **Control containers** — Any of several .NET classes that provide a standard means of containing and managing other .NET controls as a group. The key feature of these control containers is that they can be contained and managed by a .NET form as members of a control collection like any individual control, and they can typically contain other control containers as well as individual controls within their own control collection. OpenEdge provides enhanced support for a special class of control container, a *user control*, that you can define by using the **ABL User Control** option on the **New** > **File** menu in Progress Developer Studio for OpenEdge. This option creates a new ABL class that inherits the `Progress.Windows.UserControl` class (see OpenEdge .NET form and control objects on page 90). Once you define it, you can use the extended user control as a design element in the Visual Designer, like any .NET control, by adding it to the Toolbox and dragging it onto other forms and user controls that you design.
- **Inherited controls** — You can extend any .NET control class that is not defined as `sealed` (`FINAL`). For example, you might define a custom Microsoft `Label` control that conforms to certain style conventions or that contains standard initial text, or you might define a custom `TreeView` control that always supports an initial node arrangement that is standard for all `TreeView` instances in an application. OpenEdge provides enhanced support for extending .NET controls by using the **ABL Inherited Control** option on the **New** > **File** menu in Progress Developer Studio for OpenEdge. This option creates a new ABL class that inherits a .NET control class that you specify. Once you define it, you can use the inherited control as a design element in the Visual Designer, like any .NET control, by adding it to the Toolbox and dragging it onto the forms and user controls that you design.

Note: All forms that derive from `System.Windows.Forms.Form`, themselves, represent a special type of control container that can contain any other type of control or control container object except another form.

This design pattern for ABL-derived forms and control containers generally includes the following elements (listed in no particular order):

- A single default `PUBLIC` constructor, which calls a `PRIVATE InitializeComponent( )` method and subscribes to any events that are handled by the ABL class itself
- The `PRIVATE IntializeComponent( )` method, which creates and initializes the ABL-derived .NET container and all the controls and other control containers that it adds to its control collection
- Any event handlers for events that are handled internally by the ABL class
- A destructor to delete any non-class-based objects created by the ABL class, any class-based objects you do not want to leave for ABL and .NET garbage collection to clean up, and to otherwise clean up resources maintained by instances of the class

  Note: For resources that you create only for use within the scope of a particular method (or property accessor), you generally clean up these resources within the scope of that particular method (or property).

- Any additional `PUBLIC` ABL properties and methods necessary to make the extended .NET object work in your application

For forms, in addition to the previous listed elements:

- If you are implementing the main form for your application, you typically provide a `PUBLIC` ABL method to execute a .NET WAIT-FOR statement that specifies THIS-OBJECT as the main form.

  Note: For any number of non-modal .NET forms (including MDI forms) and ABL windows, you can only have one .NET `WAIT-FOR` statement active at a time in an ABL session. Therefore, if you do have a main form in your application, you typically use the `WAIT-FOR` statement that specifies a main form. If there is no main form, you have no need to encapsulate a call to the `WAIT-FOR` statement in a `PUBLIC` method, because you can execute the `WAIT-FOR` statement directly in the main line of your application without specifying a particular form as the main form.

- If you are implementing a modal dialog box that has an owning form, especially one that can be displayed from different owning forms, you might provide a `PUBLIC` ABL method to execute a .NET `WAIT-FOR` statement that calls the `ShowDialog( )` method on `THIS-OBJECT`, taking a reference to its owning form as an `INPUT` parameter.

  Caution: If you display a modal dialog box, you must call `Dispose( )` on the form after your application is done with it. For more information, see Blocking on modal dialog boxes on page 103. If the form contains any ABL-derived controls or control containers, such as user controls, those controls are also not garbage collected until you call `Dispose( )`, which enables them for garbage collection as long as there are no other references to them in the ABL session.

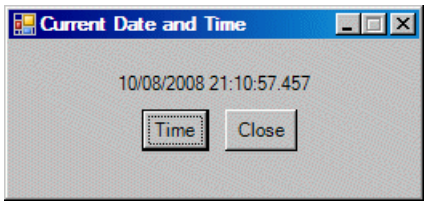
The following sections use these elements to implement sample ABL-derived versions of the basic types of .NET form and control container classes:

- Sample ABL-derived .NET non-modal form on page 112
- Sample ABL-derived .NET modal dialog box on page 115
- Sample ABL-derived .NET MDI form on page 120
- Sample ABL-derived .NET user control on page 126

Sample ABL-derived .NET non-modal form

`CurrentTimeForm` is a sample ABL-derived non-modal form class that displays as in the following figure.

Figure 5: Non-modal form displayed for CurrentTimeForm.cls



This form displays the current date and time every time you click the **Time** button and closes when you click the **Close** button. For information on locating and running this sample, see Example procedures.

Note: In Progress Developer Studio for OpenEdge, you can create ABL-derived non-modal forms to design in Visual Designer. You can initiate creation of a new non-modal form by clicking on **File** > **New** > **ABL Form**.

This is the initial section of code where the class private data and public members are defined.

```
USING Infragistics.Win.Misc.* FROM ASSEMBLY.
USING System.Windows.Forms.* FROM ASSEMBLY.
CLASS CurrentTimeForm INHERITS Progress.Windows.Form:

  /* Variables for controls on form */
  DEFINE PRIVATE VARIABLE rTimeButton AS CLASS UltraButton NO-UNDO.
  DEFINE PRIVATE VARIABLE rCloseButton AS CLASS UltraButton NO-UNDO.
  DEFINE PRIVATE VARIABLE rDateField AS CLASS UltraLabel NO-UNDO.
  DEFINE PRIVATE VARIABLE dtNow AS DATETIME NO-UNDO.

  /* Public properties and methods */
  METHOD PUBLIC VOID DoWait( ):
    WAIT-FOR Application:Run( THIS-OBJECT ).
  END METHOD.

  /* Constructor */
  CONSTRUCTOR PUBLIC CurrentTimeForm( ):
    InitializeComponent( ).
  END CONSTRUCTOR.
```

This section of code defines object references to the three controls from the Ultra Controls for .NET provided with OpenEdge (see OpenEdge Installed .NET Controls on page 201). The public members include the `DoWait( )` method, the `CurrentTimeForm` class constructor.

The `DoWait( )` method encapsulates execution of the WAIT-FOR statement to make `THIS-OBJECT` the main form of the application (see Blocking on non-modal forms on page 100). Therefore, if the application that invokes `DoWait( )` contains other .NET forms, they are all displayed and their events are processed in the context of this `WAIT-FOR` statement.

The constructor invokes the private `InitializeComponent( )` method to create and initialize the form and its controls.

This section of code is the beginning of the `InitializeComponent( )` method.

```
  /* Private form initialization method */
  METHOD PRIVATE VOID InitializeComponent( ):
```

```
/* Creates base form class and all components */
rTimeButton = NEW UltraButton( ).
rCloseButton = NEW UltraButton( ).
rDateField = NEW UltraLabel( ).

THIS-OBJECT:SuspendLayout( ).

/* Initialize current date/time field */
dtNow = NOW.    rDateField:Text = STRING(dtNow).
rDateField:Size = TextRenderer:MeasureText( rDateField:Text,
                                            rDateField:Font ).
rDateField:Top = 20.

/* Initialize the Time button */
rTimeButton:Text = "Time".
rTimeButton:Size = TextRenderer:MeasureText( rTimeButton:Text,
                                             rTimeButton:Font ).
rTimeButton:Height = rTimeButton:Height * 2.
rTimeButton:Width = rTimeButton:Width + 12.
rTimeButton:Top = rDateField:Top + rDateField:Height + 10.
rTimeButton:Click:Subscribe( TimeButton_Click ).

/* Initialize the Close button */
rCloseButton:Text = "Close".
rCloseButton:Size = TextRenderer:MeasureText( rCloseButton:Text,
                                              rCloseButton:Font ).
rCloseButton:Height = rTimeButton:Height.
rCloseButton:Width = rCloseButton:Width + 12.
rCloseButton:Top = rTimeButton:Top.
rCloseButton:Click:Subscribe( CloseButton_Click ).
```

It first instantiates, then initializes all the objects for the form. For example, it assigns an initial current date and time to the `Text` property of an `Infragistics.Win.Misc.UltraLabel` control. As part of the initialization, it also sizes the form controls based on their `Text` and `Font` settings, and sets initial positions for the controls on the form based on these sizes.

Note: The calculations for control sizes and positions in `CurrentTimeForm.cls` are convenient for hand coding. The Visual Designer generates hard values that result from the actions you take to design a form visually on the screen.

The initialization for each button also includes a subscription to a corresponding `PRIVATE` event handler that responds to the `Click` event on each button (the `TimeButton_Click( )` and `CloseButton_Click( )` methods, described in a following paragraph).

As a performance enhancement, the method also suspends the publishing of layout events as the form and its control objects are laid out. Applications do not typically handle layout events during form initialization, if at all. So, the `SuspendLayout( )` method is often called before initializing any layout logic, especially for forms and control containers that might have many controls laid out within them.

This section of code is the end of the `InitializeComponent( )` method.

```
    /* Initialize the form with its field and buttons */
    THIS-OBJECT:Text = "Current Date and Time".
    THIS-OBJECT:FormBorderStyle = FormBorderStyle:FixedSingle.
    THIS-OBJECT:MaximizeBox = FALSE.
    THIS-OBJECT:Controls:Add( rDateField ).
    THIS-OBJECT:Controls:Add( rTimeButton ).
    THIS-OBJECT:Controls:Add( rCloseButton ).
    THIS-OBJECT:AcceptButton = rTimeButton.

    /* Adjust form size and the location for field and button */
    THIS-OBJECT:Width = rDateField:Width * 2.
    THIS-OBJECT:Height = rCloseButton:Top + rCloseButton:Height
                         + rDateField:Height + 40.
    rDateField:Left = ( THIS-OBJECT:Width - rDateField:Width ) / 2.
    rTimeButton:Left = ( THIS-OBJECT:Width - ( rTimeButton:Width + 10
                                               + rCloseButton:Width ) ) / 2.
    rCloseButton:Left = rTimeButton:Left + rTimeButton:Width + 10.

    THIS-OBJECT:ResumeLayout( FALSE ).

  END METHOD.
```

This code initializes the form itself, adds the controls to its control collection, and adjusts the size of the form and the locations of its controls based on the control sizes. Setting the `AcceptButton` property to the `rTimeButton` object reference means that pressing the **ENTER** key clicks the **Time** button. Finally, the method resumes generation of layout events by calling `ResumeLayout( )` on the form, which allows the form layout to take effect.

Concluding the ABL class definition are the `PRIVATE` event handlers for the form.

```
  /* Private form event handlers */
  METHOD PRIVATE VOID TimeButton_Click( sender AS System.Object,
                                        e AS System.EventArgs ):
    dtNow = NOW.
    rDateField:Text = STRING(dtNow).
  END METHOD.

  METHOD PRIVATE VOID CloseButton_Click( sender AS System.Object,
                                         e AS System.EventArgs ):
    THIS-OBJECT:Close( ).
  END METHOD.

END CLASS.
```

The `TimeButton_Click( )` event handler method responds to a `Click` event on `rTimeButton` by updating the `Text` property on the `UltraLabel` control (`rDateField`) with the most recent date and time. The `CloseButton_Click( )` event handler responds to a `Click` event on `rCloseButton` by invoking the `Close( )` method of the form. This both closes the form and also closes the application because it blocks on `THIS-OBJECT` as the main form in the application using the `DoWait( )` method. (If the form was not displayed as a main form, the application would have to call `Application:Exit( )` to terminate the application.)

`CurrentTimeFromDriver.p` is the sample driver procedure (application) for the `CurrentTimeForm` sample class.

```
DEFINE VARIABLE rTimeForm AS CLASS CurrentTimeForm.

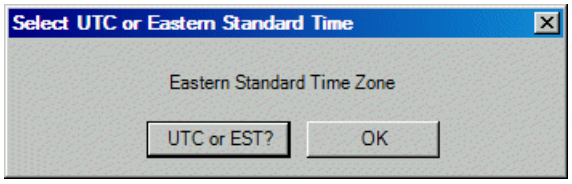
rTimeForm = NEW CurrentTimeForm().
rTimeForm:DoWait( ).
```

After instantiating the sample `CurrentTimeForm` class, the application simply invokes its public `DoWait( )` method to do all the work. While this is a simple example, it shows how an application can be simplified by encapsulating the processing for a .NET form within the ABL-derived form object itself.

Sample ABL-derived .NET modal dialog box

`UTCDialog` is a sample ABL-derived modal dialog box class that allows you to select a time zone used to display the current time, displayed as in the following figure.

Figure 6: Modal form (dialog box) displayed for UTCDialog.cls

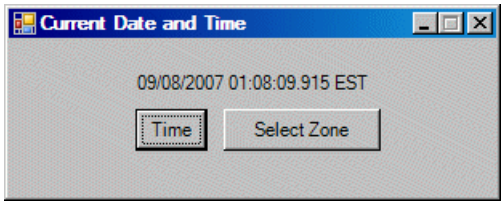


The dialog box displays two buttons, **UTC or EST?** and **OK**, and a label field displaying a message that indicates which time zone is selected, Eastern Standard Time (EST) or Coordinated Universal Time (UTC). When you click the **UTC or EST?** button, the label field toggles between one message and the other and sets a flag to indicate the selected time zone. When the you click **OK**, this closes the dialog box with a result that indicates the selected time zone.

Note: In Progress Developer Studio for OpenEdge, you can create ABL-derived modal dialog boxes to design in Visual Designer. You can initiate creation of a new modal dialog box by clicking on **File** > **New** > **ABL Dialog**.

The sample ABL-derived non-modal form class, `UTCSelectForm`, creates an instance of the `UTCDialog` class and displays its modal dialog box. This non-modal form class is similar to the `CurrentTimeForm` class described in Sample ABL-derived .NET non-modal form on page 112, and displays as in the following figure.

Figure 7: Non-modal form displayed for UTCSelectForm.cls



In this case, the **Close** button of `CurrentTimeForm` is replaced by a **Select Zone** button in `UTCSelectForm`. The non-modal form displays the current date and time with an indication of the time zone (UTC or EST). When you click the **Select Zone** button, this launches the `UTCDialog` instance where you can select the time zone for the current time. When you close the dialog box with a selection, the form now displays the time in the specified time zone, along with the appropriate time zone indication.

Note: Like `CurrentTimeForm`, `UTCSelectForm` makes its non-modal form the main form of the application. So, clicking the Close (**x**) button on the form implicitly invokes the `Close( )` method on the form, which both closes the main form and terminates the application.

You can launch these sample classes using the sample procedure, `UTCSelectFormDriver.p`., which is similar to the `CurrentTimeDriver.p` procedure described in the previous section. For information on locating and running these samples, see Example procedures.

The `UTCSelectForm` class launches its `UTCDialog` instance in an event handler for the **Select Zone** button (`TZButton_Click`). The following fragment of `UTCSelectForm` shows the `TZButton_Click` event handler and its supporting code.

```
USING Infragistics.Win.Misc.* FROM ASSEMBLY.
USING System.Windows.Forms.* FROM ASSEMBLY.
USING Progress.Util.* FROM ASSEMBLY.

CLASS UTCSelectForm INHERITS Progress.Windows.Form:

  /* Variables for controls on form */
  . . .
  DEFINE PRIVATE VARIABLE rTZButton AS CLASS UltraButton NO-UNDO.
  DEFINE PRIVATE VARIABLE rDateField AS CLASS UltraLabel NO-UNDO.
  DEFINE PRIVATE VARIABLE lUTCSelected AS LOGICAL INITIAL FALSE NO-UNDO.
  . . .

  METHOD PRIVATE VOID TZButton_Click( sender AS System.Object,
                                      e AS System.EventArgs ):
    DEFINE VARIABLE rDialog AS CLASS UTCDialog NO-UNDO.
    DEFINE VARIABLE enResult AS CLASS DialogResult NO-UNDO.
    rDialog = NEW UTCDialog( lUTCSelected ).
    rDialog:WaitForDialog( INPUT THIS-OBJECT, OUTPUT enResult ).

    rDialog:Dispose( ).

    IF EnumHelper:AreEqual( enResult, DialogResult:Yes )
    THEN
      lUTCSelected = TRUE.
    ELSE
      lUTCSelected = FALSE.

    rDateField:Text = GetTime( ).
    rDateField:Size = TextRenderer:MeasureText( rDateField:Text,
                                                rDateField:Font ).

  END METHOD.

  METHOD PRIVATE CHARACTER GetTime( ):
    DEFINE VARIABLE dTime AS DATETIME NO-UNDO.
    DEFINE VARIABLE iHour AS INTEGER INITIAL 3600000 NO-UNDO.

    dTime = NOW.
    IF lUTCSelected    THEN DO:
      dTime = dTime + (5 * iHour).
      RETURN STRING(dTime) + " UTC".
    END.
    ELSE DO:
      RETURN STRING(dTime) + " EST".
    END.
  END METHOD.
```

TZButton_Click creates the UTCDialog instance referenced by rDialog, and immediately executes the WaitForDialog( ) method on it. This method executes the WAIT-FOR statement to display and block on the dialog box, which is parented to the main form (THIS-OBJECT), passed in as an INPUT parameter. When the method returns, it also returns the enumeration value of the DialogResult property on the dialog box, passed out as an OUTPUT parameter (enResult). In the UTCDialog instance, the value of the DialogResult property is set in a Click event handler to indicate time zone selected in the dialog box.

Note: Instead of using the DialogResult property, the WaitForDialog( ) method could directly return a LOGICAL value for the lUTCSelected data member, which is also set in UTCDialog. However, this example also demonstrates another potential use for the DialogResult property in an application, especially where more complex results might be returned and you want the property to reflect a priority condition.

After the dialog box closes, this method returns, and the event handler tests enResult to determine what time zone has been selected. It then sets a UTCSelectForm class flag (lUTCSelected) accordingly and returns the current date and time, as a string, using the private GetTime( ) method. The GetTime( ) method also tests this flag and creates the date and time string from either the current date and time UTC or the current date and time EST (five hours earlier) and appends the time zone indicator before returning the result. The event handler concludes by setting the Text property of the UltraLabel instance (rDateField), which displays the date and time for the parent form.

Note: After WaitForDialog( ) returns, Dispose( ) is called on rDialog to ensure that the ABL-derived .NET dialog box object is garbage collected. For more information, see Blocking on modal dialog boxes.

This is the initial section of code for the ABL UTCDialog class, where the class private data and public members are defined.

```
USING System.Windows.Forms.* FROM ASSEMBLY.

 CLASS UTCDialog INHERITS Progress.Windows.Form:

  DEFINE PRIVATE VARIABLE buttonTZ AS Button NO-UNDO.
  DEFINE PRIVATE VARIABLE buttonOk AS Button NO-UNDO.
  DEFINE PRIVATE VARIABLE labelTZ AS Label NO-UNDO.
  DEFINE PRIVATE VARIABLE lUTCSelected AS LOGICAL NO-UNDO.

  METHOD PUBLIC VOID WaitForDialog
                      ( INPUT pForm AS CLASS Progress.Windows.Form,
                        OUTPUT pResult AS CLASS DialogResult ):
    WAIT-FOR THIS-OBJECT:ShowDialog( pForm ) SET pResult.
  END METHOD.

  CONSTRUCTOR PUBLIC UTCDialog ( INPUT pUTCSelected AS LOGICAL ):
    lUTCSelected = pUTCSelected.
    InitializeComponent( ).
  END CONSTRUCTOR.
```

The private data includes object references for the dialog box object and its control objects, and also includes its own class flag (lUTCSelected) to indicate the selected time zone. The public members include the WaitForDialog( ) method that displays and blocks on the dialog box and the UTCDialog class constructor.

The WaitForDialog( ) executes the WAIT-FOR statement calling the ShowDialog( ) instance method for the dialog box. This method takes the INPUT parameter (pForm) that references the parent form for the dialog box, which in this case will be the non-modal form instantiated by UTCSelectForm. It also returns the value of ShowDialog( ) (which is the DialogResult property value on rDialog) as an OUTPUT parameter (pResult) after the corresponding dialog box closes, thus allowing the caller to evaluate the dialog box results.

The constructor also takes a parameter (`pUTCSelected`) so that the caller can control the initial time zone selection in the dialog box. And as for any similar container class (see Sample ABL-derived .NET non-modal form on page 112), its constructor calls a private `InitializeComponent( )` method to initialize the dialog box and its controls.

This is the beginning of the `InitializeComponent( )` method for `UTCDialog`, showing the initialization of its controls.

```
METHOD PRIVATE VOID InitializeComponent ( ):

  buttonTZ = NEW Button ( ).
  buttonOk = NEW Button ( ).
  labelTZ = NEW Label ( ).

  THIS-OBJECT:SuspendLayout ( ).

  IF lUTCSelected
  THEN
    labelTZ:Text = "UTC Time Zone".
  ELSE
    labelTZ:Text = "Eastern Standard Time Zone".
  labelTZ:Size = TextRenderer:MeasureText( labelTZ:Text,
                                           labelTZ:Font ).

  labelTZ:Size = NEW System.Drawing.Size
                       ( INTEGER(labelTZ:Width * 1.1 ),
                         INTEGER(labelTZ:Height * 1.6 ) ).

  labelTZ:Top = 20.

  buttonTZ:Text = "UTC or EST?".
  buttonTZ:TabIndex = 0.
  buttonTZ:Size = NEW System.Drawing.Size
                        ( INTEGER(buttonTZ:Width * 1.2), 23 ).
  buttonTZ:Anchor = System.Windows.Forms.AnchorStyles:Bottom.
  buttonTZ:Top = 20 + labelTZ:Height + 10.
  buttonTZ:Click:Subscribe( tzButton_Click ).

  buttonOk:Text = "OK".
  buttonOk:TabIndex = 1.
  buttonOk:Size = NEW System.Drawing.Size
                        ( INTEGER(buttonOk:Width * 1.1), 23 ).
  buttonOk:Anchor = System.Windows.Forms.AnchorStyles:Bottom.
  buttonOk:Top = buttonTZ:Top.
  buttonOk:Click:Subscribe( okButton_Click ).
```

Notable initializations for this class include setting the initial label text (`labelTZ:Text`) to display an indication of currently selected time zone and subscribing event handlers to the `Click` events on the dialog box buttons that determine the latest time zone selection. Ultimately, the current time zone selection is indicated by the `lUTCSelected` flag. The value of this flag is initially set from the parameter passed to the `UTCDialog` class constructor and is reset according to the results of the dialog box.

This is the conclusion of the `InitializeComponent( )` method for `UTCDialog`, showing the dialog box initialization.

```
    THIS-OBJECT:Text = "Select UTC or Eastern Standard Time".
    THIS-OBJECT:AcceptButton = buttonOk.
    THIS-OBJECT:ClientSize = NEW System.Drawing.Size
                             ( (buttonTZ:Width + buttonOK:Width) * 2,
                               labelTZ:Height + buttonTZ:Height + 40 ).

    THIS-OBJECT:Controls:Add( labelTZ ).
    THIS-OBJECT:Controls:Add( buttonTZ ).
    THIS-OBJECT:Controls:Add( buttonOk ).
    labelTZ:Left = (THIS-OBJECT:Width - labelTZ:Width) / 2.
    buttonTZ:Left = ( THIS-OBJECT:Width -
                      ( buttonTZ:Width + buttonOk:Width + 10 ) ) / 2.
    buttonOk:Left = buttonTZ:Left + buttonTZ:Width + 10.
    THIS-OBJECT:FormBorderStyle = FormBorderStyle:FixedDialog.
    THIS-OBJECT:MaximizeBox = FALSE.
    THIS-OBJECT:MinimizeBox = FALSE.
    THIS-OBJECT:ShowInTaskbar = FALSE.
    THIS-OBJECT:StartPosition = FormStartPosition:CenterParent.

    THIS-OBJECT:ResumeLayout (FALSE).
  END METHOD.
```

The last several settings for the dialog box object (`THIS-OBJECT`) initialization show some typical properties set for a dialog box. Note especially the `StartPosition` property set to `FormStartPosition:CenterParent`. This centers the dialog box over its parent form, which in this case will be the non-modal form created by `UTCSelectForm`, whose object reference (`pForm`) is passed to the `WaitForDialog( )` method.

This is the concluding section of `UTCDialog` code showing its `PRIVATE` event handlers.

```
  METHOD PRIVATE VOID okButton_Click (sender AS System.Object,
                                      e AS System.EventArgs):
    IF lUTCSelected
    THEN
      THIS-OBJECT:DialogResult = DialogResult:Yes.
    ELSE
      THIS-OBJECT:DialogResult = DialogResult:No.
    THIS-OBJECT:Close ( ).
  END METHOD.

  METHOD PRIVATE VOID tzButton_Click (sender AS System.Object,
                                      e AS System.EventArgs):
    IF lUTCSelected
    THEN DO:
      lUTCSelected = FALSE.
      labelTZ:Text = "Eastern Standard Time Zone".
    END.
    ELSE DO:
      lUTCSelected = TRUE.
      labelTZ:Text = "UTC Time Zone".
    END.
    labelTZ:Size = TextRenderer:MeasureText( labelTZ:Text,
                                             labelTZ:Font ).
    labelTZ:Size = NEW System.Drawing.Size
                         ( INTEGER(labelTZ:Width * 1.1 ),
                           INTEGER(labelTZ:Height * 1.6 ) ).
    labelTZ:Left = (THIS-OBJECT:Width - labelTZ:Width) / 2.
  END METHOD.

END CLASS.
```

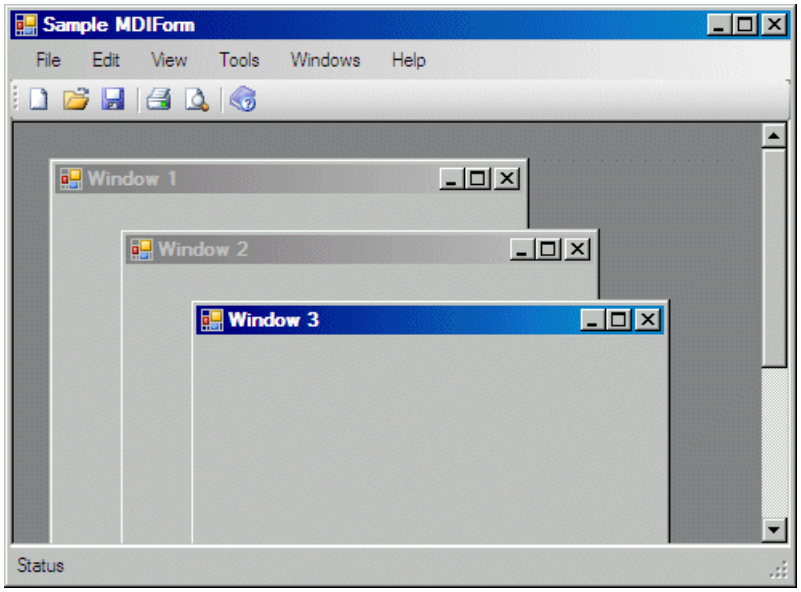
In the `okButton_Click` event handler, when the **OK** button (`buttonOK`) is clicked, it sets the `DialogResult` property of the dialog box (`THIS-OBJECT`) depending on the current value of the class private flag (`lUTCSelected`), which setting closes the dialog box.

In the `tzButton_Click` event handler, when the **UTC or EST?** button (`buttonTZ`) is clicked, it toggles the current time zone setting by reversing the value of the `lUTCSelected` flag and setting the `Text` property of the `Label` control (`labelTZ`) to indicate the newly selected time zone. Note that the label is resized and re-centered in the dialog box according to its current `Text` property value and font.

Sample ABL-derived .NET MDI form

`MDIForm` is a sample ABL-derived non-modal MDI form class. This form displays an MDI window as in the following figure, with standard menus, a toolbar, and status bar.

Figure 8: MDI form displayed for MDIForm.cls



Some of the menu functions have associated behavior. For example, when you click the **File** > **New** menu item (or its toolbar button) this opens a child non-modal form in the client area of the parent MDI form, and you can open as many child forms this way as you want. You can then manipulate the child forms within the MDI client area in the typical manner of a multiple document interface. When you click the **File** > **Open** menu item (or its toolbar button), this opens the standard .NET open file dialog box. The sample implementation allows you to select a file and click the **Open** button, which closes the dialog box, but does not actually open the specified file. Other menu functions (not documented here) have no behavior or varying types of sample behavior similar to the **New** and **Open** functions.

Note: In Progress Developer Studio for OpenEdge, you can create ABL-derived non-modal MDI forms to design in Visual Designer. You can initiate creation of a new non-modal MDI form by clicking on **File** > **New** > **ABL MDI Form**.

The `MDIForm.cls` file also has an `MDIForm.resx` resource file associated with it, which provides the icon images used in the form. For more information on accessing resource files, see Accessing resource files for .NET forms on page 109.

You can launch this sample MDI form using the sample procedure, `MDIFormDriver.p`. For information on locating and running these samples, see Example procedures.

The MDIForm class contains a relatively large amount of sample code. So, the following description focuses on the code for the previously described **New** and **Open** menu functions. In general, the basic initialization of a standard MDI form includes a hierarchy of control containers and controls, starting with the MDI parent form, and continuing with a container for the menus, a separate container for the menu items of each menu, and a container for the buttons in each toolbar.

Thus, this description provides an overview of the entire MDIForm class, with a focus on the form menu container (a System.Windows.Forms.MenuStrip), the **File** menu container (a System.Windows.Forms.ToolStripMenuItem), and the **New** and **Open** menu items (also a System.Windows.Forms.ToolStripMenuItem). So, each menu and its menu items represents a hierarchy of System.Windows.Forms.ToolStripMenuItem instances. (The related toolbar is contained by a System.Windows.Forms.ToolStrip not described, here.)

This is the initial section of code for the ABL MDIForm class, where the class private data is defined.

```
USING System.* FROM ASSEMBLY.
USING System.ComponentModel.* FROM ASSEMBLY.
USING System.Windows.Forms.* FROM ASSEMBLY.
USING Progress.Windows.* FROM ASSEMBLY.
USING Progress.Util.* FROM ASSEMBLY.

CLASS MDIForm INHERITS Progress.Windows.Form:

  DEFINE PRIVATE VARIABLE components AS System.ComponentModel.IContainer
    NO-UNDO.
  DEFINE PRIVATE VARIABLE resources AS System.Resources.ResXResourceSet
    NO-UNDO.
  DEFINE PRIVATE VARIABLE childFormNumber AS INTEGER INITIAL 0 NO-UNDO.
  DEFINE PRIVATE VARIABLE openFileDialog AS
                          System.Windows.Forms.OpenFileDialog NO-UNDO.
  DEFINE PRIVATE VARIABLE saveFileDialog AS
                          System.Windows.Forms.SaveFileDialog NO-UNDO.
  DEFINE PRIVATE VARIABLE menuStrip AS System.Windows.Forms.MenuStrip
    NO-UNDO.
  DEFINE PRIVATE VARIABLE toolStrip AS System.Windows.Forms.ToolStrip
    NO-UNDO.
  DEFINE PRIVATE VARIABLE statusStrip AS System.Windows.Forms.StatusStrip
    NO-UNDO.
  DEFINE PRIVATE VARIABLE fileMenu AS
                          System.Windows.Forms.ToolStripMenuItem NO-UNDO.
. . .
  DEFINE PRIVATE VARIABLE printPreviewToolStripButton AS
                          System.Windows.Forms.ToolStripButton NO-UNDO.
  DEFINE PRIVATE VARIABLE helpToolStripButton AS
                          System.Windows.Forms.ToolStripButton NO-UNDO.
  DEFINE PRIVATE VARIABLE toolTip AS System.Windows.Forms.ToolTip NO-UNDO.
```

The private data includes definitions for all the object references used by the form. Note especially the resources object reference defined to reference the contents of the MDIForm.resx file, the openFileDialog object reference defined to reference the open file dialog box that is launched using the **Open** menu item, the menuStrip object reference defined to reference the menu container, and the fileMenu object reference defined to reference the **File** menu container. The childFormNumber is used to provide an incremental number displayed in the title bar of each child form opened using the **New** menu function.

After its private data, MDIForm defines its public members.

```
METHOD PUBLIC VOID DoWait ( ):
  WAIT-FOR System.Windows.Forms.Application:Run (THIS-OBJECT).
END METHOD.

CONSTRUCTOR PUBLIC MDIForm( ):
  InitializeComponent ( ).
END CONSTRUCTOR.
```

The public members include the DoWait( ) method to run the form's WAIT-FOR statement and the MDIForm class constructor. Note that in DoWait( ), THIS-OBJECT is passed as a parameter to the Applicaton:Run( ) method. This allows the MDI form to function as the main form of the application, allowing the MDI form and its child form objects to be managed more tightly as a unit, for example, so they are more easily closed together (shown further on in the sample code).

Also as is typical of ABL form container classes, the constructor calls an InitilizeComponent( ) method to create and initialize most of the objects used for the form. This includes all the controls and control containers for the form.

The following shows code sections from the beginning of the InitializeComponent( ) method. This starts by creating the objects for the MDI parent form, including the menu container (menuStrip), the **File** menu (fileMenu), and continues (not shown) for all the remaining control and control container objects used by the form. This code also loads the image resources from the MDIForm.resx file into a new resource object (resources). For performance reasons, the initialization code then suspends layout events for each of the control containers, including the MDI parent form, itself (THIS-OBJECT).

```
METHOD PRIVATE VOID InitializeComponent ( ):

  resources = ResourceHelper:Load("MDIForm.resx", PROPATH).
  components = NEW System.ComponentModel.Container( ).
  menuStrip = NEW System.Windows.Forms.MenuStrip( ).
  fileMenu = NEW System.Windows.Forms.ToolStripMenuItem( ).
  . . .
  menuStrip:SuspendLayout( ).
  toolStrip:SuspendLayout( ).
  statusStrip:SuspendLayout( ).
  THIS-OBJECT:SuspendLayout( ).

  /* menuStrip */
  DEFINE VARIABLE arrayvar1 as System.Array no-undo.
  arrayvar1 = System.Array:CreateInstance
    (TypeHelper:GetType("System.Windows.Forms.ToolStripItem"), 6).
  arrayvar1:SetValue(fileMenu, 0).
  arrayvar1:SetValue(editMenu, 1).
  arrayvar1:SetValue(viewMenu, 2).
  arrayvar1:SetValue(toolsMenu, 3).
  arrayvar1:SetValue(windowsMenu, 4).
  arrayvar1:SetValue(helpMenu, 5).
  menuStrip:Items:AddRange( CAST (arrayvar1,
                            "System.Windows.Forms.ToolStripItem[]") ).
  menuStrip:Location = NEW System.Drawing.Point(0, 0).
  menuStrip:MdiWindowListItem = windowsMenu.
  menuStrip:Name = "menuStrip".
  menuStrip:Size = NEW System.Drawing.Size(632, 24).
  menuStrip:TabIndex = 0.
  menuStrip:Text = "MenuStrip".

  /* fileMenu */
  DEFINE VARIABLE arrayvar2 as System.Array no-undo.
  arrayvar2 = System.Array:CreateInstance
```

```
  (TypeHelper:GetType("System.Windows.Forms.ToolStripItem"), 11).
arrayvar2:SetValue(newToolStripMenuItem, 0).
arrayvar2:SetValue(openToolStripMenuItem, 1).
. . .
arrayvar2:SetValue(exitToolStripMenuItem, 10).
fileMenu:DropDownItems:AddRange( CAST (arrayvar2,
                                "System.Windows.Forms.ToolStripItem[]") ).
fileMenu:ImageTransparentColor
  = System.Drawing.SystemColors:ActiveBorder.
fileMenu:Name = "fileMenu".
fileMenu:Size = NEW System.Drawing.Size(35, 20).
fileMenu:Text = "&File".
```

The first control container to be initialized is the menu container (`menuStrip`), a `MenuStrip` object. This follows a typical pattern for all control containers in the form. First, it creates an object array to hold instances of the control type used in the control container, in this case for six (6) instances of `System.Windows.Forms.ToolStripItem`., representing the six menus on the MDI form menu bar. It then loads the array with the previously created menu objects, starting with the **File** menu (`fileMenu`) and ending with the **Help** menu (`helpMenu`). It adds this array of `ToolStripItem` instances to the `ToolStripCollection` (`Items` property) of the `menuStrip` object. It then sets a number of other properties on the `menuStrip` object, including the `MdiWindowListItem` property, which is set to the object representing the **Windows** menu, which lists all of the child forms that are open in the MDI parent client area. Note that the array contents, as reference type objects, remain instantiated when `InitializeComponent( )` terminates because they are still referenced in ABL as well as by the `menuStrip` object to which they are added. The code initializes all of the remaining top-level control containers in the MDI form in a similar manner.

The control container for the **File** menu (`fileMenu`) is initialized in a similar manner. In this case, the menu is represented by a `ToolStripMenuItem` object, which is itself a container for other `ToolStripMenuItem` objects (**File** menu items). The code adds these objects to the corresponding object array, starting with the `newToolStripMenuItem` and `openToolStripMenuItem` objects, which represent the **New** and **Open** menu items. In this case, the object array is added to the `ToolStripCollection` of the `fileMenu` object specified by its `DropDownItems` property. Again, the code sets properties on the `fileMenu` control container. Note the setting of the `Text` property, which uses the `"&"` character to indicate the menu shortcut. The code initializes all of the remaining menus across the menu bar of the MDI form in a similar manner.

The next object to be initialized (shown in the following code section) is the `ToolStripMenuItem` object (`newToolStripMenuItem`) that implements the **New** item in the **File** menu.

```
    /* newToolStripMenuItem */
    newToolStripMenuItem:Image
      = CAST( resources:GetObject("newToolStripMenuItem.Image"),
              System.Drawing.Image ).
    newToolStripMenuItem:ImageTransparentColor
      = System.Drawing.Color:Black.
    newToolStripMenuItem:Name = "newToolStripMenuItem".
    newToolStripMenuItem:ShortcutKeys
      = CAST( EnumHelper:Or(System.Windows.Forms.Keys:Control,
                            System.Windows.Forms.Keys:N),
              System.Windows.Forms.Keys ).
    newToolStripMenuItem:Size = NEW System.Drawing.Size(151, 22).
    newToolStripMenuItem:Text = "&New".
    newToolStripMenuItem:Click:Subscribe(ShowNewForm).
    . . .
    openToolStripMenuItem:Click:Subscribe(OpenFile).
    . . .
    /* MDIForm */
    THIS-OBJECT:Text = "<Put title here>".
    THIS-OBJECT:AutoScaleDimensions = NEW System.Drawing.SizeF(6.0, 13.0).
    THIS-OBJECT:AutoScaleMode = System.Windows.Forms.AutoScaleMode:Font.
    THIS-OBJECT:ClientSize = NEW System.Drawing.Size(632, 453).
    THIS-OBJECT:Controls:Add(statusStrip).
    THIS-OBJECT:Controls:Add(toolStrip).
    THIS-OBJECT:Controls:Add(menuStrip).
    THIS-OBJECT:IsMdiContainer = TRUE.
    THIS-OBJECT:MainMenuStrip = menuStrip.
    THIS-OBJECT:Name = "MDIForm".
    menuStrip:ResumeLayout( FALSE ).
    menuStrip:PerformLayout( ).
    toolStrip:ResumeLayout( FALSE ).
    toolStrip:PerformLayout( ).
    statusStrip:ResumeLayout( FALSE ).
    statusStrip:PerformLayout( ).
    THIS-OBJECT:ResumeLayout( FALSE ).
    THIS-OBJECT:PerformLayout( ).

  END METHOD.

END CLASS.
```

Like several menu items in this MDI form, this one has an icon image that it gets from the `resources` object. This code also defines a control key sequence as a shortcut (`ShortcutKeys` property), as well as a menu shortcut (`Text` property setting). Finally, it subscribes an event handler (`ShowNewForm( )` method) to the `Click` event on the object. The code initializes most of the remaining menu items and toolbar buttons in a similar manner. For example further on, it subscribes the `OpenFile( )` event handler to the `Click` event on the **Open** menu item object (`openToolStripMenuItem`).

The initialization code terminates by initializing the MDI parent form, itself, and by resuming layout events and performing left-over layout tasks for all the control containers. Form initialization, itself, is a relatively simple matter of adding the control containers previously initialized to its own control collection, setting its `IsMdiContainer` property to `TRUE` (which turns on most of the MDI functionality for the form), and making the `menuStrip` object its primary menu container.

All of the operational behavior for the MDI parent form occurs in the event handlers subscribed to events of all the menu items and toolbar buttons. Following from the event subscriptions for the **New** and **Open** menu items, `MDIForm` provides `ShowNewForm( )` and `OpenFile( )` event handlers as shown in the following section of code. Additional event handlers also appear in the same code section, which implement basic behavior for the MDI parent form. All event handlers execute in response to events processed during execution of the MDI form class method, `DoWait( )`.

```
METHOD PRIVATE VOID ShowNewForm(sender AS Object, e AS EventArgs):
  /* Create a NEW instance of the child form. */
  DEFINE VARIABLE childForm AS Progress.Windows.Form NO-UNDO.
  childForm = NEW Progress.Windows.Form( ).
  /* Make it a child of this MDI form before showing it. */
  childForm:MdiParent = THIS-OBJECT.
  childFormNumber = childFormNumber + 1.
  childForm:Text = "Window " + STRING(childFormNumber).
  childForm:Show( ).
END METHOD.

METHOD PRIVATE VOID ExitToolsStripMenuItem_Click(sender AS Object,
                                                 e AS EventArgs):
  THIS-OBJECT:Close( ).
END METHOD.
METHOD PRIVATE VOID OpenFile(sender AS Object, e AS EventArgs):
  openFileDialog = NEW OpenFileDialog( ).
  openFileDialog:InitialDirectory
    = System.Environment:GetFolderPath
                        (System.Environment+SpecialFolder:Personal).
  openFileDialog:Filter = "Text Files (*.txt)|*.txt|All Files (*.*)|*.*".
  openFileDialog:FileOk:Subscribe(OpenFile_Ok)./* Open clicked */
  WAIT-FOR openFileDialog:ShowDialog( THIS-OBJECT ).
  openFileDialog:Dispose( ).
END METHOD.

METHOD PRIVATE VOID OpenFile_Ok(sender AS Object, e AS CancelEventArgs):
  DEFINE VARIABLE fileName AS CHARACTER NO-UNDO.
  fileName = openFileDialog:FileName.
  /* TODO: Add code here to open the file. */
END METHOD.
```

So if you click the **File** > **New** menu item, the `ShowNewForm( )` event handler immediately instantiates a non-modal form (`childForm`) and initializes the new form as a child of the MDI parent form by setting the `MdiParent` property of `childForm` to `THIS-OBJECT`. It also titles the form by incrementing and appending the value of `childFormNumber` to its `Text` property character string. It then calls the `Show( )` method to display the new child form in the MDI form client area. The event handler thus creates and displays a new child form for each click of the **New** menu item (or its toolbar button).

If you click the **File** > **Open** menu item, the `OpenFile( )` event handler instantiates the `OpenFileDialog` class (`openFileDialog`), sets some environment properties for opening a file, subscribes the `OpenFile_Ok( )` event handler to the object's `FileOk` event, and executes the `WAIT-FOR` statement to display and block for input on the dialog box. After selecting a file, clicking **Open** in the dialog box publishes the `FileOk` event. The `OpenFile_Ok( )` event handler executes to open the selected file (not implemented). When this event handler returns, the `WAIT-FOR` statement terminates in the `OpenFile( )` event handler.

Note: Clicking the `Cancel` button in the **Open** dialog box also causes the dialog box to close and the associated `WAIT-FOR` statement to terminate.

Sample ABL-derived .NET user control

`SampleUserControl` is a sample ABL-derived user control class based on Progress.Windows.UserControl. `Progress.Windows.UserControl` inherits from the .NET `System.Windows.Forms.UserControl` class and allows you to define a control container that functions as a user-defined .NET control (*user control*). In an ABL-derived user control, you can contain a set of other ABL-derived user controls, .NET controls, and control containers that you might want to function together as a single control for use in several different applications.

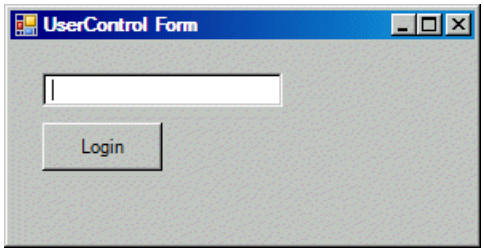
Note: In Progress Developer Studio for OpenEdge, you can create ABL-derived user controls for reuse as new design components in Visual Designer. You can initiate creation of a new user control by clicking on **File** > **New** > **ABL User Control**.

`SampleUserControl` initializes a user control much like an ABL-derived form might initialize itself (see the previous sample ABL-derived .NET classes, for example, as described in Sample ABL-derived .NET non-modal form on page 112). In this case, the user control contains two .NET controls:

- A password field implemented by an Advanced UI Control (`Infragistics.Win.UltraWinEditors.UltraTextEditor`)
- A **Login** button implemented by another Advanced UI Control (`Infragistics.Win.Misc.UltraButton`)

The sample ABL class, `UserControlForm`, displays this control container in a non-modal form, and the sample `UserControlFormDriver.p` procedure displays the form, as in the following figure.

Figure 9: User control displayed for UserControlForm.cls



In the user control, you can type a masked password string in a text box and you can click the **Login** button, but there is no other behavior implemented for this control. For information on locating and running these samples, see Example procedures.

Note that in this example, `UserControlForm` is the only form class where `SampleUserControl` is used. Typically, you design a single ABL-derived user control for use in multiple forms or applications. For example, a single login user control might be used in any number of application modules where a user might be required to enter security credentials.

This is the initial section of code for the ABL `SampleUserControl` class, where the class private data and public members are defined.

```
USING Infragistics.Win.UltraWinEditors.* FROM ASSEMBLY.
USING Infragistics.Win.Misc.* FROM ASSEMBLY.

CLASS SampleUserControl INHERITS Progress.Windows.UserControl:

  /* Class private data */
  DEFINE PRIVATE VARIABLE rPasswordEditor AS CLASS UltraTextEditor NO-UNDO.
  DEFINE PRIVATE VARIABLE rLoginButton AS CLASS UltraButton NO-UNDO.

  /* Class public members */
  DEFINE PUBLIC PROPERTY PasswordEntry AS CLASS UltraTextEditor NO-UNDO
    GET.
    PRIVATE SET.

  DEFINE PUBLIC PROPERTY LoginButton AS CLASS UltraButton NO-UNDO
    GET.
    PRIVATE SET.

  CONSTRUCTOR PUBLIC SampleUserControl( ):
    InitializeComponent( ).
  END CONSTRUCTOR.
```

The `rPasswordEditor` and `rLoginButton` variables define `PRIVATE` object references to the control objects contained by the user control. The read-only `PasswordEntry` and `LoginButton` properties provide `PUBLIC` object references to these control objects for use by the form that contains the user control. This sample does not use these properties except to assign them references to their associated control objects.

Note: Because ABL does not allow you to define events for ABL classes, you cannot define events for an ABL-derived .NET class, including a user control. A form class that contains a user control and wants to respond to events on the controls it contains has two options. It must either subscribe event handlers directly to events on the public controls contained by the user control or the user control, itself, can subscribe proxy event handlers to its contained control events that call the real event handlers defined in the form class. For more information on creating and using proxy event handlers, see Handling events on controls contained by ABL-derived .NET classes on page 82.

As is typical for ABL container classes, the constructor invokes an `InitializeComponent( )` method, which instantiates and initializes the user control.

The `InitializeComponent( )` method follows, which completes the `SampleUserControl` class.

```
  METHOD PRIVATE VOID InitializeComponent( ):

    /* Initializes extended user control container class with
       controls and initializes public properties to access them */
    rPasswordEditor = NEW UltraTextEditor( ).
    PasswordEntry = rPasswordEditor.
    rLoginButton = NEW UltraButton( ).
    LoginButton = rLoginButton.

    CAST(rPasswordEditor,
       System.ComponentModel.ISupportInitialize):BeginInit( ).
    THIS-OBJECT:SuspendLayout( ).

    /* Set properties */
    rPasswordEditor:Location = NEW System.Drawing.Point( 20, 0 ).
    rPasswordEditor:Size = NEW System.Drawing.Size( 150, 21 ).
    rPasswordEditor:TabIndex = 0.
```

```
    rPasswordEditor:Text = "".
    rPasswordEditor:PasswordChar = "*".

    rLoginButton:Text = "Login".
    rLoginButton:Location = NEW System.Drawing.Point( 20, 31 ).
    rLoginButton:Size = NEW System.Drawing.Size( 75, 30 ).
    rPasswordEditor:TabIndex = 1.

    THIS-OBJECT:AutoScaleDimensions = NEW System.Drawing.SizeF( 6, 13 ).
    THIS-OBJECT:AutoScaleMode = System.Windows.Forms.AutoScaleMode:Font.

    /* Add controls to form */
    THIS-OBJECT:Controls:Add( rPasswordEditor ).
    THIS-OBJECT:Controls:Add( rLoginButton ).
    THIS-OBJECT:ClientSize = NEW System.Drawing.Size( 200, 200 ).

    CAST(rPasswordEditor,
       System.ComponentModel.ISupportInitialize):EndInit( ).
    THIS-OBJECT:ResumeLayout( FALSE ).
    THIS-OBJECT:PerformLayout( ).

  END METHOD.

END CLASS.
```

The general initialization process is exactly like initializing a .NET form. It starts by initializing its two control objects contained by the user control (`rPasswordEditor` and `rLoginButton`) and setting public properties for them. The **`Location`** property settings for each control are relative to the client area of the user control. When the user control is added to a form, its `Location` property is then specified relative to the client area of the form, thus positioning all constituent controls of the user control on the form as a unit. The **`AutoScale*`** properties allow the user control and its controls to scale with its form, according to the system font size.

Note: The `CAST` function enables access to the explicit interface members, `BeginInit( )` and `EndInit( )`, which are required to initialize the `UltraTextEditor` control (see Accessing members of .NET interfaces on page 47).

This is the code for the sample ABL-derived client form class of `SampleUserControl`, `UserControlForm`.

```
CLASS UserControlForm INHERITS Progress.Windows.Form:

  /* Class private data */
  DEFINE PRIVATE VARIABLE demoControl AS CLASS SampleUserControl NO-UNDO.

  /* Class public members */
  METHOD PUBLIC VOID DoWait( ):
    WAIT-FOR System.Windows.Forms.Application:Run( THIS-OBJECT ).
  END METHOD.

  CONSTRUCTOR UserControlForm( ):
    InitializeComponents( ).
  END CONSTRUCTOR.

  METHOD PRIVATE VOID InitializeComponents( ):

    /* Creates base form class and all components */
    demoControl = NEW SampleUserControl( ).

    THIS-OBJECT:SuspendLayout( ).

    /* Set properties */
    THIS-OBJECT:AutoScaleDimensions = NEW System.Drawing.SizeF(6, 13).
```

```
    THIS-OBJECT:ClientSize = NEW System.Drawing.Size(290, 260).
    demoControl:Location = NEW System.Drawing.Point(0, 20).

    /* Add controls to form */
    THIS-OBJECT:Controls:Add( demoControl ).

    THIS-OBJECT:ResumeLayout( FALSE ).

  END METHOD.

END CLASS.
```

With the `SampleUserControl` class, form initialization in the ABL `UserControlForm` class becomes much simpler than for the previously described ABL-derived non-modal form classes (for example, `CurrentTimeForm` described in Sample ABL-derived .NET non-modal form on page 112). In this case, there is one object reference (`demoControl`) for all the controls on the form. Control initialization is a matter of using this one object reference to position the control container in the client area of the form by setting its `Location` property and to add the control container to the form's control collection. `UserControlForm` could also use the other public properties of `SampleUserControl` to subscribe event handlers to its control events, if the application required it.

4

Binding ABL Data to .NET Controls

In ABL, the browse is the most sophisticated UI widget for displaying data. Despite that sophistication, it is simple to use. You build a query against a temp-table to link the browse and the data. This integration between user interface components and data objects is called *data binding*.

ABL provides tight binding between its native data objects (database buffer fields, temp-table fields, variables, and so on) and its native user interface. You need little ABL code to display and update data in an ABL user interface. If you do not specify formatting, ABL even applies the default formatting to the data without any extra application code.

Data binding also exists in .NET. If a particular .NET control supports data binding, you have access to similar display and update capabilities. For example, binding a `System.Windows.Forms.DataGrid` to a `System.Data.DataTable` provides automatic data display and the capability to add, update, or delete rows. This data binding is accomplished through prescribed interfaces. Any data object that complies with the binding interface can bind with the .NET control.

To access data from an ABL data source object in a .NET control, ABL provides an intermediary binding object with the prescribed interface to act as a conduit between the two objects. This conduit is the Progress.Data.BindingSource object, which is referred to as the ProBindingSource in the documentation and the Progress Developer Studio for OpenEdge UI. The ProBindingSource extends the .NET binding source object, `System.Windows.Forms.BindingSource`, to facilitate binding ABL data to a .NET control.

For details, see the following topics:

- ABL and .NET data binding models
- ABL data binding model for .NET
- Understanding the ProBindingSource
- Managing data updates
- Example of an updatable grid

ABL and .NET data binding models

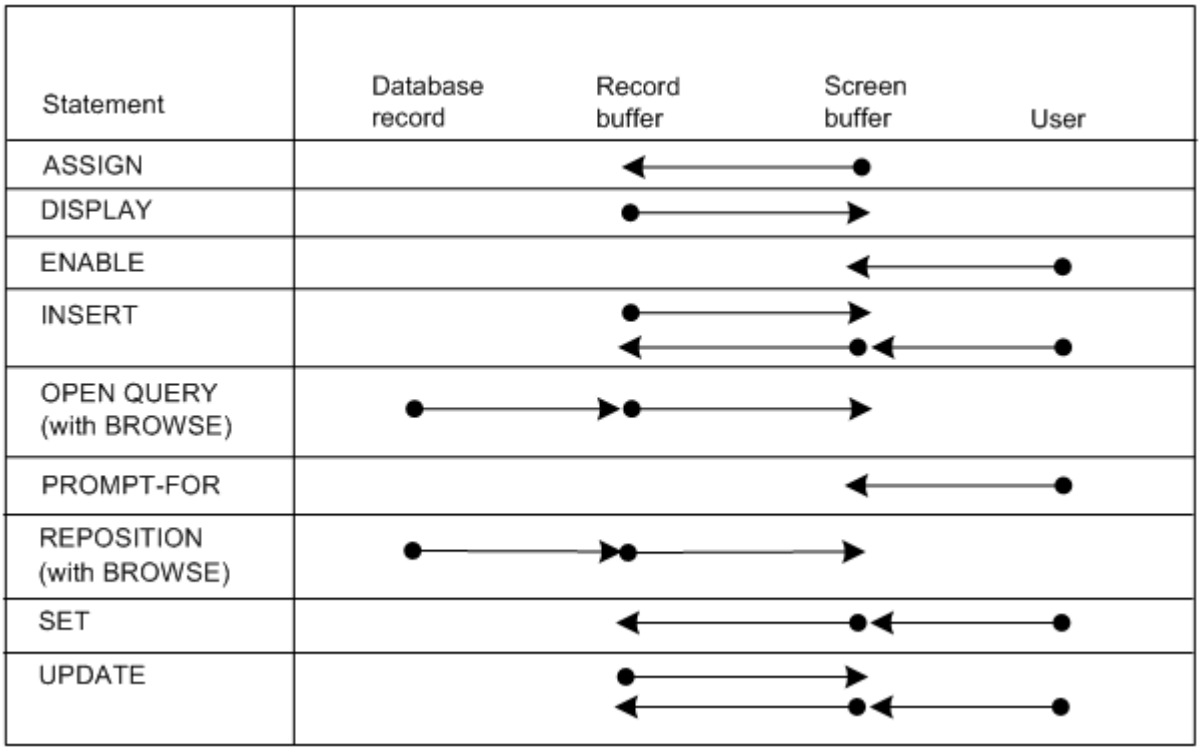
Data binding exists in both ABL and .NET. Native ABL data binding is action-based; the integration between UI widgets and data objects is encapsulated in the language statements that handle the data. Native .NET data binding is object-based; the integration is built into the controls.

ABL data binding

The ABL statements that handle data transfers have built-in data binding for static widgets. You can think of these statements as providing customized data binding for particular use cases. For example, the DISPLAY statement provides the functionality to transfer data from the record buffer to the screen buffer and automatically create a default UI object to display the data.

When you use these statements on static widgets, you get the functionality necessary to handle a particular action by default. Thus, each action controls the data binding needed to accomplish it. The following figure illustrates the different data binding support provided by various ABL statements.

Figure 10: Data binding by ABL statements



When you use ABL's dynamic UI programming, you bypass this built-in data binding. You have to explicitly create and configure widgets to hold your data. Then, you have to explicitly code how your data moves to and from widgets.

.NET data binding

In .NET, a specialized object, the `System.Windows.Forms.BindingSource`, can provide data binding between a control and a data object. The .NET binding source encapsulates a currency manager and a set of interfaces with which the data object must comply. The *currency* manager controls the position of the cursor in the data. The interfaces provide methods and properties that a bound control can access. The interfaces also provide events to notify a bound control about changes in the data object.

A bound control considers the binding source to be its data source. In turn, the binding source sets the underlaying data object as its data source. The following C# code fragment shows two controls bound to a binding source:

```
DataTable custTable = custOrderDataSet.Tables[Customer];
bindingSource = new BindingSource();
bindingSource.DataSource = custTable;

customerGrid = new DataGrid();
customerGrid.DataSource = bindingSource;
textBoxName = new TextBox();

textBoxName.DataBindings.Add(Text, bindingSource, Name);
```

Because the controls use the same `BindingSource` instance, they share currency. When you select a new row in the `customerGrid`, `textBoxName` displays the value of the new row's `Name`. If you do not want the two controls to share currency, you need two binding sources for the `Customer` table, binding each control to a different instance.

ABL data binding model for .NET

ABL supports binding an ABL data source object to a .NET control through the `Progress.Data.BindingSource` object. The ProBindingSource is an OpenEdge built-in class that extends the .NET `System.Windows.Forms.BindingSource` class. The `Progress.Data.BindingSource` class contains the added functionality needed to make ABL data source objects comply with the `System.Windows.Forms.BindingSource` data binding interfaces. When you bind an ABL data source object to a .NET control, you must use the ProBindingSource.

Extent of integration with .NET controls

In the general model of data binding, a data source has no semantic understanding of any controls that are bound to it. In the ABL model, this means that the ProBindingSource does not know what .NET controls are bound to it. The ProBindingSource presents the data for any control to use, and the .NET control accesses the data that it can use.

However, without a semantic understanding of the controls, ABL data-handling statements cannot provide the automatic behaviors that you see with ABL UI widgets. For example, the ABL statement, "`DISPLAY customer.name.`", cannot create an `Infragistics.Win.UltraWinEditors.UltraTextEditor`, set its format properties, and provide an `Infragistics.Win.Misc.UltraLabel` as a default label.

Instead, your application must explicitly create the .NET controls and set their properties. If you have worked with dynamic ABL widgets, you are familiar with this style of programming by setting widget attributes. You cannot use the ABL data-handling statements to manipulate screen values in .NET controls. Your application must perform such tasks through object manipulation.

The reverse case is also true: bound .NET controls have no direct understanding of the bound ABL data source objects. The .NET controls cannot access the schema of a bound ABL data source object to find the default format, validation phrase, or help text for a field. The .NET controls can only access what the binding interfaces in the `Progress.Data.BindingSource` class make available.

Supported .NET controls

Not all .NET controls support data binding. Since the ProBindingSource is an extension of the .NET `BindingSource`, you can only bind ABL data source objects to .NET controls that can use that class. The exact mechanism of the data binding depends on how each control implements data binding. Two common methods are using a data source property to specify the ProBindingSource or using a bindings collection.

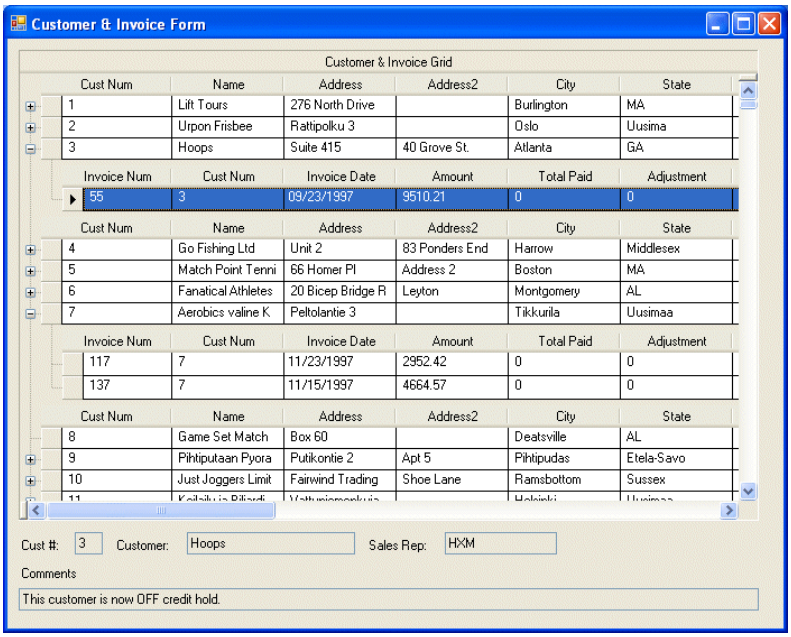
Supported ABL data sources

Only certain ABL data objects can act as the data source for the ProBindingSource. The following list shows the ABL data objects that can act as data sources to the ProBindingSource:

- Database or temp-table buffers:
  - Buffer fields can be part of static or dynamic temp-tables.
  - Buffer fields can be part of temp-tables in ProDataSets.
  - Fields cannot be the `RAW` data type.
- Database or temp-table queries:
  - Queries can be static or dynamic.
  - Queries can be a navigation query of ProDataSets.
  - If a temp-table contains a `RAW` field, the field is ignored by default. If you explicitly ask for it, ABL raises an error.
- ProDataSets

  In ABL applications, you can only bind a single temp-table from a ProDataSet to a browse widget. If you want data from more than one temp-table in your browse, you first join the separate temp-tables into a new temp-table.

  However, you can bind a static or dynamic ProDataSet, as a whole, to the ProBindingSource. This enables you to create a hierarchical display of parent records and their child records in the same .NET control. For example, this Infragistics.Win.UltraWinGrid.UltraGrid displays the data from a ProDataSet containing `Customer` and `Invoice` tables:



Understanding the ProBindingSource

As described in previous sections, the ProBindingSource is an OpenEdge built-in class that provides a customized version of the .NET `BindingSource` object. The ProBindingSource provides extended constructors, properties, methods, and events that enable you to bind an ABL data source object to .NET objects.

The following sections describe the class members of the Progress.Data.BindingSource class that you should use to bind to an ABL data source object. For details about these class members, see OpenEdge Development: ABL Reference.

Constructors

The ProBindingSource provides several overloaded constructors to handle binding to the various ABL data source objects. The following syntax shows the various constructors:

Syntax

```
PUBLIC BindingSource ( INPUT query-hdl AS HANDLE
                  [ , INPUT include-fields AS CHARACTER,
                       INPUT except-fields AS CHARACTER] )

PUBLIC BindingSource ( INPUT dataset-hdl AS HANDLE
                [ , { INPUT parent-buffer-name AS CHARACTER
                    | INPUT parent-buffer-hdl AS HANDLE } ]
                  [ , INPUT include-fields AS CHARACTER,
                      INPUT except-fields AS CHARACTER ] )

PUBLIC BindingSource ( INPUT buffer-hdl AS HANDLE
                  [ , INPUT include-fields AS CHARACTER,
                      INPUT except-fields AS CHARACTER] )

PUBLIC BindingSource ( )
```

Field lists

As optional parameters, the constructors can accept comma-separated lists of data object fields, in display order, to make available to or exclude from the UI. If you want to make only a few fields available, list them in the included field list. If you want to make most fields available, set the available fields to be all fields and list the remaining fields in the excluded field list. When the included field list is an asterisk (*), all fields in the data object are made available to any bound UI control.

The excluded field list is only meaningful when the included field list is an asterisk (*). If the included field list contains specific fields, the excluded field list is ignored. However, when you are not specifying any excluded fields, you still must specify a blank value ("") to match the constructor's signature.

Note: The ProBindingSource automatically excludes `RAW` fields; you do not have to explicitly exclude them. If you add them to the included field list, ABL raises a run-time error.

Some UI controls need to access and expose schema definition information from the data object, such as a grid needing column headers. The ProBindingSource makes the metadata available to the bound control. For the grid example, the ProBindingSource creates a field label that follows the same rules as column headers for an ABL Browse widget. The ProBindingSource uses, in order, the first of the following as the field label: the column label, the field label, or the field name.

When one of the fields is an array, the ProBindingSource treats each array element as a separate field. To generate unique names for each element, the ProBindingSource appends a suffix, [*n*], to the field name. Thus, if you wanted to include only the first three months of a monthly quota array, you could add the following field names to your include list:

```
. . . MonthQuota[1], MonthQuota[2], MonthQuota[3], . . .
```

When a ProBindingSource includes fields from multiple tables, consider qualifying any ambiguous field names. Otherwise, the first field matching that name is used.

The following sections discuss the particular situations that might arise with each kind of data object.

Binding to queries

You can bind the ProBindingSource to a query on a temp-table or database table using the following syntax:

Syntax

```
PUBLIC BindingSource ( INPUT query-hdl AS HANDLE
                  [ , INPUT include-fields AS CHARACTER,
                         INPUT except-fields AS CHARACTER] )
```

Where *query-hdl* is the query handle, *include-fields* is an optional, comma-separated list of fields from the data object to make available to the UI, and *except-fields* is a comma-separated list of fields from the data object to exclude from the UI. Note that you must supply the included and excluded field lists together whenever you use them.

Note: You must use the query handle to specify the query, even if the query is static.

You cannot simultaneously bind the same query to multiple ProBindingSources. A query can only be bound to a single ProBindingSource at any time. If you try to bind a query that is already bound to another ProBindingSource, the ProBindingSource throws an error.

Generally, you should open the ProBindingSource's query with the PRESELECT option, because the ProBindingSource needs the actual record count in the query at several points. Using this option optimizes getting the record count. If your application code does not specify this option for a dynamic query, the AVM applies the option. However, this behavior is less efficient than specifying the `PRESELECT` option yourself.

For large result sets, opening the query with the `PRESELECT` option might be time-consuming. In these cases, you might instead specify the `MaxDataGuess` property which provides the bound .NET controls with an approximate record count. The bound .NET control is immediately rendered based on the approximation and corrects itself when the actual record count is available.

You can use either a static or dynamic query for the ProBindingSource's data object. But, you must choose a scrolling query. If you use a static query, you must define it with the `SCROLLING` keyword. Dynamic queries are scrolling by default. Do not change the default behavior by setting FORWARD-ONLY to `TRUE`.

If you use a join query, you should qualify any ambiguous field names with the appropriate buffer name. Because queries can do self-joins, the qualifier must be the buffer name, rather than the table name. Using the buffer name also handles joins across databases. Your application must use an explicitly defined buffer for such joins. The buffer definition provides the database and table name for the ambiguous field.

For example, you have a buffer, `bMgr`, for the `Employee` table. You use that buffer in the following query, `EMqry`:

```
FOR EACH Employee, EACH bMgr WHERE Employee.Manager = bMgr.Name
```

If you wanted a ProBindingSource to make available the employee's name, his manager's name, and the manager's phone number, the signature is as follows:

```
PUBLIC BindingSource ( INPUT EMqry AS HANDLE,
            INPUT "Employee.Name, Employee.Manager, bMgr.Phone" AS CHARACTER,
             INPUT "" AS CHARACTER)
```

Binding to buffers

You can bind the ProBindingSource to a buffer using the following syntax:

Syntax

```
PUBLIC BindingSource ( INPUT buffer-hdl AS HANDLE
                  [ , INPUT include-fields AS CHARACTER,
                         INPUT except-fields AS CHARACTER ] )
```

Where *buffer-hdl* is the buffer handle, *include-fields* is an optional, comma-separated list of fields from the data object to make available to the UI, and *except-fields* is a comma-separated list of fields from the data object to exclude from the UI. Note that you must supply the included and excluded field lists together whenever you use them.

Note: You must use the buffer handle to specify the buffer, even if the buffer is static.

You cannot simultaneously bind the same buffer to multiple ProBindingSources. A buffer can only be bound to a single ProBindingSource at any time. If you try to bind a buffer that is already bound to another ProBindingSource, the ProBindingSource throws an error.

Typically, you use a buffer when the ProBindingSource supplies data for single-value UI controls, like text boxes or toggles. In such cases, there is only one record in the buffer. Because the ProBindingSource's Position property works off a 0-based index, this record always yields a `Position` value of zero. As a result, you manage currency by changing the row in the data object programmatically.

In the typical case, you have multiple UI controls that each bind to a different field in the buffer. So, any other available fields do not matter much. If you have a buffer with a large number of fields that the UI does not need, you should consider using the optional field list parameters to improve performance.

Binding to ProDataSets

When you bind a ProBindingSource to a ProDataSet, you make the hierarchy of tables, starting at one particular parent buffer, available to the bound UI controls. You can bind the ProBindingSource to a ProDataSet using the following syntax:

Syntax

```
PUBLIC BindingSource ( INPUT dataset-hdl AS HANDLE
                [ , { INPUT parent-buffer-name AS CHARACTER
                    | INPUT parent-buffer-hdl AS HANDLE } ]
                [ , INPUT include-fields AS CHARACTER,
                       INPUT except-fields AS CHARACTER ] )
```

Where *dataset-hdl* is the ProDataSet handle, *parent-buffer-name* is the name of a parent buffer object, *parent buffer-hdl* is the handle of a parent buffer object, *include-fields* is an optional, comma-separated list of fields from the data object to make available to the UI, and *except-fields* is a comma-separated list of fields from the data object to exclude from the UI. Note that you must supply the included and excluded field lists together whenever you use them.

Note: If you have to prepare one of the navigation queries, you should use the PRESELECT option, as described in Binding to queries on page 137.

You cannot simultaneously bind the same ProDataSet query to multiple ProBindingSources. A ProDataSet query can only be bound to a single ProBindingSource at any time. If you try to bind a ProDataSet query that is already bound to another ProBindingSource, the ProBindingSource throws an error.

Note: Although you specify the parent buffer or its handle, you are actually binding to the ProDataset's query for that buffer.

The buffer that you choose as the parent buffer controls what tables in the ProDataSet are available through the ProBindingSource. Typically, you choose a top-level buffer for the parent buffer. If you do not specify a parent buffer, the ProBindingSource defaults to using the first top-level buffer listed in the ProDataSet's definition. A ProBindingSource can only make available a single top-level buffer from a ProDataSet.

However, you can choose a buffer that is not a top-level buffer. In that case, the hierarchy of available tables in the ProBindingSource starts with that table and extends to its children. In effect, the ProBindingSource considers its parent buffer as the top-level buffer in the part of the ProDataSet that it makes available. For example, consider a ProDataSet that contains the `Customer`, `Order`, and `Orderline` tables. If you create a ProBindingSource and specify the `Order` table as the parent buffer, only the `Order` and `Orderline` tables are accessible through that ProBindingSource.

Note: When the ProDataSet as a whole is the data object for a ProBindingSource, the ProBindingSource does not show the BEFORE-TABLE for each primary table as an available table. However, you can explicitly specify a `BEFORE-TABLE` as the parent buffer of another ProBindingSource to make that data available to a bound UI control.

When the specified parent buffer is a top-level buffer in the ProDataSet, the ProBindingSource uses the ProDataSet's top-level navigation query, TOP-NAV-QUERY, and the relationship queries for each of its children to navigate the records in the ProDataSet. When the specified parent buffer is not a top-level buffer, the ProBindingSource uses the parent buffer's relationship query to populate the primary set of records and the relationship queries for any child buffers to populate the subsequent levels.

If you do not use the optional field list parameters, the ProBindingSource makes all fields in the parent buffer and any child buffers available to bound UI controls. When you use the field lists with a ProDataSet-bound ProBindingSource, you must qualify all fields with their buffer. When you specify a group of included fields, you must include at least one field from every buffer that is available through the ProBindingSource. If you do not, an error is raised.

Certain .NET UI controls can display data from both parent and child tables within the same control. For example, the Infragistics.Win.UltraWinGrid.UltraGrid can display the associated rows from a child table under each row from the parent table. For such controls, a single ProBindingSource that makes multiple tables from a ProDataSet available works well.

ProDataSet tables with uncommitted changes might have the ERROR and ERROR-STRING attributes set for one or more rows. The ProBindingSource's Refresh( ) method picks up any modifications to these attributes. The ProBindingSource can make the errors available to a bound UI control. If the UI control is sensitive to these settings on its data source, the UI control adjusts its display to show the errors. For example, a Microsoft `System.Windows.Forms.DataGridView` displays an error icon on the row header and displays the error string when a mouse hovers over the icon. In the UltraGrid, you need to set the `DisplayLayout:Override:SupportDataErrorInfo` attribute to see the error icons.

Special case: unbound class instance

If you invoke the `Progress.Data.BindingSource` constructor without supplying any parameters, you create an unbound class instance. At run time, you can bind the ProBindingSource instance to an ABL data source object by setting the Handle property to the data object's handle.

The ProBindingSource follows its default behavior for binding to the specified data object. The ProBindingSource makes all fields in the data object available to the UI control. In the case of a ProDataSet, it binds to the first top-level buffer in the ProDataSet. If you do not want to make all fields available, you can use the ProBindingSource's `SetFields( )` method to choose which fields are exposed to the .NET controls.

The `SetFields( )` method uses the same parameters as the ProBindingSource constructors. The first parameter is an *`include-fields`* list. The second parameter is an *`except-fields`* list. The final parameter depends on the type of data source object to which the ProBindingSource will bind:

- For a ProDataSet, the final parameter is either the *`parent-buffer-name`* or *`parent-buffer-hdl`*.
- For any other data source, the final parameter must be an empty string, "".

Note: You must use the `SetFields( )` method before you specify the `Handle` property. If you use the `SetFields( )` method after binding to a data source object, the ProBindingSource throws a .NET exception.

The following code fragment shows the correct sequence for using this method:

1. Create an unbound ProBindingSource instance.
2. Use `SetFields( )` to specify the appropriate columns to bind.
3. Set the ProBindingSource's `Handle` property to bind the data source object.

```
DEFINE VARIABLE pbs AS Progress.Data.BindingSource.

/* 1 */
pbs = NEW Progress.Data.BindingSource().

/* 2 */
pbs:SetFields("Customer.CustNum,Customer.Name,Customer.State","","").

/* 3 */
pbs:Handle = myQryHdl.
```

Properties

The ProBindingSource provides properties to handle binding to the various ABL data source objects. The following table lists these extended properties. For more information about indexed properties, see Accessing .NET indexed properties and collections on page 52.

Property	Data source objects	Access and Type	Description
AllowEdit	Buffer, query, ProDataSet	R/W LOGICAL	Indicates whether the .NET control should allow the user to edit values in the bound ABL data source object. The default value is TRUE. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control. Use ChildAllowEdit to enable editing for child tables.
AllowNew	Query, ProDataSet	R/W LOGICAL	Indicates whether the .NET control should allow the user to add new records to the bound ABL data source object. The default value is TRUE. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control. Use ChildAllowNew to enable adding for child tables.
AllowRemove	Query, ProDataSet	R/W LOGICAL	Indicates whether the .NET control should allow the user to remove records from the bound ABL data source object. The default value is TRUE. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control. Use ChildAllowRemove to enable removing for child tables.
AutoSort	Query, ProDataSet	R/W LOGICAL	Indicates whether the ProBindingSource object automatically resorts records in the ABL data source object in response to appropriate user actions in the bound .NET control. The default value is FALSE. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control.
AutoSync	Buffer, query, ProDataSet	R/W LOGICAL	Indicates whether the ProBindingSource object automatically synchronizes (refreshes) all data displayed in any bound .NET control after one of the following ABL operations on the bound ABL data source object: • Reopening the query associated with the data source • Repositioning the query associated with the data source using either the REPOSITION statement or any of the REPOSITION methods The default value is TRUE. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control.

Property	Data source objects	Access and Type	Description
`AutoUpdate`	Buffer, query, ProDataSet	R/W `LOGICAL`	Indicates whether the ProBindingSource object automatically updates records in the ABL data source object in response to appropriate user actions in the bound .NET control. The default value is `FALSE`. **Note:** Intended only for rapid prototyping purposes.
`Batching`	Query, ProDataSet	R/W `LOGICAL`	Indicates whether record batching is enabled for the ProBindingSource object. The default value is `FALSE`, which disables record batching. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control.
`ChildAllowEdit[` *`buffer-handle | buffer-name`* `]`	ProDataSet	R/W `LOGICAL`	An indexed property that indicates whether the .NET control should allow the user to edit values in a specified child temp-table buffer in the bound ABL data source object. The default value is `TRUE`.
`ChildAllowNew[` *`buffer-handle | buffer-name`* `]`	ProDataSet	R/W`LOGICAL`	An indexed property that indicates whether the .NET control should allow the user to add new records to the specified child temp-table buffer in the bound ABL data source object. The default value is `TRUE`.
`ChildAllowRemove[` *`buffer-handle | buffer-name`* `]`	ProDataSet	R/W`LOGICAL`	An indexed property that indicates whether the .NET control should allow the user to remove records from the specified child temp-table buffer in the bound ABL data source object. The default value is `TRUE`.
`ChildInputValue[` *`buffer-handle | buffer-name`* `]`	ProDataSet	Read-only `Progress.Data.InputValue`	Returns a Progress.Data.InputValue instance containing input values for all fields in the current row of the specified child temp-table displayed in the bound .NET control. Use the indexers in this instance to access the input value of a specific field in the row.
`Count`	Buffer, query, ProDataSet	Read-only `INTEGER`	The number of records in the result set for the query associated with the top-level table displayed in the bound .NET control. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control.

Property	Data source objects	Access and Type	Description
`Handle`	Buffer, query, ProDataSet	R/W `HANDLE`	The handle to the ABL data source object to which the ProBindingSource object is bound. You can use this property to associate an ABL data source object with an unbound ProBindingSource instance at run time.
`InputValue[` *`field-index`* `|` *`field-name`* `]` `Keywords:`	Buffer, query, ProDataSet	Read-only <*same as field*>	Returns the input value of the specified field in the current row of the top-level table displayed in the bound .NET control. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control.
`MaxDataGuess`	Query, ProDataSet	R/W `INTEGER`	An estimate of the number of records that a query returns.
`NewRow`	Query, ProDataSet	Read-only `LOGICAL`	Indicates whether the current row in the bound .NET control is a newly created row that can still be undone. When bound to a ProDataSet object, this property applies only to the top-level table displayed in the .NET control.
`NoLOBs`	Buffer, query, ProDataSet	R/W `LOGICAL`	Specifies whether or not the AVM ignores BLOB or CLOB fields while executing the `Assign( )` method or the `CURRENT-CHANGED` function.
`Position`	Buffer, query, ProDataSet	R/W `INTEGER`	The zero-based position (index) of the current row in the bound .NET control.
`RowModified`	Buffer, query, ProDataSet	Read-only `LOGICAL`	Indicates whether the current row in the bound .NET control is currently being edited.
`TableSchema`	Buffer, query, ProDataSet	R/W `Progress.Data.TableDesc`	Intended for design time use only. Allows code generated by the Progress Developer Studio for OpenEdge's Visual to specify schema at design time.
`Tag`	Buffer, query, ProDataSet	R/W `System.Object`	An arbitrary user-defined object containing any information or data that you want for the ProBindingSource object. The default is `Unknown value (?).`

Note: A particular bound UI control might not support a particular ProBindingSource property. By the standard .NET protocol, the bound UI control reads the properties that it supports and adjusts its behavior to match. For example, a Microsoft `System.Windows.Forms.TextBox` does not support record deletion. Even when linked to a ProBindingSource that has AllowRemove set to `TRUE`, you cannot use the `TextBox` for record deletion.

AutoUpdate and rapid prototyping

Because of the standard interaction between bound UI controls and the ProBindingSource, the ProBindingSource can perform basic update tasks automatically. The ProBindingSource can create and delete records in response to appropriate user actions. This capability might be useful during the rapid prototyping stage of your development.

You control this behavior with the AutoUpdate property. Note that using this automatic behavior has undesirable limitations that make it inappropriate for most business applications. Because of these limitations, Progress Software Corporation recommends that you not use this behavior outside a rapid prototyping environment. In accord with this recommendation, the `AutoUpdate` property defaults to `FALSE`.

Before deciding to use automatic updates outside a rapid prototyping environment, consider the following limitations carefully:

- The ProBindingSource can only update the buffer from the screen values after all validation opportunities have passed and the transaction is complete from the UI control's perspective.

  To maintain separation between UI and business logic, a business application should perform data validation on the buffer fields, rather than on the screen values. Using this approach, the application probably moves the screen values into the buffer itself, either directly or with the ProBindingSource's Assign( ) method, then does the validation before the ProBindingSource updates the buffer automatically. This technique eliminates the utility of this part of the automatic behavior.

  A business application must handle any errors that occur while updating the database or temp-table. Because the UI control considers the transaction complete before the ProBindingSource makes its updates, the application could not effectively tell the UI control about errors encountered during creates, assigns, and deletes. The UI control would not behave appropriately when errors occur.

- A business application commonly has initial values that must be set programmatically during a create operation. The ProBindingSource's automatic behavior does not include setting these values. Therefore, the application needs to set these values itself, reducing the utility of the automatic behavior.
- A business application often needs to make changes to related data during an update or delete operation. In case the changes are canceled, the application needs all the changes scoped to a single, larger transaction. The ProBindingSource's automatic behavior cannot accommodate this requirement.
- The ProBindingSource might have a join query as its data source object. In this case, the ProBindingSource does not know which records in the joined tables need to be created or deleted. For example, consider a query that joins the Customer and Salesrep tables to provide the UI control with each customer's sales region. If a user deletes a Customer record, the application should not also delete the Salesrep record.
- The CreateRow and CancelCreateRow events override the `AutoUpdate` behavior. If you subscribe to these events, your application does not check the value of `AutoUpdate` when it creates or cancels a new row.

Methods

The ProBindingSource provides several extended methods to handle binding to the various ABL data source objects. The following table lists these extended methods.

Method	Returns	Description
`Assign( )`	`LOGICAL`	Assigns input values from the current record in the bound UI control to the corresponding record in the bound ABL data source object.
`Dispose( )`	`VOID`	Cleans up ProDataSet resources held by a ProBindingSource object while awaiting .NET garbage collection. If the ProBindingSource binds a ProDataSet, call this method before releasing it for garbage collection.
`Refresh( [ INPUT record-index AS INTEGER ] )`	`VOID`	Refreshes the displayed field values for the specified record in all bound UI controls with the values in the bound ABL data source object. When used with a bound ProDataSet, this method applies to only the top-level table.
`RefreshAll( )`	`VOID`	Refreshes the field values displayed for all rows (parent and child) in any bound .NET control with the values from the corresponding records in the bound ABL data source object.
`SetFields( INPUT` *`include-fields`* `AS CHARACTER, INPUT` *`except-fields`* `AS CHARACTER, { "" | INPUT` *`parent-buffer-hdl`* `AS HANDLE | INPUT` *`parent-buffer-name`* `AS CHARACTER } )`	`VOID`	Specifies the fields that an unbound ProBindingSource should expose from whatever data source object binds to it. The final parameter can be an empty string (""), a buffer handle, or a buffer name. **Note:** You must use this method before the ProBindingSource binds to the data source object. Using the method after binding the data source object cause the ProBindingSource to throw a .NET exception.

Events

The ProBindingSource provides several extended events to handle binding to the various ABL data source objects. These events rely on the .NET `System.EventArgs` class or built-in OpenEdge classes that extend the .NET class. The following table lists these extended events.

Event	Description
`CancelCreateRow`	Published when some user action cancels a create row operation (for example, pressing **ESCAPE** in a new empty row).
`CreateRow`	Published when some user action initiates a create row operation (for example, clicking in a new empty row at the bottom of a control).

Event	Description
OffEnd	Published when record batching is enabled and some user action reaches the last row of the current result set. Use this event to retrieve the next batch of records from the AppServer.
PositionChanged	Published when the value of the Position property changes either programmatically or through a UI action.
SortRequest	Published when some user action initiates a sort operation (for example, clicking on a column header in a grid control).

Note: These events signal changes in a bound UI control or a ProBindingSource. For changes to a bound ABL data source object, use standard ABL mechanisms, such as triggers.

Data binding examples

The following sections show basic examples of using the ProBindingSource to bind to various ABL data source objects.

Note: To bind a ProBindingSource to a .NET control, the .NET control must support data binding. There are several approaches by which a control might accomplish data binding. Two common approaches are using either a DataSource or a DataBindings property. Consult the documentation for your controls to see how they support data binding.

Query binding example

QueryBinding.p uses a ProBindingSource to bind a query on the Customer table to a grid. First, the procedure creates the query. Then, it creates the ProBindingSource. To disable edits, the procedure sets the ProBindingSource's AllowEdit and AllowRemove properties to FALSE. It does not need to set AllowNew because adding new rows is disabled by default in the grid's properties. Finally, it binds the grid to the ProBindingSource through the grid's DataSource property.

```
/* QueryBinding.p
   Bind to a Customer query and display the entire table in an Infragistics UltraWinGrid
 */
/* USING statements must be the first in the procedure. Note that you could
   have USING statements for the OpenEdge classes also.*/
USING System.Windows.Forms.*.
USING Infragistics.Win.UltraWinGrid.*.

DEFINE VARIABLE rMainForm  AS Progress.Windows.Form       NO-UNDO.
DEFINE VARIABLE rCustGrid  AS UltraGrid                   NO-UNDO.
DEFINE VARIABLE rBindS     AS Progress.Data.BindingSource NO-UNDO.
DEFINE VARIABLE controls   AS Control+ControlCollection   NO-UNDO.

DEFINE VARIABLE hCustQuery AS HANDLE                      NO-UNDO.

CREATE QUERY hCustQuery.hCustQuery:SET-BUFFERS(BUFFER Customer:HANDLE).
hCustQuery:QUERY-PREPARE("PRESELECT EACH Customer").
hCustQuery:QUERY-OPEN.

/* This will display all of the Customer fields in the grid. */
```

```
rBindS = NEW Progress.Data.BindingSource(hCustQuery).

/* Alternately, specify fields using the optional include-fields and
   except-fields lists. */

/* rBindS = NEW Progress.Data.BindingSource(hCustQuery,
       "CustNum,Name,Address,City,PostalCode,Phone,Contact,Salesrep",""). */
/* Disable editing because procedure does not include event logic to handle
   changes. */
rBindS:AllowEdit = FALSE.
rBindS:AllowRemove = FALSE.

/* Main block */
IF VALID-OBJECT(rBindS) THEN
  DO ON ERROR UNDO, LEAVE:

    rMainForm        = NEW Progress.Windows.Form().
    rMainForm:Width  = 840.
    rMainForm:Height = 500.
    rMainForm:Text   = "Customer Form".

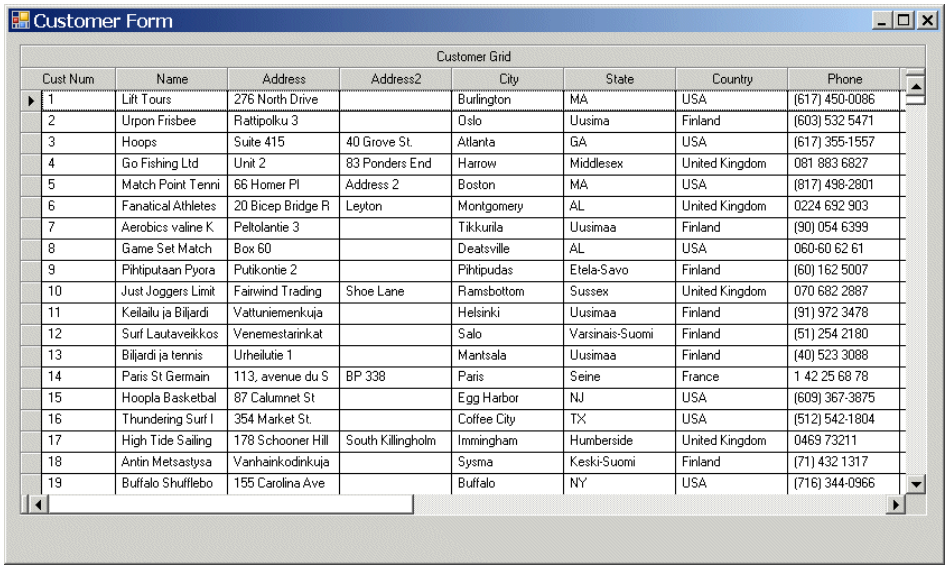
    rCustGrid            = NEW UltraGrid().
    rCustGrid:Left       = 10.
    rCustGrid:Top        = 10.
    rCustGrid:Width      = 810.
    rCustGrid:Height     = 420.
    rCustGrid:Name       = "CustomerGrid".
    rCustGrid:Text       = "Customer Grid".
    rCustGrid:DataSource = rBindS.
    rCustGrid:TabIndex   = 1.

    controls = rMainForm:Controls.
    controls:Add(rCustGrid).

    WAIT-FOR Application:RUN(rMainForm).
END. /* Main block */
```

When the procedure runs, a simple form appears displaying a customer grid, as shown in the following figure.

Figure 11: Grid bound to query



Buffer binding example

`BufferBinding.p` uses a ProBindingSource to bind fields from the Customer table to .NET UI controls. First, the procedure gets the handle of the `Customer` table buffer. Then, it creates the ProBindingSource. Finally, it binds the text boxes to the ProBindingSource through their `DataBindings` property.

```
/* BufferBinding.p
    Bind to a buffer on the Sports2000.Customer table and display some fields
    on a simple form. */

/* USING statements must be the first in the procedure. Note that you could
   have USING statements for the OpenEdge classes also.*/
USING System.Windows.Forms.*.

DEFINE VARIABLE rMainForm  AS Progress.Windows.Form       NO-UNDO.
DEFINE VARIABLE rBindS     AS Progress.Data.BindingSource NO-UNDO.
DEFINE VARIABLE rControls  AS Control+ControlCollection   NO-UNDO.
DEFINE VARIABLE numTBox    AS TextBox NO-UNDO.
DEFINE VARIABLE nameTBox   AS TextBox NO-UNDO.
DEFINE VARIABLE phoneTBox  AS TextBox NO-UNDO.
DEFINE VARIABLE numLabel   AS Label   NO-UNDO.
DEFINE VARIABLE nameLabel  AS Label   NO-UNDO.
DEFINE VARIABLE phoneLabel AS Label   NO-UNDO.
DEFINE VARIABLE bCustHdl   AS HANDLE  NO-UNDO.

bCustHdl = BUFFER Sports2000.Customer:HANDLE.

FIND FIRST Customer.rBindS = NEW Progress.Data.BindingSource(bCustHdl).

/* Main block */
IF VALID-OBJECT(rBindS) THEN
  DO ON ERROR UNDO, LEAVE:

    rMainForm        = NEW Progress.Windows.Form().
    rMainForm:Width  = 300.
    rMainForm:Height = 200.
    rMainForm:Text   = "Customer Form".
    numTBox          = NEW TextBox().
    numTBox:Left     = 50.
    numTBox:Top      = 40.
    numTBox:Width    = 50.
    numTBox:Height   = 40.
    numTBox:Name     = "numcntl".
    numTBox:ReadOnly = TRUE.

    numLabel        = NEW Label().
    numLabel:Left   = 25.
    numLabel:Top    = 45.
    numLabel:Width  = 75.
    numLabel:Height = 15.
    numLabel:Text   = "Customer #:".

    nameTBox        = NEW TextBox().
    nameTBox:Left   = 50.
    nameTBox:Top    = 60.
    nameTBox:Width  = 150.
    nameTBox:Height = 40.
    nameTBox:Name   = "namecntl".

    nameLabel        = NEW Label().
    nameLabel:Left   = 25.
    nameLabel:Top    = 65.
    nameLabel:Width  = 75.
    nameLabel:Height = 15.
    nameLabel:Text   = "Customer:".
```

```
    phoneTBox        = NEW TextBox().
    phoneTBox:Left   = 50.
    phoneTBox:Top    = 80.
    phoneTBox:Width  = 100.
    phoneTBox:Height = 40.
    phoneTBox:Name   = "phonecntl".

    phoneLabel        = NEW Label().
    phoneLabel:Left   = 25.
    phoneLabel:Top    = 85.
    phoneLabel:Width  = 50.
    phoneLabel:Height = 15.
    phoneLabel:Text   = "Phone #:".

    rControls = rMainForm:Controls.
    rControls:Add(numTBox).
    rControls:Add(numLabel).
    rControls:Add(nameTBox).
    rControls:Add(nameLabel).
    rControls:Add(phoneTBox).
    rControls:Add(phoneLabel).

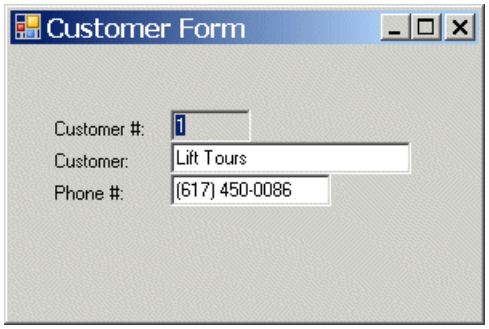
    numTBox:DataBindings:Add("Text", rBindS, "CustNum").
    nameTBox:DataBindings:Add("Text", rBindS, "Name").
    phoneTBox:DataBindings:Add("Text", rBindS, "Phone").

    WAIT-FOR Application:Run(rMainForm).

END. /* Main block */
```

When this procedure runs, a simple form appears displaying the first customer's number, name, and phone number, as shown in the following figure.

Figure 12: Form bound to database buffer



ProDataSet binding example

`ProDataSetBinding.p` uses a ProBindingSource to bind a ProDataSet to a hierarchical grid. First, the procedure creates the ProDataSet. Then, it creates the ProBindingSource. It binds the grid to the ProBindingSource through the grid's `DataSource` property. Finally, it sets up some text boxes and binds them to display particular fields from the `Customer` table.

```
/* ProDataSetBinding.p
   Bind to a ProDataSet for the Customer and Invoice tables then display the
   records hierarchically in an Infragistics UltraWinGrid. */

/* USING statements must be the first in the procedure. Note that you could
   have USING statements for the OpenEdge classes also.*/
USING System.Windows.Forms.*.
```

```
USING Infragistics.Win.UltraWinGrid.*.

DEFINE VARIABLE rMainForm      AS Progress.Windows.Form       NO-UNDO.
DEFINE VARIABLE rCustInvGrid   AS UltraGrid                   NO-UNDO.
DEFINE VARIABLE rCustNumTB     AS TextBox                     NO-UNDO.
DEFINE VARIABLE rCustNumLabel  AS Label                       NO-UNDO.
DEFINE VARIABLE rCustNameTB    AS TextBox                     NO-UNDO.
DEFINE VARIABLE rCustNameLabel AS Label                       NO-UNDO.
DEFINE VARIABLE rSalesRepTB    AS TextBox                     NO-UNDO.
DEFINE VARIABLE rSalesRepLabel AS Label                       NO-UNDO.
DEFINE VARIABLE rCommentTB     AS TextBox                     NO-UNDO.
DEFINE VARIABLE rComLabel      AS Label                       NO-UNDO.
DEFINE VARIABLE rBindS         AS Progress.Data.BindingSource NO-UNDO.
DEFINE VARIABLE rControls      AS Control+ControlCollection   NO-UNDO.

DEFINE VARIABLE hTTCustomer    AS HANDLE                      NO-UNDO.
DEFINE VARIABLE hTTInvoice     AS HANDLE                      NO-UNDO.

DEFINE VARIABLE hDataSet       AS HANDLE                      NO-UNDO.
DEFINE VARIABLE hBufTTCust     AS HANDLE                      NO-UNDO.
DEFINE VARIABLE hBufTTInv      AS HANDLE                      NO-UNDO.
DEFINE VARIABLE hTopQuery      AS HANDLE                      NO-UNDO.

DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE Customer.
DEFINE TEMP-TABLE ttInvoice  NO-UNDO LIKE Invoice.

DEFINE DATASET dsCustInv FOR ttCustomer, ttInvoice
  DATA-RELATION FOR ttCustomer, ttInvoice RELATION-FIELDS(CustNum, CustNum).

hDataSet    = DATASET dsCustInv:HANDLE.

hTTCustomer = TEMP-TABLE ttCustomer:HANDLE.
hTTInvoice  = TEMP-TABLE ttInvoice:HANDLE.

hBufTTCust  = hTTCustomer:DEFAULT-BUFFER-HANDLE.
hBufTTInv   = hTTInvoice:DEFAULT-BUFFER-HANDLE.

/* Define the data-sources */
DEFINE DATA-SOURCE dCust FOR Customer.
DEFINE DATA-SOURCE dInv FOR Invoice.

/* Attach the data-sources to the dataset buffers */
hBufTTCust:ATTACH-DATA-SOURCE(DATA-SOURCE dCust:HANDLE,?,?,?).
hBufTTInv:ATTACH-DATA-SOURCE(DATA-SOURCE dInv:HANDLE,?,?,?).

/* Fill the dataset using the data-sources */
hDataSet:FILL().

/* customer navigation query */
hTopQuery = hDataSet:TOP-NAV-QUERY(1).
hTopQuery:QUERY-PREPARE("PRESELECT EACH ttCustomer").
hTopQuery:QUERY-OPEN.

/* This makes all of Customer and Invoice fields available for display. */
rBindS = NEW Progress.Data.BindingSource(hDataSet, hBufTTCust, "*", "").

/* Alternatively, specify fields using the optional include-fields and
   except-fields lists. */
/* rBindS = NEW Progress.Data.BindingSource(hDataSet, hBufTTCust, "*",
         "ttCustomer.Address2,ttCustomer.Country,ttInvoice.CustNum"). */

/* Disable editing because procedure does not have event logic for changes. */
rBindS:AllowEdit                    = FALSE.
rBindS:ChildAllowEdit["ttInvoice"]  = FALSE.
rBindS:AllowRemove                  = FALSE.
rBindS:ChildAllowRemove["ttInvoice"] = FALSE.

/* Main block */
```

```
IF VALID-OBJECT(rBindS) THEN
  DO ON ERROR UNDO, LEAVE:

    rMainForm        = NEW Progress.Windows.Form().
    rMainForm:Width  = 700.
    rMainForm:Height = 560.
    rMainForm:Text   = "Customer & Invoice Form".

    rCustInvGrid            = NEW UltraGrid().
    rCustInvGrid:Left       = 10.
    rCustInvGrid:Top        = 10.
    rCustInvGrid:Width      = 660.
    rCustInvGrid:Height     = 420.
    rCustInvGrid:Text       = "Customer & Invoice Grid".
    rCustInvGrid:DataSource = rBindS.

    rCustNumLabel        = NEW Label().
    rCustNumLabel:Left   = 10.
    rCustNumLabel:Top    = 445.
    rCustNumLabel:Width  = 50.
    rCustNumLabel:Height = 15.
    rCustNumLabel:Text   = "Cust #:".

    rCustNumTB          = NEW TextBox().
    rCustNumTB:Left     = 60.
    rCustNumTB:Top      = 440.
    rCustNumTB:Width    = 25.
    rCustNumTB:Height   = 15.
    rCustNumTB:ReadOnly = TRUE.

    rCustNameLabel        = NEW Label().
    rCustNameLabel:Left   = 95.
    rCustNameLabel:Top    = 445.
    rCustNameLabel:Width  = 60.
    rCustNameLabel:Height = 15.
    rCustNameLabel:Text   = "Customer:".

    rCustNameTB          = NEW TextBox().
    rCustNameTB:Left     = 160.
    rCustNameTB:Top      = 440.
    rCustNameTB:Width    = 150.
    rCustNameTB:Height   = 15.
    rCustNameTB:ReadOnly = TRUE.

    rSalesRepLabel        = NEW Label().
    rSalesRepLabel:Left   = 320.
    rSalesRepLabel:Top    = 445.
    rSalesRepLabel:Width  = 60.
    rSalesRepLabel:Height = 15.
    rSalesRepLabel:Text   = "Sales Rep:".

    rSalesRepTB          = NEW TextBox().
    rSalesRepTB:Left     = 390.
    rSalesRepTB:Top      = 440.
    rSalesRepTB:Width    = 100.
    rSalesRepTB:Height   = 15.
    rSalesRepTB:ReadOnly = TRUE.

    rComLabel        = NEW Label().
    rComLabel:Left   = 10.
    rComLabel:Top    = 470.
    rComLabel:Width  = 75.
    rComLabel:Height = 15.
    rComLabel:Text   = "Comments".

    rCommentTB          = NEW TextBox().
    rCommentTB:Left     = 10.
    rCommentTB:Top      = 490.
    rCommentTB:Width    = 660.
```

```
    rCommentTB:Height   = 100.
    rCommentTB:ReadOnly = TRUE.

    controls = rMainForm:Controls.
    controls:Add(rCustInvGrid).
    controls:Add(rCustNumLabel).
    controls:Add(rCustNumTB).
    controls:Add(rCustNameLabel).
    controls:Add(rCustNameTB).
    controls:Add(rSalesRepLabel).
    controls:Add(rSalesRepTB).
    controls:Add(rComLabel).
    controls:Add(rCommentTB).

    /* Data bindings for text boxes. */
    rCustNumTB:DataBindings:Add("Text", rBindS, "CustNum").
    rCustNameTB:DataBindings:Add("Text", rBindS, "Name").
    rSalesRepTB:DataBindings:Add("Text", rBindS, "SalesRep").
    rCommentTB:DataBindings:Add("Text", rBindS, "Comments").

    WAIT-FOR Application:RUN(rMainForm).

    RUN cleanup.

END. /* Main block */

PROCEDURE cleanup:

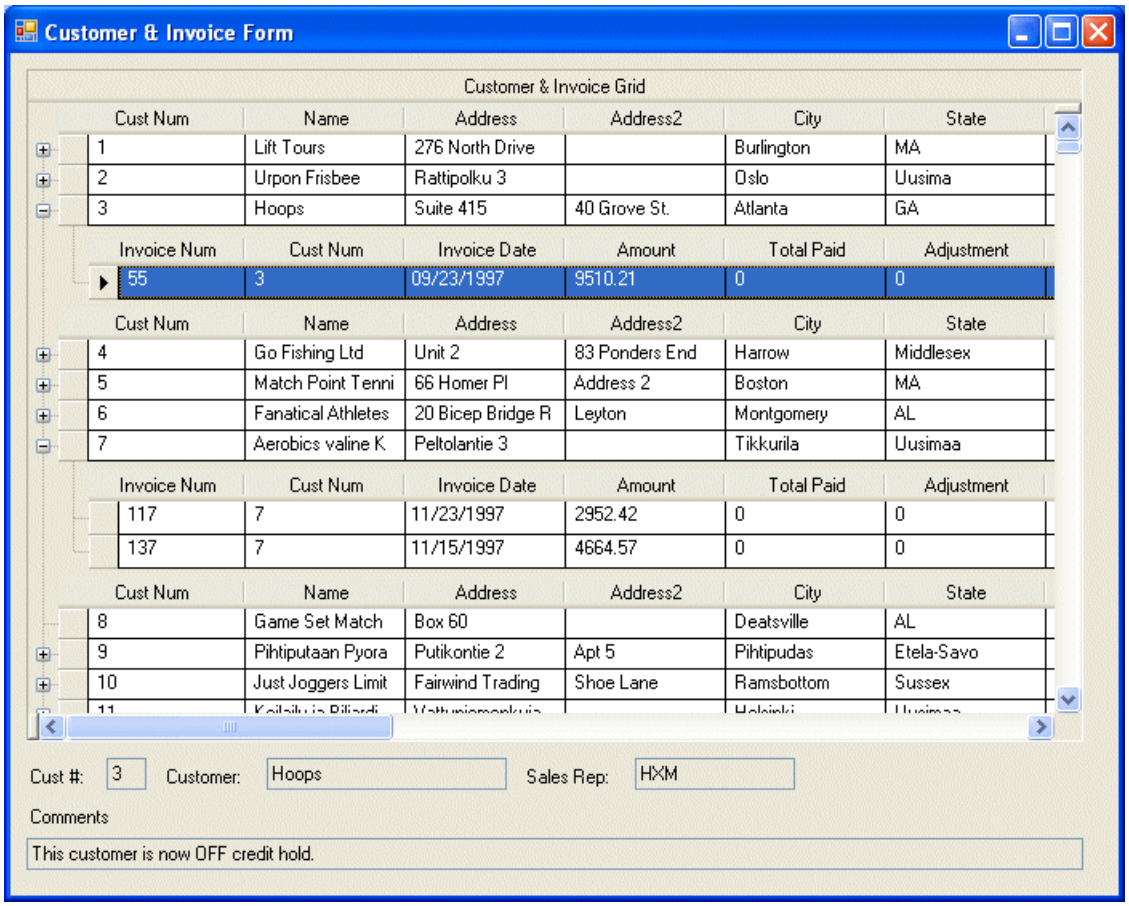
/* Cleanup ProDataSet resources */

  rBindS:Dispose( ).

END PROCEDURE.
```

When this procedure runs, a form appears displaying a hierarchical grid and a simple details viewer, as shown in the following figure.

Figure 13: Grid bound to ProDataSet



As discussed in Binding to ProDataSets on page 138, you might need to display parent and child records in separate grids. `MultipleBindings.p` binds two ProBindingSource objects to the same ProDataSet to display parent and child records in separate grids. The internal procedure, `CustPositionChanged`, synchronizes the grids. The main block calls this procedure in response to the ProBindingSource's PositionChanged event. The procedure then synchronizes and refreshes the child grid.

```
/* MultipleBindings.p
   Bind two BindingSource objects to the same ProDataSet then display the
   parent/child records in separate grids. */

/* USING statements must be the first in the procedure */
USING System.Windows.Forms.*.
USING Infragistics.Win.UltraWinGrid.*.

DEFINE VARIABLE rMainForm   AS Progress.Windows.Form       NO-UNDO.
DEFINE VARIABLE rCustGrid   AS UltraGrid                   NO-UNDO.
DEFINE VARIABLE rInvGrid    AS UltraGrid                   NO-UNDO.
DEFINE VARIABLE rCustBindS  AS Progress.Data.BindingSource NO-UNDO.
DEFINE VARIABLE rInvBindS   AS Progress.Data.BindingSource NO-UNDO.
DEFINE VARIABLE rControls   AS Control+ControlCollection   NO-UNDO.

DEFINE VARIABLE hTTCustomer AS HANDLE                      NO-UNDO.
DEFINE VARIABLE hTTInvoice  AS HANDLE                      NO-UNDO.

DEFINE VARIABLE hDataSet   AS HANDLE NO-UNDO.
DEFINE VARIABLE hBufTTCust AS HANDLE NO-UNDO.
```

```
DEFINE VARIABLE hBufTTInv  AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustQry   AS HANDLE NO-UNDO.
DEFINE VARIABLE hInvQry    AS HANDLE NO-UNDO.

DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE Customer.
DEFINE TEMP-TABLE ttInvoice  NO-UNDO LIKE Invoice.

DEFINE DATASET dsCustInv FOR ttCustomer, ttInvoice
  DATA-RELATION FOR ttCustomer, ttInvoice RELATION-FIELDS(CustNum, CustNum).

hDataSet    = DATASET dsCustInv:HANDLE.

hTTCustomer = TEMP-TABLE ttCustomer:HANDLE.
hTTInvoice  = TEMP-TABLE ttInvoice:HANDLE.

hBufTTCust  = hTTCustomer:DEFAULT-BUFFER-HANDLE.
hBufTTInv   = hTTInvoice:DEFAULT-BUFFER-HANDLE.

/* Define the data-sources */
DEFINE DATA-SOURCE dCust FOR Customer.
DEFINE DATA-SOURCE dInv  FOR Invoice.

/* Attach the data-sources to the dataset buffers */
hBufTTCust:ATTACH-DATA-SOURCE(DATA-SOURCE dCust:HANDLE,?,?,?).
hBufTTInv:ATTACH-DATA-SOURCE(DATA-SOURCE dInv:HANDLE,?,?,?).

/* Fill the dataset using the data-sources */
hDataSet:FILL().

/* Navigation query for Customer */
hCustQry = hDataSet:TOP-NAV-QUERY().
hCustQry:QUERY-PREPARE("PRESELECT EACH ttCustomer").
hCustQry:QUERY-OPEN().
hCustQry:GET-NEXT().

/* Navigation query for Invoice */
hInvQry = hDataSet:GET-TOP-BUFFER():GET-CHILD-RELATION():QUERY.
hInvQry:QUERY-OPEN().

/* Display all Customer fields in the grid. */
rCustBindS = NEW Progress.Data.BindingSource(hCustQry, "*", "").
/* Display all Invoice fields in the grid. */
rInvBindS  = NEW Progress.Data.BindingSource(hInvQry, "*", "").

/* Disable editing because procedure does not include event logic to handle
   changes. */
rCustBindS:AllowEdit   = FALSE.
rCustBindS:AllowRemove = FALSE.
rInvBindS:AllowEdit    = FALSE.
rInvBindS:AllowRemove  = FALSE.

/* Main block */
IF VALID-OBJECT(rCustBindS) AND VALID-OBJECT(rInvBindS) THEN
  DO ON ERROR UNDO, LEAVE:

    rMainForm        = NEW Progress.Windows.Form().
    rMainForm:Width  = 900.
    rMainForm:Height = 520.
    rMainForm:Text   = "Customer & Invoice Form".

    rCustGrid            = NEW UltraGrid().
    rCustGrid:Left       = 10.
    rCustGrid:Top        = 10.
    rCustGrid:Width      = 860.
    rCustGrid:Height     = 250.
    rCustGrid:Name       = "CustGrid".
    rCustGrid:Text       = "Customer Grid".
    rCustGrid:DataSource = rCustBindS.
```

```
    rCustGrid:TabIndex   = 1.

    rInvGrid            = NEW UltraGrid().
    rInvGrid:Left       = 10.
    rInvGrid:Top        = 270.
    rInvGrid:Width      = 860.
    rInvGrid:Height     = 200.
    rInvGrid:Name       = "InvGrid".
    rInvGrid:Text       = "Invoice Grid".
    rInvGrid:DataSource = rInvBindS.
    rInvGrid:TabIndex   = 2.

    rControls = rMainForm:Controls.
    rControls:Add(rCustGrid).
    rControls:Add(rInvGrid).

    rCustBindS:PositionChanged:Subscribe("CustPositionChanged").

    WAIT-FOR Application:RUN(rMainForm).

END. /* Main block */

RUN cleanup.

PROCEDURE CustPositionChanged:
/* When Position changes in CustGrid, synchronize and refresh InvGrid.
   Alternately, you could set the buffer's Auto-Synchronize attribute to
   TRUE.*/

  DEFINE INPUT PARAMETER rSender AS Progress.Data.BindingSource NO-UNDO.
  DEFINE INPUT PARAMETER rArgs   AS System.EventArgs            NO-UNDO.

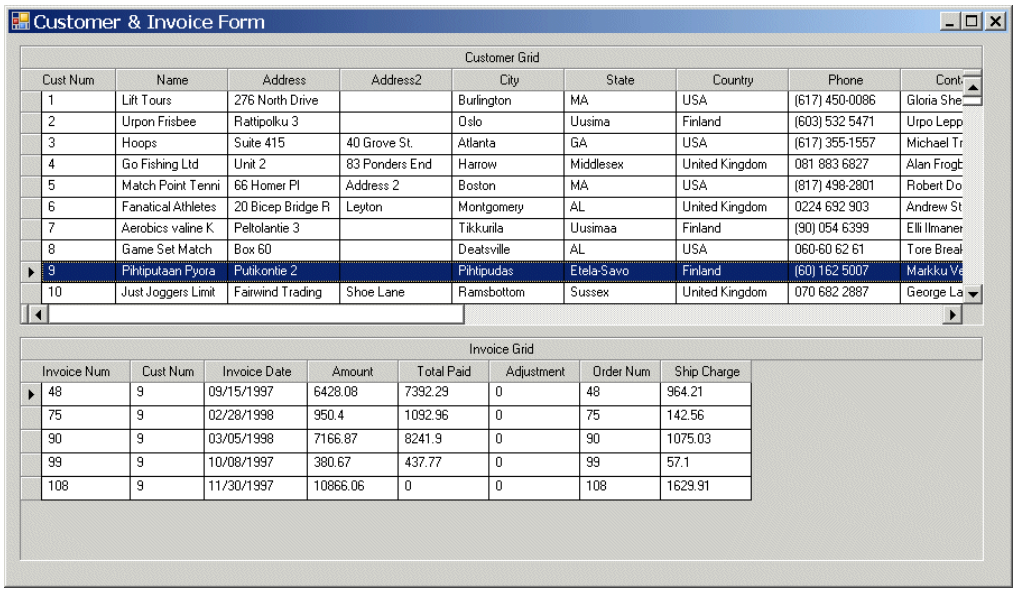
  hBufTTCust:SYNCHRONIZE().
  rInvBindS:RefreshAll().

END PROCEDURE.

PROCEDURE cleanup:
/* Cleanup ProDataSet resources */
  rInvBindS:Dispose( ).
  rCustBindS:Dispose( ).
END PROCEDURE.
```

When this procedure runs, a simple form appears displaying a customer grid and an invoice grid, as shown in the following figure.

Figure 14: Multiple grids bound to same binding source



Programming considerations

The sections that follow discuss other features of the ProBindingSource that you should consider.

1-based or 0-based indexes

In general, .NET objects use 0-based values for parameters and properties. In contrast, ABL uses 1-based indexing. Because of its role as an intermediary between the .NET and ABL objects, the ProBindingSource uses both types of indexes. When an index in the ProBindingSource refers to the .NET side, it is 0-based. When an index refers to the ABL side, it is 1-based. For example, the Position property is 0-based and the InputValue property's *`field-index`* parameter is 1-based.

Row transactions

The ProBindingSource supports a row-based transaction model. Some .NET controls, such as the .NET `System.Windows.Forms.DataGridView` and Infragistics `Infragistics.Win.UltraWinGrid.UltraGrid`, take advantage of this feature to provide the user with Undo functionality. For example, in the `DataGridView`, a user can press **ESC** once to undo a change to the last column changed and then press **ESC** again to undo all changes to the row.

Using the Dispose( ) method

Progress Software Corporation recommends calling the `Dispose( )` method before you delete or release the ProBindingSource instance for garbage collection, especially for ProBindingSource instances bound to ProDataSets, which can use a lot of memory. A ProBindingSource creates a separate query for each expanded row in a hierarchical grid. If you delete the ProBindingSource object but maintain the ProDataSet, those extra queries remain in memory consuming resources until the automatic .NET garbage collection releases it. Using the `Dispose( )` method, you can free the memory for these queries without deleting the associated ProDataSet.

Note: OpenEdge strongly recommends not relying upon the .NET Garbage Collector for cleaning up application objects such as `AppObjects`, `SubAppObjects`, and `ProcObjectsdispose`. The recommended practice is to manually call the `Dispose ()` method on each of these application objects when they are no longer required.

Support for Visual Designer

The ProBindingSource includes the following elements that support the Progress Developer Studio for OpenEdge Visual Designer:

- TableSchema property
- Progress.Data.TableDesc class
- Progress.Data.ColumnPropDesc class
- Progress.Data.DataType enumeration class

The Visual Designer uses these elements to hold a logical schema for the ProBindingSource. You can then design .NET controls based on that schema. This design time schema binding allows you to modify properties of the bound .NET control, such as column widths, column labels, and so forth. At runtime, you must still provide the data source used to populate the data for the specified schema by setting the ProBindingSource `Handle` property. While these elements might be used outside the Visual Designer, you probably will not find them useful.

For more information about choosing a data source at run time, see Special case: unbound class instance on page 140.

Binding .NET controls to BLOB fields

Binding .NET controls to an ABL BLOB (a Binary Large Object) field requires special care. You need to know that the ProBindingSource exposes the BLOB data type to the .NET control as a .NET `System.Byte[]` data type.

You also need to know if the .NET control can render the BLOB's contents appropriately. Check the .NET control's user documentation to see if it can render the type of file that the BLOB represents. A particular .NET control might not be able to render the BLOB at all. In other cases, your application might need to perform some extra step before the .NET control can properly display the BLOB's contents.

For example, consider binding a BLOB that represents an image to a grid. A Microsoft `System.Windows.Forms.DataGridView` automatically represents the BLOB with the `Systems.Windows.Forms.DataGridViewImageColumn` class. If the BLOB is not in a recognized format, the class throws an error and displays a default image in the column's cells.

However, when you bind the same BLOB field to an Infragistics.Win.UltraWinGrid.UltraGrid, the grid represents the BLOB field as an `UltraGridColumn` of the `System.Byte[]` data type. The `UltraGridColumn` cannot directly display the image. You must then add an editor control to the column using something like the following code:

```
myGrid:DisplayLayout:Bands[0]:Columns[1]:Editor =
  NEW Infragistics.Win.EmbeddableImageRender().
```

Binding .NET controls to CLOB fields

Binding .NET controls to an ABL CLOB (a Character Large Object) field requires special care. You need to know that the ProBindingSource exposes the CLOB data type to the .NET control as a .NET `System.String` data type. Before binding a CLOB field to a .NET control, check that it can handle the `System.String` data type.

Working with the NoLOBs property

The `Assign( )` method and CURRENT-CHANGED function cannot check the current values against the initial values of BLOB or CLOB fields. By default, they raise an error if a row contains a BLOB or CLOB field.

The NoLOBs property specifies whether or not the AVM ignores BLOB or CLOB fields while executing the ProBindingSource's `Assign( )` method or the `CURRENT-CHANGED` function. The default value of the NoLOBs property is FALSE.

Using the .NET DataMember property

In general, when you want to display child records in a separate control from the parent record, you create multiple ProBindingSources. You bind one ProBindingSource to the parent table's navigation query and bind a different ProBindingSource to each child's relation query, as shown below:

```
rCustBindS = NEW Progress.Data.BindingSource(dshdl:TOP-NAV-QUERY(1)).
rCustGrid:DataSource = rCustBindS.
rOrderBindS = NEW Progress.Data.BindingSource(dshdl:GET-RELATION(1):QUERY).
rOrderGrid:DataSource = rOrderBindS.
rOlineBindS = NEW Progress.Data.BindingSource(dshdl:GET-RELATION(2):QUERY).
rOlineGrid:DataSource = rOlineBindS.
```

However, you can use a single ProBindingSource for a ProDataSet and still display the child records in a separate control. After binding the ProBindingSource to the ProDataSet, you can set each control's `DataMember` property to point to the appropriate table, as shown below:

```
rCustOrdOLineBindS = NEW Progress.Data.BindingSource(dshdl, "ttCustomer").
rCustGrid:DataSource = rCustOrdOLineBindS.
rOrderGrid:DataSource = rCustOrdOLineBindS.
rOrderGrid:DataMember = "ttOrder".
rOlineGrid:DataSource = rCustOrdOLineBindS.
rOlineGrid:DataMember = "ttOrder.ttOrderLine".
```

In the example above, the ProBindingSource binds to a hierarchical set of tables starting at the top-level table, `ttCustomer`. The Customer grid binds to the ProBindingSource. Because the top-level table in the ProBindingSource is automatically assumed, the code does not need to specify the grid's `DataMember`. However, the code specifies the Order grid's `DataMember` as `ttOrder`, so that grid binds to the orders for the current customer. Similarly, the code specifies the OrderLine grid's `DataMember` as `ttOrder.ttOrderLine`, so that grid binds to the order lines for the current order.

Notice that in order to set the `DataMember` property, you must specify the full hierarchy of child tables. Also, when you are interested in the parent records, you **do not set** the `DataMember` property, since the ProBindingSource already binds directly to the top-level table.

Note: The OpenEdge GUI for .NET uses the `DataMember` property a little differently than other .NET implementations. You can set the `DataMember` property to the top-level table in .NET, but not in ABL.

Similarly, you can use a control's `DataMember` property to bind to a single field within a ProBindingSource bound to a ProDataSet, as shown below:

```
rCustOrdBindS = NEW Progress.Data.BindingSource(dshdl, "ttCustomer").
rComboBox1:DataSource = rCustOrdBindS.
rComboBox1:DataMember = "Order.SalesRep".
```

For a field in the top-level table, you can omit the table name, as shown below:

```
/* specifies "Name" field in ttCustomer table */
rComboBox1:DataMember = "Name".
```

Managing data updates

Your application's .NET UI needs to handle the same data management tasks that every other UI handles. The following sections discuss these tasks.

Enabling and disabling updates

Even if a particular .NET UI control supports adding, deleting, or editing records, your application cannot perform those updates unless the ProBindingSource is enabled for that type of update. The `AllowXxx` and `ChildAllowXxx` properties on the ProBindingSource control what types of updates are allowed. All these properties are set to `TRUE` by default. If you want to disable a certain type of record update, you can set the appropriate property to `FALSE`.

Note: A particular bound UI control might not support a particular ProBindingSource property. By the standard .NET protocol, the bound UI control reads the properties that it supports and adjusts its behavior to match.

For example, suppose you have a hierarchical grid for `Customer` records and `Order` records. The user should be able to delete orders, but not customers. You can achieve the desired result through the ProBindingSource, rather than through the grid itself. The following code disables deleting parent records and enables deleting child records:

```
pbs:AllowRemove = FALSE.
pbs:ChildAllowRemove["ttOrders"] = TRUE.
```

Sorting

Most .NET grid controls support sorting in response to some user action. However, the grids might differ on how the sort is accomplished. For example, the `Infragistics.Win.UltraWinGrid.UltraGrid` does its own sorting if the `DisplayLayout:Override:HeaderClickAction` property is set to `SortSingle` or `SortMulti`. You can override the `UltraGrid` sorting by setting this property to `ExternalSortSingle` or `ExternalSortMulti`, subscribing to the `AfterSortChange` event, and handling the sort there.

By contrast, the Microsoft `System.Windows.Forms.DataGridView` relies on its data source to handle the sorting. When this control asks the ProBindingSource to sort the data, the AutoSort property controls the ProBindingSource's response. If `AutoSort` is `FALSE` (the default), the ProBindingSource fires its SortRequest event. Your application can trap this event and handle the sort. If your application ignores the event, no sort happens. If `AutoSort` is `TRUE`, the ProBindingSource automatically reopens the query with the new sort criteria and does not fire its `SortRequest` event. In that case, any query that can be sorted must have been opened with the QUERY-PREPARE and QUERY-OPEN methods, rather than the OPEN QUERY statement. Otherwise, the AVM generates a run-time error.

Generally, a direct sort with an ABL query provides better performance than a control's built-in sort mechanism. This difference can be very noticeable, especially for a grid control displaying a hierarchy of tables. So for most cases, Progress Software Corporation recommends that you handle the sorting with ABL.

Using the UltraGrid's BeforeSortChange and AfterSortChange events

The `UltraGrid` fires both a `BeforeSortChange` and an `AfterSortChange` event when the grid header is clicked and the `HeaderClickAction` is set to one of the sort options.

The `UltraGrid` default sort processing is asynchronous. A drawback to asynchronous processing is that the cursor does not automatically change to indicate ongoing processing. To provide the end-user with that visual clue, you can switch to synchronous processing and change the cursor by including code like the following in the `BeforeSortChange` event:

```
e:ProcessMode = Infragistics.Win.UltraWinGrid.ProcessMode:Synchronous.
System.Windows.Forms.Cursor:Current =
    System.Windows.Forms.Cursors:WaitCursor.
```

You would then set the cursor back to default in the `AfterSortChange` event as follows:

```
System.Windows.Forms.Cursor:Current = System.Windows.Forms.Cursors:Default.
```

Setting the `HeaderClickAction` to `ExternalSortSingle` or `ExternalSortMulti` tells the `UltraGrid` not to do its own sorting. To give access to the sort specified in the UI, the `UltraGrid` exposes a `SortColumns` collection on the band. Because this is a collection, you can sort on multiple columns when you specify the `ExternalSortMulti` option. This collection contains `UltraGridColumns` that expose a `SortIndicator`, which reflects the direction of the sort in the UI. The `SortColumns` are updated in the event, not before the event. So, you implement the ABL sort in the `AfterSortChange` event to access the `SortColumns` that correspond to the fired event.

The following code snippet shows an example function that returns a sort string from a band using the band's `SortedColumns` collection and the sorted columns' `SortIndicator`. An `AfterSortChange` event can call this function with the band attribute on the passed `BandEventArgs` as input and append the returned string to the query of the grid's data source before reopening the query:

```
METHOD STATIC PUBLIC CHARACTER SortExpression
  (band AS Infragistics.Win.UltraWinGrid.UltraGridBand):
  DEFINE VARIABLE sortColumn AS     Infragistics.Win.UltraWinGrid.UltraGridColumn
NO-UNDO.
  DEFINE VARIABLE i AS INT NO-UNDO.
  DEFINE VARIABLE sortString AS CHAR NO-UNDO.

  /* build a sort string from the band's SortedColumns */
  DO i = 0 TO band:SortedColumns:Count - 1:
    sortColumn = cast(band:SortedColumns[i],
      Infragistics.Win.UltraWinGrid.UltraGridColumn).
    sortString = sortString + " by " + sortColumn:Key.
    IF Progress.Util.EnumHelper:AreEqual(sortColumn:SortIndicator,
      Infragistics.Win.UltraWinGrid.SortIndicator:DESCENDING) THEN
      sortString = sortString + " descending".
  END.
  RETURN left-trim(sortString).
END METHOD.
```

Sorting child tables for a recursive data-relation

If a grid control is bound to a ProDataSet with a recursive data-relation, it presents an additional level of complexity. When the recursive data-relation is defined in the ProDataSet, it has a single buffer for the child table. For example:

```
DEFINE DATASET dsetMyrecurs FOR ttEmp
  DATA-RELATION drel1 FOR ttEmp, ttEmp
  RELATION-FIELDS(Emp-name, Manager) RECURSIVE.
```

But because the data-relation is defined recursively, this maps to multiple bands in the grid's hierarchical control. A *band* consists of all the records at a given level in the hierarchical display. Internally, the AVM needs to create a separate buffer for each band in the control, each with a unique buffer name. For example, if you have defined the following recursive data-relation with child table `ttemp`, the AVM will create a new buffer object for each band in the hierarchy naming them `ttemp1`, `ttemp2`, `ttemp3`, and so on.

You need to be aware of this if you are subscribing to the grid's sort event and writing your own sorting code in ABL. The band's key name from the grid control will be the unique buffer name, the one corresponding to the band that the sort request was made from. When using this to set a new value for the data-relation `WHERE-STRING` attribute in the sort event handler, you must map it to the original child table's buffer name. For example, using an UltraGrid control:

```
USING Infragistics.Win.UltraWinGrid.* FROM ASSEMBLY.

PROCEDURE ultraGrid1_AfterSortChange:
  /* Progress.Data.BindingSource */
DEFINE INPUT PARAMETER sender AS System.Object.
DEFINE INPUT PARAMETER eventArgs AS BandEventArgs.

  DEFINE VARIABLE sortColumn AS UltraGridColumn NO-UNDO.
  DEFINE VARIABLE sortString AS CHARACTER NO-UNDO.
  DEFINE VARIABLE cBandKey AS CHARACTER NO-UNDO.

  /* cOriginalWhereString to be set from initial drelHdl:WHERE-STRING */
  DEFINE VARIABLE cOriginalWhereString AS CHARACTER NO-UNDO.
  DEFINE VARIABLE drelHdl AS HANDLE NO-UNDO.

  sortColumn = CAST(eventArgs:Band:SortedColumns[0], UltraGridColumn).
  IF sortColumn:Band:Key BEGINS "ttemp" THEN
    ASSIGN cBandKey = "ttemp".
  ELSE
    ASSIGN cBandKey = sortColumn:Band:Key.

  sortString =  " BY " + cBandKey + "." + sortColumn:Key.

  IF EnumHelper:AreEqual
       (sortColumn:SortIndicator, SortIndicator:Descending) THEN
    sortString = sortString + " DESCENDING".

  ASSIGN
    drelHdl = DATA-RELATION drel1:HANDLE /* From the dsetMyrecurs example */
    drelHdl:WHERE-STRING = cOriginalWhereString + " " + sortString.

  CAST(sender, Progress.Data.BindingSource):RefreshAll( ).
END.
```

Note: For more information on using bands to access the child queries of recursive data-relations, see Finding the proper child query in a ProDataSet on page 166.

Using the SortRequest event

.NET UI controls that rely on their data source for sorting, like the `System.Windows.Forms.DataGridView`, publish the `SortRequest` event. The ProBindingSource publishes the SortRequest event when a user action causes the bound UI control to request a sort from the ProBindingSource. This event uses an OpenEdge built-in class, Progress.Data.SortRequestEventArgs, that extends the .NET `System.EventArgs` class to pass the event arguments. The event argument class contains the FieldIndex, FieldName, ArrayIndex, Ascending, and Sorted properties.

Note: Unlike the `DataGridView`, the `UltraGrid` does not publish this event.

An event handler for this event uses the following syntax:

Syntax

```
EventHandlerName
  (
     INPUT sender AS CLASS System.Object,
     INPUT args AS CLASS Progress.Data.SortRequestEventArgs
  ).
```

Where *EventHandlerName* is the name of the event handler, *sender* is the object reference to the ProBindingSource, and *args* is the object reference to the SortRequestEventArgs instance that contains the event arguments.

If the bound ABL data source object is a ProDataSet, this event always refers to the top-level query. If your application needs to sort the child records, one approach is to create two ProBindingSource objects, one for the parent and one for the child, and to display the child records for a specific parent record in another UI control.

When this event occurs, the event handler must use a modified sort criteria based on the values in the SortRequestEventArgs object to reopen the query associated with the ABL data source object. The `FieldIndex`, `FieldName`, and `ArrayIndex` properties indicate the field on which to sort the query. The `Ascending` property indicates the order of the sort, with the TRUE meaning an ascending sort and `FALSE` meaning a descending sort. The event handler should set the `Sorted` property to `TRUE` when the query reopens successfully with the new sort criteria.

The `FieldIndex`, `FieldName`, and `ArrayIndex` properties combine to indicate the new sort field. `FieldName` is the unqualified field name as specified in the ProBindingSource. Even if you had to qualify the field name in the constructor, the `FieldName` is unqualified in the SortRequestEventArgs class. `FieldIndex` is a 1-based index of the ProBindingSource fields in the order that they appear in the constructor's included field list. If the specified field is an array field, the `ArrayIndex` property indicates the proper element in the array with a 1-based index of the array elements.

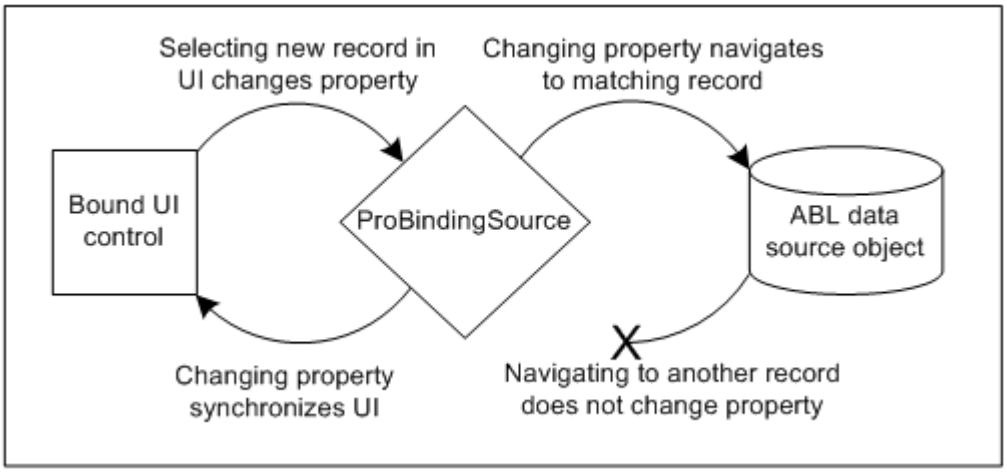
Maintaining currency with the Position property

The Position property's value indicates the currently selected record in the bound UI control. Whenever its value changes, the ProBindingSource automatically synchronizes the buffer in the ABL data source object to the corresponding record. In this way, the application always has the correct record available when an event procedure runs.

Note: When the bound ABL data source object is a ProDataSet, this property always refers to the top-level query.

As shown in the following figure, the currency control works in both directions between the bound UI control and the ProBindingSource. If you select a different record in the UI, the bound UI control updates the property value in the ProBindingSource. If you programmatically change the property value in the ProBindingSource, the ProBindingSource synchronizes the bound UI control to display the newly selected record.

Figure 15: Currency control with Position property



However, the currency control only works going from the ProBindingSource to the ABL data source object. If your application directly navigates to another record in the ABL data source object, the `Position` property's value does not update. Generally, rather than letting the application change the record, you should change the property value and let the ProBindingSource handle navigating to the new record and updating the bound UI control.

If your application must change the record directly, it also needs to synchronize the bound UI control to the new record. You can achieve this synchronization using the REPOSITION statement and REPOSITION-Xxx( ) methods.

Using any of the Reposition instructions changes the ProBindingSource's `Position` property to the specified record. These instructions move the cursor to before the specified record in the result list. When you have an ABL Browse widget linked to a query, the Browse automatically does a GET NEXT to get the specified record into the buffer. So, using any of the Reposition instructions automatically results in the Browse positioning to the specified record. The ProBindingSource is designed to match this behavior.

Synchronizing data

When your application has updated the bound ABL data source object or another user has altered a record, your application needs to push those changes out to the bound UI control. Setting the ProBindingSource's AutoSync property to `TRUE` enables you to synchronize the data using certain ABL statements and methods that reopen or reposition a query. The synchronization is similar to what happens with a Browse widget.

If the `AutoSync` property is `FALSE`, you can use the ProBindingSource's Refresh methods to synchronize bound UI controls with the bound ABL data source object.

Reopening queries

If the `AutoSync` property is `TRUE`, using the OPEN QUERY statement or the QUERY-OPEN( ) method to reopen the query for a bound ABL data source object automatically synchronizes any bound UI control. Each control's design determines how that control responds to this synchronization. For example, synchronizing the screen values in a grid does not usually reset the current position of the cursor which would produce a change in the ProBindingSource's `Position` property.

Note: This automatic synchronization does **not** occur if the query is reopened within the context of an OffEnd event handler.

When you bind to a ProDataSet, the ProDataSet buffer's AUTO-SYNCHRONIZE attribute affects the synchronization behavior. By default, this attribute is `FALSE`. So, the AVM does not automatically synchronize all the ProDataSet's relation queries when one of them is positioned to a different row. Automatic synchronization might incur unnecessary overhead and slow your application.

However, if the attribute is `TRUE` for a specific buffer, the AVM automatically reopens any child query when your application navigates to a new parent record in a ProDataSet. When the ProBindingSource's `AutoSync` property is `TRUE`, this behavior leads to the automatic synchronization of any UI control bound to the child query. For more information about how to synchronize ProDataSets, see the chapter on attributes and methods in OpenEdge Development: ProDataSets.

If the ProDataSet buffer's `AUTO-SYNCHRONIZE` attribute is `FALSE`, you can synchronize the data by capturing the ProBindingSource's PositionChanged event and calling the ProDataSet's SYNCHRONIZE( ) method.

Caution: Your application should not reopen a query while a user is editing a record. This might cause UI errors or other incorrect results.

Manipulating result list entries

When you have an ABL Browse widget linked to a query, the CREATE-RESULT-LIST-ENTRY( ) and DELETE-RESULT-LIST-ENTRY( ) methods do synchronize the Browse's screen values. However, these methods do not cause any synchronization when called on a query bound to a ProBindingSource. If a refresh is necessary in conjunction with these methods, your application needs to call the ProBindingSource's RefreshAll( ) method or reopen the query.

Using the Refresh methods

The Refresh methods provide a way to ensure that changes to the records in the underlying ABL data source object are immediately propagated to the bound UI control. If you do not specify the optional *`record-index`* parameter, the `Refresh( )` method acts on the current record, indicated by the Position property, of the top-level query. The optional parameter enables you to specify a different record in the top-level query.

Note: Because it is the index into the result list, the optional parameter is a 1-based index. For further information, see 1-based or 0-based indexes on page 156.

The `RefreshAll( )` method acts on all records in the ABL data source object. The current values in the ABL data source object are pushed out to all bound UI controls. This method also refreshes values in the ProBindingSource's child tables. The `Refresh( )` method does not work on child buffers.

The Refresh methods do not change the state of any ongoing transactions. If your application calls to reset the current screen values while a user is editing a record, the Update transaction is still open, the bound UI control is still in edit mode, and the user can redo his edits. For UI controls with built-in Undo functionality, the Refresh methods do not interfere with that functionality.

Handling user interface events

The sections that follow discuss how to employ the ProBindingSource's properties, methods, and events to handle UI events.

Checking for updates

The RowModified property provides a means for checking whether the current record is currently being edited. This property is useful when responding to validation events that can fire even if no data has changed.

For example, the Microsoft `System.Windows.Forms.DataGridView` always fires the `RowValidating` event when the current row changes, whether or not the data was modified. When your application only needs to run validation on changed data, it could trap this event and then use the value of `RowModified` to determine whether to run the validation.

Writing screen values to the data source object

The Assign( ) method provides a convenient way to assign the current screen values for a specified record in a bound UI control back to the corresponding record in the bound ABL data source object. This method delivers all the screen values as a unit. If the bound UI control is a primitive value, that is a variable or field, only that value is updated. However, if the bound UI control displays rows, the entire row's screen values are updated in the ABL data source object. The normal use for this method in multi-field control, like a grid, is with full rows. You cannot use this method to assign the changes for a single field in a multi-field control, because those values are not available until after any field- or cell-leaving events have fired.

If the assignment fails (for example, due to a validation error), `Assign( )` returns `FALSE`. Otherwise, it returns `TRUE`. This method's results do not indicate whether or not the screen values changed. The results only indicate whether the assignment succeeded or failed.

Finding the proper child query in a ProDataSet

When a ProDataSet with data-relations binds with a ProBindingSource, the ProBindingSource creates and maintains a new child query for every expanded parent record in a hierarchical grid. So, for each child table, you are likely to have not a single query, but several different queries. The data-relation's `CURRENT-QUERY( )` method provides a way to find the child query that corresponds to the currently selected parent record. This attribute is useful when writing code to create, modify, or delete child records.

For example, take the GetCurrentQuery function, shown here:

```
FUNCTION GetCurrentQuery RETURNS HANDLE (INPUT cBufferName AS CHARACTER).
  DEFINE VAR hDataSet      AS HANDLE.
  DEFINE VAR hDataRelation AS HANDLE.
  DEFINE VAR hQuery        AS HANDLE.

  hDataSet = DATASET dsCustOrdOrdlines:HANDLE.
  IF cBufferName EQ "ttCustomer" THEN DO:
    hQuery = hTopQuery.
  END.

  ELSE IF cBufferName EQ "ttOrder" THEN DO:
    hDataRelation = hDataSet:GET-RELATION(1).
    hQuery = hDataRelation:CURRENT-QUERY().
  END.

  ELSE IF cBufferName EQ "ttOrderLine" THEN DO:
    hDataRelation = hDataSet:GET-RELATION(2).
    hQuery = hDataRelation:CURRENT-QUERY().
  END.

  ELSE
    hQuery = ?.

  RETURN hQuery.

END FUNCTION.
```

You pass in a buffer name and the function matches the name to one of the tables in the dsCustOrdOrdlines ProDataSet. If the buffer is one of the child tables, the function uses the CURRENT-QUERY( ) method to find the appropriate query and passes it back. For an example of this technique, see UpdatableDataBindingGrid.p (Part 4 of 11) in the Internal procedures and functions on page 175.

If the ProDataSet has a recursive data-relation, it presents an additional level of complexity. Because of the recursive data-relation, a single buffer name might be associated with several different bands in a hierarchical control. A *band* consists of all the records at a given level in the hierarchical display. This makes the buffer name insufficient to access the correct query.

To handle recursive data-relations, you must specify the optional BandIndex parameter when using the CURRENT-QUERY( ) method. The BandIndex property indicates which level of the hierarchical display contains the currently selected record. Using the BandIndex, the CURRENT-QUERY( ) method can determine which query corresponds to the focused row in that band.

Note: The BandIndex is a 0-based index.

The following procedure is a `CreateRow` event handler designed to handle a ProDataSet with a recursive data-relation. First, it checks to see if the `BandIndex` is 0, which always uses the top query. If not, it uses the `BandIndex` as the parameter for the `CURRENT-QUERY( )` method to find the handle of the query that corresponds to the focused row in that band.

```
PROCEDURE recursiveRelationCreateRow:

DEFINE INPUT PARAMETER sender AS System.Object.
DEFINE INPUT PARAMETER args   AS Progress.Data.CreateRowEventArgs.

DEFINE VARIABLE hBuffer AS HANDLE.
DEFINE VARIABLE hQuery  AS HANDLE.

hBuffer = args:BufferHdl.
IF args:BandIndex EQ 0 THEN
  hQuery = hTopQuery.
ELSE
  hQuery = hRelation:CURRENT-QUERY(args:BandIndex).

hBuffer:BUFFER-CREATE().
hQuery:CREATE-RESULT-LIST-ENTRY().
args:Created = TRUE.

END.
```

Using the CreateRow event

The ProBindingSource publishes the CreateRow event when the user requests a new record through a bound UI control. This event uses an OpenEdge built-in class, Progress.Data.CreateRowEventArgs, that extends the .NET `System.EventArgs` class to pass the event arguments. The event argument class uses the BufferHdl and BufferName properties to pass handle and name of the buffer for the appropriate table in the bound ProDataSet. If the bound ABL data source object is a query, rather than a ProDataSet, these properties contain the `Unknown value (?)`.

An event handler for this event uses the following syntax:

Syntax

```
EventHandlerName
  (
    INPUT sender AS CLASS System.Object,
    INPUT args   AS CLASS Progress.Data.CreateRowEventArgs
  ).
```

Where *`EventHandlerName`* is the name of the event handler, *`sender`* is the object reference to the ProBindingSource, and *`args`* is the object reference to the `CreateRowEventArgs` instance that contains the event arguments.

Subscribe to this event when your application needs to create the new record. For example, your application might need to calculate certain field values before the user can access the record. The event handler then needs to create a record in the bound ABL data source object and a corresponding record in the result set. This keeps the data in the bound UI control synchronized with the ABL data source object.

If the record was successfully created, the event handler should set the Created property for the `CreateRowEventArgs` object to `TRUE` (the default value). If the record was not successfully created, set the `Created` property to `FALSE`.

Some .NET UI controls publish their own events when a user requests a new record. However, Progress Software Corporation recommends that your application subscribe to the CreateRow event in all cases to handle these events in a consistent manner. In particular, you must use this event to create rows in a child table in a ProDataSet because the CURRENT-QUERY( ) method on the child table's `DATA-RELATION` is not necessarily correct when the bound UI control fires its event. Therefore, your application cannot call CREATE-RESULT-LIST-ENTRY for the current child query. Because the ProBindingSource creates and maintains a separate child query for each expanded parent row, using the `CreateRow` event guarantees that the `CURRENT-QUERY( )` method is correct.

Caution: The event handler should not reopen the query or call the Refresh( ) method after creating the record.

Using the CancelCreateRow event

The ProBindingSource publishes the CancelCreateRow event when the user cancels adding a new record through a bound UI control. This event uses an OpenEdge built-in class, Progress.Data.CancelCreateRowEventArgs, that extends the .NET `System.EventArgs` class to pass the event arguments. The event argument class uses the BufferHdl and BufferName properties to pass the handle and name of the buffer for the appropriate table in the bound ProDataSet. If the bound ABL data source object is a query, rather than a ProDataSet, these properties contain the `Unknown value (?)`.

An event handler for this event uses the following syntax:

Syntax

```
EventHandlerName
  (
    INPUT sender AS CLASS System.Object,
    INPUT args   AS CLASS Progress.Data.CancelCreateRowEventArgs
  ).
```

Where *`EventHandlerName`* is the name of the event handler, *`sender`* is the object reference to the ProBindingSource, and *`args`* is the object reference to the `CancelCreateRowEventArgs` instance that contains the event arguments.

The event handler then needs to delete both the previously created row and the corresponding record in the result set. This keeps the data in the bound UI control synchronized with the ABL data source object.

Using the OffEnd event

The ProBindingSource publishes the OffEnd event when record batching is enabled and the bound UI control reaches the last row of the current result set. This event uses an OpenEdge built-in class, Progress.Data.OffEndEventArgs, that extends the .NET `System.EventArgs` class to pass the event arguments. The event argument class contains the RowsAdded property, which passes to the ProBindingSource the number of records that you just added to the result set.

An event handler for this event uses the following syntax:

Syntax

```
EventHandlerName
  (
    INPUT sender AS CLASS System.Object,
    INPUT args   AS CLASS Progress.Data.OffEndEventArgs
  ).
```

Where *EventHandlerName* is the name of the event handler, *sender* is the object reference to the ProBindingSource, and *args* is the object reference to the OffEndEventArgs instance that contains the event arguments.

If the Batching property is TRUE, this event fires when a bound control asks for column values for the last record of the query on the client. For example, the event fires when the user encounters the last record while scrolling, or if the application gets the last record.

When your event handler receives this event, it should request the next batch of records from the AppServer and add them to the result set. Before returning, the event handler must set the RowsAdded property of the OffEndEventArgs object parameter to the number of added rows. Once the application has retrieved all the records from the database, the event handler should set the Batching property to FALSE. Setting Batching to FALSE prevents this event from firing again.

There is a special case involving the **End** key and grids. If you enable the **End** key to scroll the grid, you cannot have a new row at the bottom of the grid when the Batching property is TRUE. In this case, the **End** key applies focus to the empty new row and the OffEnd event is not called.

The Microsoft DataGridView always displays the new row at the bottom of the grid. You should never enable both the new row and the **End** key with the DataGridView.

The UltraGrid has an enumerator whose members control the placement of the new row. You can work around this special case by moving the new row to the top of the grid. For example, the following code displays the new row at the bottom of the grid:

```
pbsGrid:DisplayLayout:Override:AllowAddNew =
    Infragistics.Win.UltraWinGrid.AllowAddNew:FixedAddRowOnBottom.
```

By using the FixedAddRowOnTop member instead, the new row displays at the top of the grid:

```
pbsGrid:DisplayLayout:Override:AllowAddNew =
    Infragistics.Win.UltraWinGrid.AllowAddNew:FixedAddRowOnTop.
```

Note that the ProBindingSource only supports batching for the top-level query. When the ProBindingSource is bound to a ProDataSet, the event handler retrieves all child records for a specific parent record. In cases where you have large numbers of child records for each parent, you might need to create another ProBindingSource with its top-level query set to the child table and bind it to a different UI control to display the child records. You can then apply batching to the child records independent of the parent records.

Rather than calling CREATE-RESULT-LIST-ENTRY each time your application adds a batch of records to the result set, your application could reopen the query. However, reopening the query while handling this event does not prompt an automatic refresh.

Using the PositionChanged event

The ProBindingSource publishes the PositionChanged event when the value of the Position property changes, either programmatically or in response to a user action. This event passes the standard .NET `System.EventArgs` class.

An event handler for this event uses the following syntax:

Syntax

```
EventHandlerName
  (
    INPUT sender AS CLASS System.Object,
    INPUT args   AS CLASS System.EventArgs
  ).
```

Where *`EventHandlerName`* is the name of the event handler, *`sender`* is the object reference to the ProBindingSource, and *`args`* is the object reference to the `System.EventArgs` instance that contains the event arguments.

Your event handler can access the ProBindingSource's `Position` property using the object reference in the `System.Object` instance. As discussed in Maintaining currency with the Position property on page 163, when the bound ABL data source object is a ProDataSet, the `Position` property always refers to the top-level query. When the user selects a child record, the `Position` property is set to the position of the corresponding parent record. If your application needs to know the position of the child record, one approach is to create two ProBindingSource objects, one for the parent and one for the child, and to display the child records for a specific parent record in another UI control.

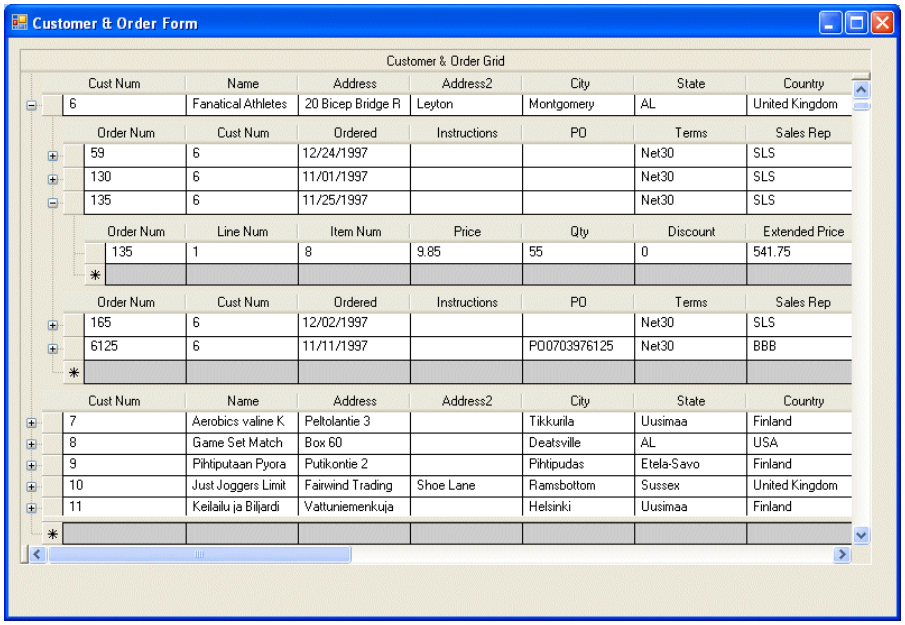
When the value of the `Position` property changes, the ProBindingSource object automatically synchronizes the buffer in the bound ABL data source object to correspond to the selected record in the bound UI control.

Note that if your application directly changes the record in the ABL data source object, the `Position` property's value does not update. Generally, rather than letting the application change the record, you should change the property value and use the event handler for `PositionChanged` to handle any necessary processing. For an example of this technique, see `MultipleBindings.p` in ProDataSet binding example on page 149.

Example of an updatable grid

When run, `UpdatableDataBindingGrid.p` creates a `Infragistics.Win.UltraWinGrid.UltraGrid` that displays hierarchical data from a ProDataSet on the `Customer`, `Order`, and `Orderline` tables, as shown in the following figure.

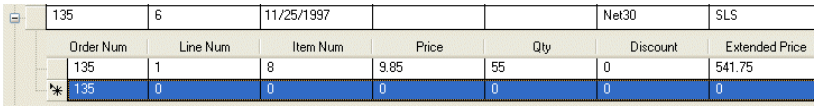
Figure 16: Updatable grid



The application's event logic can handle adding, deleting, and editing records. For information on using the sample applications for this manual, see Example procedures.

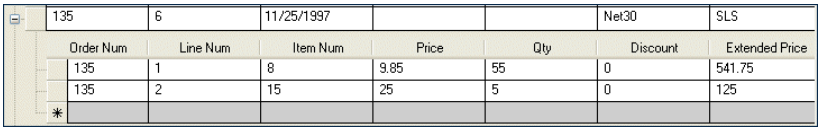
To test the event logic:

1. Run `UpdatableDataBindingGrid.p` with the `Sports2000` database attached.
2. Select **CustNum 6** and expand the child grids to display an order and its orderlines.
3. Select the gray row at the bottom of the **Orderlines** grid. The default values for the fields are filled in automatically, as shown:

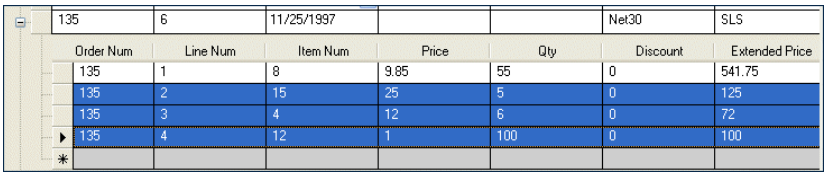


   Obviously, in a business application, the event logic would perform some additional steps, such as incrementing the **Line Num** field.

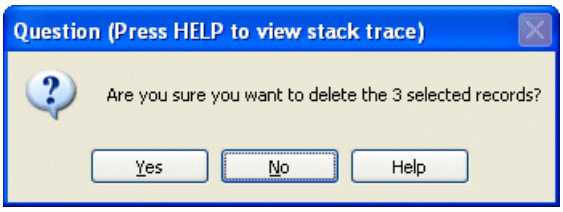
4. Enter data for the new orderline and select another row in the grid. The procedure stores the new record, adds a new row to the child query, and refreshes the grid to display the new orderline:



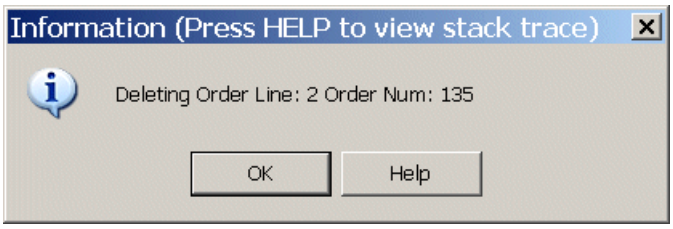
5. Add several more orderlines:



6. Select your orderlines and press **DELETE**. A dialog appears to confirm the deletion:



7. Click **Yes** and a message box like the following appears confirming the deletion of each orderline:



The sections that follow examine the inner workings of this procedure.

Definitions

Now that you have seen the procedure in action, take a look at the code. As shown here, the procedure starts with the expected definitions for the UI controls, the ProBindingSource, the ABL data source object, and the function prototypes.

```
/* UpdatableDataBindingGrid.p
   Bind to a ProDataSet for the Customer, Order, and Orderline tables. Display
the records hierarchically in an Infragistics UltraWinGrid. Add event logic
for adding, updating, and deleting. */

/* USING statements must be the first in the procedure. Note that you could
   have USING statements for the OpenEdge classes also.*/
USING System.Windows.Forms.*.
USING Infragistics.Win.UltraWinGrid.*.

DEFINE VARIABLE rMainForm   AS Progress.Windows.Form       NO-UNDO.
DEFINE VARIABLE rUpdateGrid AS UltraGrid                   NO-UNDO.
DEFINE VARIABLE rBindS      AS Progress.Data.BindingSource NO-UNDO.
DEFINE VARIABLE rControls   AS Control+ControlCollection   NO-UNDO.

DEFINE VARIABLE hTTCustomer  AS HANDLE                     NO-UNDO.
DEFINE VARIABLE hTTOrder     AS HANDLE                     NO-UNDO.
DEFINE VARIABLE hTTOrderLine AS HANDLE                     NO-UNDO.

DEFINE TEMP-TABLE ttCustomer NO-UNDO
    LIKE Customer  BEFORE-TABLE ttCustomerB.
DEFINE TEMP-TABLE ttOrder NO-UNDO
    LIKE Order     BEFORE-TABLE ttOrderB.
DEFINE TEMP-TABLE ttOrderLine NO-UNDO
    LIKE OrderLine BEFORE-TABLE ttOrderLineB.
```

```
DEFINE DATASET dsCustOrder FOR ttCustomer, ttOrder, ttOrderLine
  DATA-RELATION FOR ttCustomer, ttOrder
    RELATION-FIELDS(CustNum, CustNum)
  DATA-RELATION FOR ttOrder, ttOrderLine
    RELATION-FIELDS(OrderNum, OrderNum).

/* Define the data-sources with their queries */
DEFINE DATA-SOURCE dCust    FOR Customer.
DEFINE DATA-SOURCE dOrder   FOR Order.
DEFINE DATA-SOURCE dOrdLine FOR OrderLine.

FUNCTION GetCurrentQuery   RETURNS HANDLE
  (INPUT cBufferName AS CHARACTER) FORWARD.
FUNCTION DeleteRow         RETURNS INTEGER (INPUT cBufferName AS CHARACTER,
  INPUT hQuery AS HANDLE, INPUT rCurrentRow AS UltraGridRow) FORWARD.
FUNCTION CreateRow         RETURNS INTEGER
  (INPUT cBufferName AS CHARACTER) FORWARD.
FUNCTION ProcessRowChanges RETURNS LOGICAL
  (INPUT cBufferName AS CHARACTER) FORWARD.
```

The next section sets up the ProBindingSource, `rBindS`, the ABL data source object, `dsCustOrder`, and their navigation.

Note: For convenience, this example relies on the Sports2000 database trigger to prevent the user from deleting a `Customer` record that still has associated `Order` records.

```
/* Attach the data-sources */
BUFFER ttcustomer:ATTACH-DATA-SOURCE(DATA-SOURCE dCust:HANDLE,?,?,?).
BUFFER ttorder:ATTACH-DATA-SOURCE(DATA-SOURCE dOrder:HANDLE,?,?,?).
BUFFER ttorderline:ATTACH-DATA-SOURCE(DATA-SOURCE dOrdLine:HANDLE,?,?,?).

/* Fill the dataset from the data-sources */
DATASET dsCustOrder:FILL().
hTTCustomer  = TEMP-TABLE ttCustomer:HANDLE.
hTTOrder     = TEMP-TABLE ttOrder:HANDLE.
hTTOrderLine = TEMP-TABLE ttOrderLine:HANDLE.

hTTCustomer:TRACKING-CHANGES  = YES.
hTTOrder:TRACKING-CHANGES     = YES.
hTTOrderLine:TRACKING-CHANGES = YES.

/* Create the binding source and its navigation. */
DEFINE VARIABLE hTopQuery AS HANDLE NO-UNDO.
DEFINE VARIABLE hDataSet  AS HANDLE NO-UNDO.
DEFINE VARIABLE hTT       AS HANDLE NO-UNDO.

hDataSet  = DATASET dsCustOrder:HANDLE.

hTopQuery = hDataSet:TOP-NAV-QUERY(1). /* navigation query for customer */
hTopQuery:QUERY-PREPARE("PRESELECT EACH ttCustomer").
hTopQuery:QUERY-OPEN.

hTT = BUFFER ttCustomer:HANDLE.

/* Create the binding source, excluding the ShipDate, PromiseDate, and Carrier
   fields. */
rBindS = NEW Progress.Data.BindingSource(hDataSet, hTT, "*",
  "ttOrder.ShipDate,ttOrder.PromiseDate,ttOrder.Carrier").
```

Main block

In the main block, after instantiating and configuring the UI controls, the procedure enables each level to add new records using the `UltraWinGrid` enumeration, `AllowAddNew`. The procedure then subscribes to events from the ProBindingSource and the grid.

```
/* Main block */
IF VALID-OBJECT(rBindS) THEN
  DO ON ERROR UNDO, LEAVE:
    /* Create form */
    rMainForm        = NEW Progress.Windows.Form().
    rMainForm:Width  = 800.
    rMainForm:Height = 550.
    rMainForm:Text   = "Customer & Order Form".

    /* Create grid */
    rUpdateGrid            = NEW UltraGrid().
    rUpdateGrid:Left       = 10.
    rUpdateGrid:Top        = 10.
    rUpdateGrid:Width      = 760.
    rUpdateGrid:Height     = 460.
    rUpdateGrid:Text       = "Customer & Order Grid".
    rUpdateGrid:DataSource = rBindS.

    /* Add controls to form. */
    rControls = rMainForm:Controls.
    rControls:Add(rUpdateGrid).

    /* Add new record row at the bottom of each level in the grid. AllowAddNew
       is an UltraWinGrid ennumeration. */
    rUpdateGrid:DisplayLayout:Override:AllowAddNew =
      AllowAddNew:FixedAddRowOnBottom.

    /* Subscribe to binding source events */
    rBindS:CreateRow:Subscribe("BindSCreateRow").
    rBindS:CancelCreateRow:Subscribe("BindSCancelCreateRow").

    /* Subscribe to grid control events */
    rUpdateGrid:BeforeRowUpdate:Subscribe("gridBeforeRowUpdate").
    rUpdateGrid:BeforeRowsDeleted:Subscribe("gridBeforeRowsDeleted").

    WAIT-FOR Application:Run(rMainForm).
END.
RUN cleanup.
```

Internal procedures and functions

The CreateRow event handler, `BindSCreateRow`, gets the BufferName from the CreateRowEventArgs class instance. Before attempting to create the new record, the handler ensures that the Created property is `FALSE`. The handler calls the `CreateRow` function, which returns zero if the create operation fails. If the create operation succeeds, the handler adds the new record to the result list for the appropriate query in the ProBindingSource. Finally, the handler sets the `Created` property to `TRUE` to signal the creation succeeded.

```
/* Internal Procedures */

PROCEDURE cleanup:

/* Cleanup ProDataSet resources */
```

```
  rBindS:Dispose( ).

END PROCEDURE.

/* BindingSource event procedures */

PROCEDURE BindSCreateRow:
  DEFINE INPUT PARAMETER rSender AS System.Object                    NO-UNDO.
  DEFINE INPUT PARAMETER rArgs   AS Progress.Data.CreateRowEventArgs NO-UNDO.

  DEFINE VARIABLE hQuery AS HANDLE NO-UNDO.

  /* Set Created to FALSE until we are sure the record has been created. */
  rArgs:Created = FALSE.

  CreateBlock:
  DO ON ERROR UNDO, LEAVE:
    IF CreateRow(INPUT rArgs:BufferName) > 0 THEN
      LEAVE CreateBlock.
    hQuery = GetCurrentQuery(INPUT rArgs:BufferName).
    hQuery:CREATE-RESULT-LIST-ENTRY().
    rArgs:Created = TRUE.
  END.

END PROCEDURE.
```

The CancelCreateRow event handler, `BindSCancelCreateRow`, gets the BufferName from the CancelCreateRowEventArgs class instance. The handler finds the correct query in the ProBindingSource with the `GetCurrentQuery` function. Then, the handler deletes the record from the appropriate temp-table, raising a warning message if the record cannot be deleted.

```
PROCEDURE BindSCancelCreateRow:
  DEFINE INPUT PARAMETER rSender AS System.Object NO-UNDO.
  DEFINE INPUT PARAMETER rArgs
    AS Progress.Data.CancelCreateRowEventArgs     NO-UNDO.

  DEFINE VARIABLE cBufferName AS CHARACTER        NO-UNDO.
  DEFINE VARIABLE hQuery      AS HANDLE           NO-UNDO.

  cBufferName = rArgs:BufferName.

  DO TRANSACTION:
    hQuery = GetCurrentQuery(INPUT cBufferName).
    hQuery:GET-CURRENT(EXCLUSIVE-LOCK).

    CASE cBufferName:
      WHEN "ttCustomer" THEN
        DELETE ttCustomer NO-ERROR.
      WHEN "ttOrder" THEN
        DELETE ttOrder NO-ERROR.
      WHEN "ttOrderLine" THEN
        DELETE ttOrderLine NO-ERROR.
    END CASE.

    IF ERROR-STATUS:ERROR THEN
      MESSAGE "New" cBufferName "record cannot be deleted."
        VIEW-AS ALERT-BOX WARNING.
    ELSE hQuery:DELETE-RESULT-LIST-ENTRY().

  END. /* transaction */

END PROCEDURE.
```

The grid's BeforeRowUpdate event fires when an action occurs that should result in any changed screen values being written to the data source object. However, the event does not indicate that the screen values have in fact changed. So, the sample procedure needs to verify that there are changes and then write those changes to the ProDataSet.

In the UltraGrid, each row belongs to a band which represents a specific level in the hierarchical grid. The event handler, gridBeforeRowUpdate, determines the correct buffer name for a row by retrieving the UltraGridBand class' Key property. The handler uses the RowModified property to verify that the current row in the ProBindingSource is being edited. The handler then uses the ASSIGN( ) method to update the ProDataSet, raising an error if it fails. The handler calls the ProcessRowChanges function to write the changes in the ProDataSet back to the database.

Note: Combining the logic for updating the data source object and the database in the same procedure is done for convenience in this example. If you were building an OpenEdge Reference Architecture-compliant application, you would separate these functions into the correct layers of your application.

```
PROCEDURE gridBeforeRowUpdate:
  DEFINE INPUT PARAMETER rSender AS System.Object          NO-UNDO.
  DEFINE INPUT PARAMETER rArgs   AS CancelableRowEventArgs NO-UNDO.

  DEFINE VARIABLE cBufferName AS CHARACTER                 NO-UNDO.
  DEFINE VARIABLE lResult     AS LOGICAL                   NO-UNDO.

  cBufferName = rArgs:Row:Band:Key.

  IF rBindS:RowModified THEN
  DO TRANSACTION:
    lResult = rBindS:Assign().

    IF NOT lResult THEN
    /* Error assigning changes from grid to dataset */
    DO:
      rArgs:Cancel = TRUE.
      LEAVE.
    END.

    /* Any necessary validation logic goes here. */

    IF NOT ProcessRowChanges(INPUT cBufferName) THEN
    /* Error assigning changes from dataset to database */
    DO:
      rArgs:Cancel = TRUE.
      LEAVE.
    END.
  END.    /* transaction  */

END PROCEDURE.
```

The grid's BeforeRowsDeleted event fires before any rows are deleted. The BeforeRowsDeletedEventArgs class stores the selected rows in the Rows property. The event handler, gridBeforeRowsDeleted, determines the number of rows from the Length property. Note that the Rows property is a 0-based array, so the REPEAT block counts up from zero. The handler then calls the DeleteRow function to delete each row in turn.

To refresh the grid properly, the handler turns off the ProBindingSource's AutoSync property at the start. If only a single row is deleted, the handler removes that row directly from the current query's result list. Otherwise, the handler reopens the query. The handler then reactivates the `AutoSync` property before leaving.

```
PROCEDURE gridBeforeRowsDeleted:
  DEFINE INPUT PARAMETER rSender AS System.Object              NO-UNDO.
  DEFINE INPUT PARAMETER rArgs   AS BeforeRowsDeletedEventArgs NO-UNDO.

  DEFINE VARIABLE rCurrentRow AS UltraGridRow                  NO-UNDO.
  DEFINE VARIABLE cBufferName AS CHARACTER                     NO-UNDO.
  DEFINE VARIABLE hQuery      AS HANDLE                        NO-UNDO.
  DEFINE VARIABLE rRows       AS System.Array                  NO-UNDO.
  DEFINE VARIABLE iNumRows    AS INTEGER                       NO-UNDO.
  DEFINE VARIABLE idx         AS INTEGER                       NO-UNDO.
  DEFINE VARIABLE iStatus     AS INTEGER                       NO-UNDO.
  DEFINE VARIABLE cPromptText AS CHARACTER                     NO-UNDO.

  iNumRows        = rArgs:Rows:Length.
  rRows           = rArgs:Rows.
  rBindS:AutoSync = FALSE. /* query-open should not force a refresh */

  /* Suppress Infragistics default prompt. We need to do this because the
     prompt appears after this procedure has completed - when the records
     have already been deleted. Our own prompt is substituted below. */
  rArgs:DisplayPromptMsg = FALSE.

  /* Prompt for confirmation */
  IF iNumRows > 1 THEN
    cPromptText = "Are you sure you want to delete the "
                + STRING(iNumRows)
                + " selected records?".
  ELSE cPromptText = "Are you sure
                      you want to delete the selected record?".

  MESSAGE cPromptText VIEW-AS ALERT-BOX QUESTION
    BUTTONS YES-NO
    UPDATE lConfirmDelete AS LOGICAL.

  IF NOT lConfirmDelete THEN
    rArgs:Cancel = TRUE.
  ELSE
  deleteTransaction:
  DO TRANSACTION:
    REPEAT idx = 0 TO iNumRows - 1 WHILE rArgs:Cancel = FALSE:
      rCurrentRow = CAST(rRows:GetValue(idx), UltraGridRow).
      cBufferName = rCurrentRow:Band:Key.
      hQuery      = GetCurrentQuery(INPUT cBufferName).

      iStatus     = DeleteRow(INPUT cBufferName, INPUT hQuery,
                              INPUT rCurrentRow).

      IF iStatus > 0 THEN
      DO:
        rArgs:Cancel = TRUE.
        UNDO deleteTransaction, LEAVE deleteTransaction.
      END.

      ELSE DO:
        /*If only deleting one record, use DELETE-RESULT-LIST-ENTRY.*/
        IF iNumRows = 1 THEN
        hQuery:DELETE-RESULT-LIST-ENTRY().
      END.
    END.   /* END REPEAT */

    /* If deleting multiple records, reopen query now. */
    IF iNumRows > 1 THEN
      hQuery:QUERY-OPEN.
```

```
  END. /* transaction */

  rBindS:AutoSync = TRUE.

END PROCEDURE.
```

The `ProcessRowChanges` function uses the ProDataSet SAVE-ROW-CHANGES( ) and ACCEPT-ROW-CHANGES( ) methods to write the changes from the ProDataSet back to the database.

Note: Combining the logic for updating the data source object and the database in the same procedure is done for convenience in this example. If you were building an OpenEdge Reference Architecture-compliant application, you would separate these functions into the correct layers of your application.

```
FUNCTION ProcessRowChanges RETURNS LOGICAL (INPUT cBufferName AS CHARACTER).
  DEFINE VARIABLE hQuery        AS HANDLE NO-UNDO.
  DEFINE VARIABLE hBeforeBuffer AS HANDLE NO-UNDO.
  DEFINE VARIABLE hAfterBuffer  AS HANDLE NO-UNDO.
  DEFINE VARIABLE lResult       AS LOGICAL NO-UNDO.

  hBeforeBuffer = ?.

  CASE cBufferName:
    WHEN "ttCustomer" THEN
    DO:
      hAfterBuffer  = BUFFER ttCustomer:HANDLE.
      hBeforeBuffer = hAfterBuffer:BEFORE-BUFFER.
    END.
    WHEN "ttOrder" THEN
    DO:
      hAfterBuffer  = BUFFER ttOrder:HANDLE.
      hBeforeBuffer = hAfterBuffer:BEFORE-BUFFER.
    END.
    WHEN "ttOrderLine" THEN
    DO:
      hAfterBuffer  = BUFFER ttOrderLine:HANDLE.
      hBeforeBuffer = hAfterBuffer:BEFORE-BUFFER.
    END.
  END CASE.

  /* Save-Row-Changes causes db triggers to fire. */
  lResult = hBeforeBuffer:SAVE-ROW-CHANGES() NO-ERROR.
  IF NOT lResult THEN
    DO:
      hBeforeBuffer:REJECT-ROW-CHANGES().
      RETURN FALSE.
    END.

  IF NOT hBeforeBuffer:ACCEPT-ROW-CHANGES() THEN
    RETURN FALSE.

  RETURN TRUE.

END FUNCTION.
```

The CreateRow function creates a new record in the appropriate temp-table and sets some initial properties. Note the use of the CATCH block here. The ProBindingSource ensures that the parent record is in the buffer by this point. So, the function uses the structured error handlings approach.

```
FUNCTION CreateRow RETURNS INTEGER (INPUT cBufferName AS CHARACTER).

  DO TRANSACTION ON ERROR UNDO, LEAVE:

    CASE cBufferName:
      WHEN "ttCustomer" THEN DO:
        CREATE ttCustomer.
        ASSIGN
          ttCustomer.Custnum = NEXT-VALUE(NextCustNum, sports2000)
          ttCustomer.Name    = "default".
      END.
      WHEN "ttOrder" THEN DO:
        CREATE ttOrder.
        ASSIGN
          ttOrder.OrderNum = NEXT-VALUE(NextOrdNum, sports2000)
          ttOrder.CustNum  = ttCustomer.CustNum.
      END.
      WHEN "ttOrderLine" THEN DO:
        CREATE ttOrderLine.
        ASSIGN
          ttOrderLine.Ordernum = ttOrder.OrderNum.
      END.
    END CASE.

    CATCH rAssignError AS Progress.Lang.Syserror:

      /* Trap any error so we can return a status of 1. */
      MESSAGE rAssignError:GETMESSAGE(1) VIEW-AS ALERT-BOX ERROR.
    END.

  END. /* transaction */

  IF VALID-OBJECT(rAssignError) THEN
  DO:
    RETURN 1.
  END.

  RETURN 0.

END FUNCTION.
```

For more information on CATCH blocks and structured error handling, see the section on error handling enhancements in OpenEdge Getting Started: New and Revised Features.

The DeleteRow function finds the cell in the input UltraGridRow that corresponds to the primary index for each of the temp-tables in the ProDataSet. Using that cell's value, the function finds the ROWID for the passed in row. The handler then repositions to that row and deletes the row from the temp-table. The handler then calls the ProcessRowChanges function to write the changes from the ProDataSet to the database.

Note: Combining the logic for updating the data source object and the database in the same procedure is done for convenience in this example. If you were building an OpenEdge Reference Architecture-compliant application, you would separate these functions into the correct layers of your application.

```
FUNCTION DeleteRow RETURNS INTEGER (INPUT cBufferName AS CHARACTER,
  INPUT hQuery AS HANDLE, INPUT rCurrentRow AS UltraGridRow).
```

```
DEFINE VARIABLE rCells        AS CellsCollection NO-UNDO.
DEFINE VARIABLE rCell0        AS UltraGridCell   NO-UNDO.
DEFINE VARIABLE rCell1        AS UltraGridCell   NO-UNDO.
DEFINE VARIABLE iNum          AS INTEGER         NO-UNDO.
DEFINE VARIABLE iLineNum      AS INTEGER         NO-UNDO.
DEFINE VARIABLE rRowID        AS ROWID           NO-UNDO.
DEFINE VARIABLE cErrorMessage AS CHARACTER       NO-UNDO.

rCells = rCurrentRow:Cells.
rCell0 = rCells[0].

CASE cBufferName:
  WHEN "ttCustomer" THEN DO:
    iNum = INTEGER(rCell0:Text).  /* CustNum */
    FIND ttCustomer WHERE ttCustomer.CustNum = iNum.
    rRowID = ROWID(ttCustomer).
  END.
  WHEN "ttOrder" THEN DO:
    iNum = INTEGER(rCell0:Text).  /* OrderNum */
    FIND ttOrder WHERE ttOrder.OrderNum = iNum.
    rRowID = ROWID(ttOrder).
  END.
  WHEN "ttOrderLine" THEN DO:
    rCell1 = rCells[1].
    iNum = INTEGER(rCell0:Text).  /* OrderNum */
    iLineNum = INTEGER(rCell1:Text).  /* LineNum */
    FIND ttOrderLine WHERE ttOrderLine.OrderNum = iNum
                     AND   ttOrderLine.LineNum = iLineNum.
    rRowID = ROWID(ttOrderLine).
  END.
END CASE.

IF rRowID = ? THEN
  RETURN 1.

CASE cBufferName:
  WHEN "ttCustomer" THEN
    DELETE ttCustomer NO-ERROR.
  WHEN "ttOrder" THEN
    DO:
      /* First, delete the OrderLines of the Order */
      DEFINE VARIABLE hOrderLineQuery AS HANDLE NO-UNDO.
      hOrderLineQuery = GetCurrentQuery(INPUT "ttOrderLine").
      FOR EACH ttOrderLine OF ttOrder:
        DELETE ttOrderline NO-ERROR.
        IF ERROR-STATUS:ERROR THEN
          DO:
            IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
              cErrorMessage = ERROR-STATUS:GET-MESSAGE(1).
            ELSE
              cErrorMessage = "Error deleting temp-table record for
                               Orderline "
                            + STRING(ttOrderLine.OrderNum)
                            + "/"
                            + STRING(ttOrderLine.LineNum).
              MESSAGE cErrorMessage VIEW-AS ALERT-BOX ERROR.
          RETURN 1.
          END.
        IF ProcessRowChanges(INPUT "ttOrderLine") = FALSE THEN
        RETURN 1.
    END.
    DELETE ttOrder NO-ERROR.
  END.
  WHEN "ttOrderLine" THEN
    DELETE ttOrderLine NO-ERROR.
END CASE.

IF ERROR-STATUS:ERROR THEN
```

```
  DO:
    IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
      cErrorMessage = ERROR-STATUS:GET-MESSAGE(1).
    ELSE cErrorMessage = "Temp-table record delete failed.".

    MESSAGE cErrorMessage VIEW-AS ALERT-BOX ERROR.

    RETURN 1.
  END.

  IF ProcessRowChanges(INPUT cBufferName) = TRUE THEN
    RETURN 0.

  ELSE RETURN 1.

END FUNCTION.
```

The `GetCurrentQuery` function determines the correct query to use in the ProBindingSource.

```
FUNCTION GetCurrentQuery RETURNS HANDLE (INPUT cBufferName AS CHARACTER).
  DEFINE VARIABLE hDataSet  AS HANDLE NO-UNDO.
  DEFINE VARIABLE hRelation AS HANDLE NO-UNDO.
  DEFINE VARIABLE hQuery    AS HANDLE NO-UNDO.

  hDataSet = DATASET dsCustOrder:HANDLE.

  CASE cBufferName:
    WHEN "ttCustomer" THEN
      hQuery = hTopQuery.
    WHEN "ttOrder" THEN DO:
      hRelation = hDataSet:GET-RELATION(1).
      hQuery = hRelation:CURRENT-QUERY().
    END.
    WHEN "ttOrderLine" THEN DO:
      hRelation = hDataSet:GET-RELATION(2).
      hQuery = hRelation:CURRENT-QUERY().
    END.
    OTHERWISE
      hQuery = ?.
  END CASE.

  RETURN hQuery.

END FUNCTION.
```

5

Using .NET Forms with ABL Windows

.NET forms and ABL windows have a number of similarities, particularly in visual presentation, but also many differences in their behavior and how you work with them in an ABL session. .NET forms and ABL windows each rely on entirely different object models. .NET forms are classes within a strongly-typed .NET class hierarchy, and ABL windows are widgets, each of which is a kind of handle-based visual object with weak typing and with no explicit object hierarchy. In their native environments, .NET forms interact with, contain, and extend only other .NET classes, while ABL windows interact with and contain only other widgets.

Nevertheless, the user interface for many applications, especially existing applications that you might want to enhance with .NET forms, is already built from the traditional OpenEdge GUI based on ABL windows. You might want to add a few .NET forms to your existing traditional OpenEdge GUI, or you might want to add many more, perhaps replacing all of your traditional GUI with the OpenEdge GUI for .NET, based only on .NET forms. ABL provides features that help to combine .NET forms with ABL windows in a single application, regardless of the balance of integration and migration. In the sections that follow, unless specifically noted, a reference to *form* refers to a .NET form, and a reference to *window* refers to an ABL window.

For details, see the following topics:

- Features for using forms and windows together
- ABL session architecture for forms and windows
- Parenting forms and windows to each other
- Embedding ABL windows in .NET forms
- Handling form and window input
- Managing form and window run-time behavior
- Configuring common session features

Features for using forms and windows together

However you use .NET forms, ABL supports a few mechanisms to enable .NET forms and ABL windows to work together in an ABL session. These mechanisms allow ABL windows to share the following common interactions with .NET forms:

- Managing both form and window objects as a single type of form object on a session form chain
- Parenting .NET forms and ABL windows to each other in combinations of form and window hierarchies using the ABL PARENT attribute
- Embedding the client area of an unrealized ABL window in the client area of a .NET form, allowing the client-area widgets of the window to be displayed as part of the .NET form
- Processing form and window events in common using a single set of WAIT-FOR statement and PROCESS EVENTS statement constructs
- Identifying the most recent form or window to receive focus using a common ACTIVE-FORM system reference

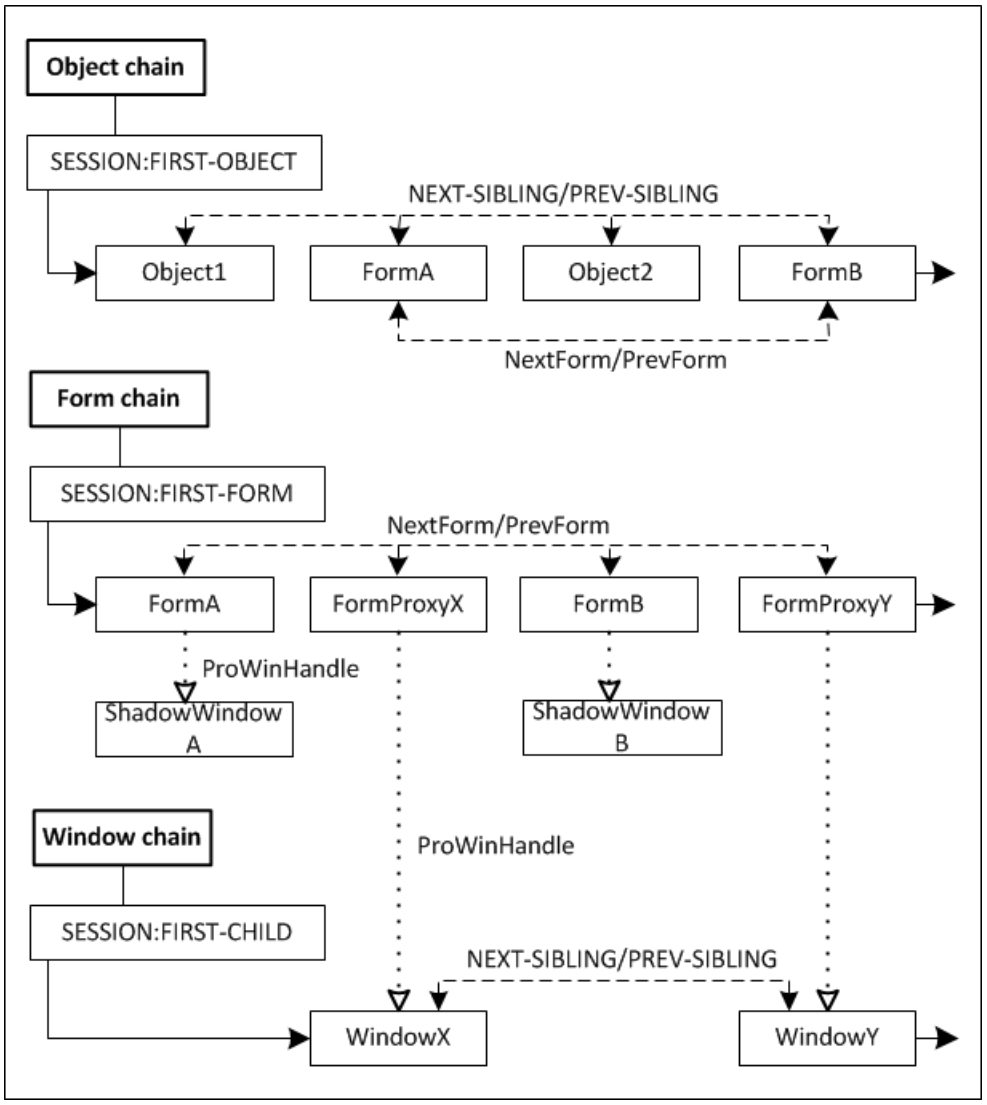
Outside of these common form and window mechanisms, you must continue to work with individual .NET forms and ABL windows using their own instantiation and access mechanisms. Other session features, such as internationalization and localization, fonts, colors, and icon usage require coordination for which both OpenEdge and .NET provide some help. However, you might have to explicitly manage some of these features separately in both the .NET and ABL contexts in order to fully integrate .NET forms with ABL windows in your ABL application.

ABL session architecture for forms and windows

ABL organizes forms and windows into three different chains of objects in a session—the object, form, and window chains. Each form or window that you create appears directly on one or two of these chains depending on the kind of object it is. These chains allow ABL to maintain relationships among different kinds of objects and to search for and manage different types of objects, depending on your application requirements.

The following figure shows how ABL organizes forms and windows in session chains.

Figure 17: ABL session object, form, and window chains



This figure shows the three chains of objects that ABL builds for forms and windows:

- **Object chain** — Contains Progress.Lang.Object references to all class-based objects that you create in a session, including ABL user-defined and .NET objects. This includes both ABL-derived .NET form objects and pure .NET form objects you might create, such as `FormA` and `FormB` in the above figure. As shown in the above figure, the object chain is anchored to the SESSION system handle by the FIRST-OBJECT attribute at one end and by the LAST-OBJECT attribute (not shown) at the other end. All class instances that you create, including .NET objects, are then linked to each other on the object chain in order of creation (until deleted or garbage collected), starting with `SESSION:FIRST-OBJECT`, using the NEXT-SIBLING property, and are linked back from `SESSION:LAST-OBJECT` using the PREV-SIBLING property in reverse order of creation.
- **Form chain** — Contains object references to two types of .NET forms, both of which implement the OpenEdge .NET interface, Progress.Windows.IForm. This interface defines common properties that allow an implementing .NET form object to interact more easily with other form objects, and with ABL windows, in an ABL session. As shown in the above figure, ABL anchors the form chain to the SESSION system handle by the FIRST-FORM attribute at one end and by the LAST-FORM attribute (not shown) at the other end. The data type of these attributes is defined as `Progress.Windows.IForm`, and two properties that this interface defines are NextForm and PrevForm, which allow different types of forms that implement the

interface to be linked together in the form chain. The two basic form classes that can appear in the form chain include:

- Progress.Windows.Form — An OpenEdge .NET form class that inherits from `System.Windows.Forms.Form`, including any of its ABL and .NET subclasses. (OpenEdge provides one .NET subclass, `Progress.Windows.MDIChildForm`.) ABL automatically creates and associates with each `Progress.Windows.Form` object that you instantiate (see `FormA` in the above figure) a unique window widget referred to as a shadow window (see `ShadowWindowA` in the above figure). Note that a shadow window never appears on the window chain (described in the following bullet), and you can only access it through its associated form using the ProWinHandle property (defined by `Progress.Windows.IForm`). For more information on shadow windows and their relationship to form objects, see Shadow windows on page 186.
- Progress.Windows.FormProxy — An OpenEdge .NET form class that inherits directly from `System.Object`. ABL automatically creates and associates a unique `Progress.Windows.FormProxy` object (see `FormProxyX` in the above figure) with every ABL window that you create in a session (see `WindowX` in the above figure). Note that a `FormProxy` object never appears on the object chain (its `NEXT-SIBLING` and `PREV-SIBLING` properties always return the `Unknown value (?)`), and you can reference its associated window using the ProWinHandle property (defined by `Progress.Windows.IForm`). For more information on `FormProxy` objects and their relationship to ABL windows, see FormProxy objects on page 187.

Note: In non-GUI ABL sessions, there is no form chain; any `Progress.Windows.IForm` objects that you instantiate appear only on the object chain. However, note that creating such objects is not typical in a non-GUI session.

- **Window chain** — The standard ABL window chain available in all ABL client applications, which contains the handles to all ABL windows created in the session, such as `WindowX` in the above figure. ABL anchors the window chain to the SESSION system handle by the FIRST-CHILD attribute at one end and by the LAST-CHILD attribute (not shown) at the other end, with all window widgets linked to each other in the chain using the NEXT-SIBLING and PREV-SIBLNG attributes. As noted in the previous bullet, the window chain in an ABL session accessing .NET forms does not contain shadow windows.

Shadow windows

ABL provides a shadow window solely to enable forms and windows to parent each other in an ABL session. A shadow window has no visualization and has only one attribute, the PARENT attribute, which you can use to parent the associated form to another form or window. Note that the ABL virtual machine (AVM) creates a shadow window for each Progress.Windows.Form object that you create in a session and destroys the associated shadow window when the form closes, either by the user or programmatically using the `Close( )` method on the form. You cannot explicitly create or delete a shadow window on a form's ProWinHandle property, and any attempt to do so raises a run-time error. For more information on using the shadow window of a form to parent forms and windows, see Parenting forms and windows to each other on page 187.

FormProxy objects

ABL provides the Progress.Windows.FormProxy class solely as a form object for referencing each ABL window that you create in a session. A `FormProxy` object has no visualization of its own and serves only to provide the ProWinHandle property used to access its associated ABL window and the PrevForm and NextForm properties to reference that ABL window on the form chain. When an ABL session references .NET forms in any way, the AVM automatically creates a `FormProxy` object for each ABL window that you create. For example, such a reference can include querying the FIRST-FORM or LAST-FORM attribute on the SESSION handle, even if you have not yet instantiated any .NET forms in the session. If the first reference to a .NET form occurs after ABL windows are already created, the AVM also automatically creates the necessary `FormProxy` objects retroactively.

You cannot directly instantiate a `FormProxy` object or explicitly delete an existing `FormProxy` object using the `DELETE OBJECT` statement, and any attempt to do so raises a run-time error. ABL automatically deletes any `FormProxy` object associated with an ABL window, when you delete the window using the `DELETE OBJECT` statement.

In addition to linking all windows that you create together with the forms that you create on the form chain, ABL uses the `FormProxy` object to identify its associated ABL window when that window is the most recent form or window to receive focus in a session. For more information on identifying the most recent form or window to receive focus, see Handling form and window input on page 195.

Note: ABL does not create a `FormProxy` object to reference the default window for an ABL session. GUI applications typically do not use the default window and you can access the default window, if necessary, using the DEFAULT-WINDOW system handle. For more information on this system handle in an ABL session accessing .NET forms, see Table 9: ABL elements that change behavior with .NET forms on page 197.

Embedded windows

Embedded windows, whose client areas are displayed in .NET forms, also appear on the window chain, and could be represented by both `WindowX` and `WindowY` in Figure 17: ABL session object, form, and window chains on page 185. At the same time, both `FormA` and `FormB` could represent forms with associated embedded windows. While the client areas of embedded windows and associated forms are linked, you must manage each type of object using its native mechanisms. The behavior and state of embedded windows changes compared to non-embedded windows, but their status as window widgets in an ABL session is the same. For more information, see Embedding ABL windows in .NET forms on page 189.

Parenting forms and windows to each other

ABL allows you to link non-modal windows in a hierarchy by setting the PARENT attribute of a child window to the handle of its parent window, and allows you to repeat this for as many child windows as you want using any window as a parent. When a parent window in such a window hierarchy is minimized, all of its child windows (and their descendents) are hidden. Also, if you delete the handle of a parent window, the handles of all its child windows and their descendents are deleted.

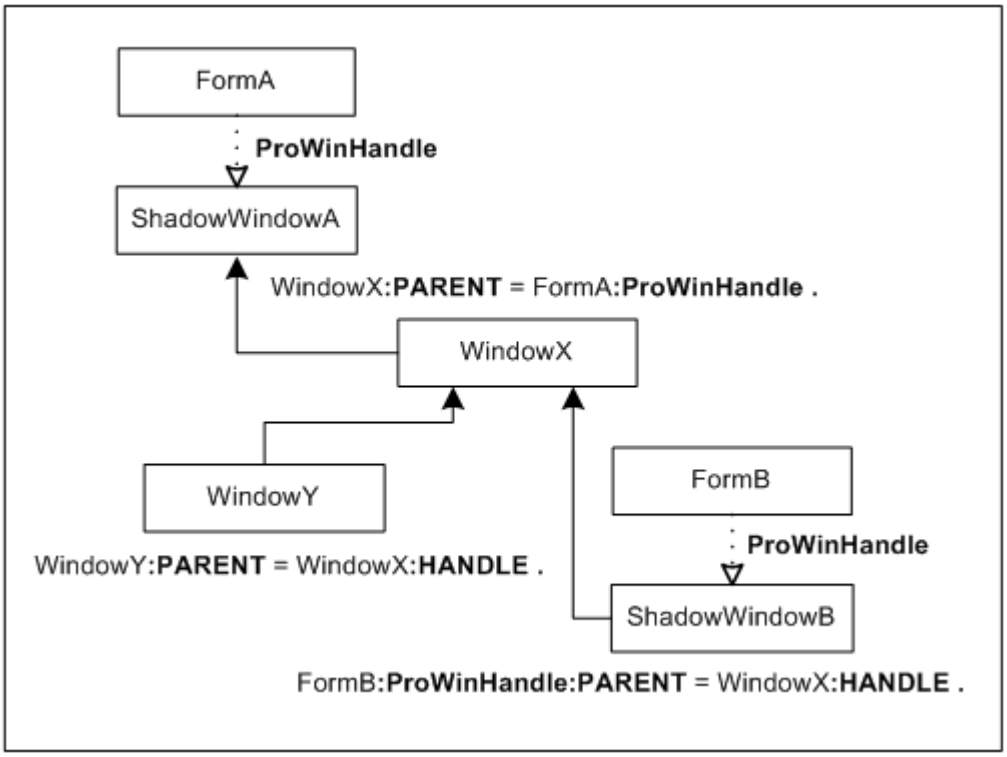
.NET allows you to link non-modal forms in a similar hierarchy by setting the `Owner` property of a child `System.Windows.Forms.Form` to the object reference of its owner (parent) form, or by passing the object reference of a child form to the `AddOwnedForm( )` method of its parent form. For form hierarchies, child windows (and their descendents) are also hidden when you minimize a parent form. In addition, form hierarchies have a feature not available with window hierarchies, which is that child forms never display behind their parent form. Within .NET, the deletion of form objects in hierarchies is handled by .NET garbage collection.

In an ABL session, you can set up these separate hierarchies for non-modal windows and forms using the `PARENT` attributes of windows and the `Owner` properties or `AddOwnedForm( )` methods of forms. Each hierarchy then behaves in its native ABL or .NET fashion.

In addition to these separate form and window hierarchies, ABL allows you to create mixed form and window hierarchies by parenting to each other the non-modal ABL windows that you create and the shadow windows that ABL creates for non-modal forms. In order to create such a mixed form and window hierarchy, you must instantiate the form objects as `Progress.Windows.Form` objects (**not**`System.Windows.Forms.Form`). This allows ABL to create the required shadow window for each .NET form. (For more information on shadow windows, see ABL session architecture for forms and windows on page 184.) To create a mixed hierarchy, you can then use the `PARENT` attributes of the non-modal ABL windows and form shadow windows to set up the parent-child relationships between them in any combination. This mixed hierarchy then behaves much like a pure ABL window hierarchy.

The following figure shows a mixed form and window hierarchy and the PARENT attribute assignments used to set it up.

Figure 18: Form and window hierarchies



This figure shows a hierarchy consisting of a form (`FormA` object reference) that is the parent of a window (`WindowX` handle), which is in turn the parent of both a window (`WindowY` handle) and a form (`FormB` object reference). Each of the three parent-child relationships appears with a line of sample ABL code that sets it up. For example, to make `FormB` a child of `WindowX`, it sets the `PARENT` attribute of the `FormB` shadow window (referenced by the `ProWinHandle` property of `FormB`) to the handle of `WindowX`, and so on.

Note: If `WindowX` is a window handle (as shown), `WindowX:HANDLE` is a valid reference to the ABL `HANDLE` attribute, but is redundant and shown here only for clarity.

You can also parent a form to a form within a mixed hierarchy this way. For example, like adding a `FormC` as a child form of `FormB` in the above figure:

```
FormC:ProWinHandle:PARENT = FormB:ProWinHandle.
```

Minimizing the parent of a mixed form and window hierarchy works the same as for a separate form or window hierarchy. Also, if you close a parent form with child windows, the AVM deletes all the connected windows in the hierarchy, as with deleting a parent window in a pure window hierarchy. However, for a parent window with a child form, deleting the parent window does not automatically delete its child forms in a mixed hierarchy. You must handle the closing and deletion of these child forms yourself.

Embedding ABL windows in .NET forms

You can embed the client area of an ABL window in a .NET host form as long as the ABL window has not yet been realized. This allows all the client area widgets in the window to be displayed in the client area of the .NET form instead of in the ABL window to which they are attached. Using this feature, you can retain much of your existing OpenEdge GUI code as you migrate from a traditional OpenEdge GUI to an OpenEdge GUI for .NET application.

Using a common mechanism, ABL supports two basic ways to embed an ABL window in a .NET host form:

- Embedding a single window in an MDI child form
- Embedding one or more windows in any .NET or ABL-derived .NET form that is based on the `System.Windows.Forms.Form` class

Thus, this one feature allows you to migrate an existing OpenEdge GUI to a GUI for .NET in several different ways. You can use it to transform your existing OpenEdge GUI application into a .NET MDI application, migrate each window to a corresponding .NET host form, migrate elements of more than one window into a single .NET host form, or use a combination of these approaches in migrating your existing application.

Elements of an embedded ABL window

When you embed an ABL window in a .NET form, all of the frames that are attached to the window (and their child widgets) can be displayed in the client area of the form. No other widgets or components of the ABL window become part of the associated form. Thus, the following components of an embedded ABL window never appear in its host form:

- **Menu and submenu widgets** — You must implement all menus for the form as .NET menu and toolbar objects. For an example of a .NET form that implements menus and toolbars, see Sample ABL-derived .NET MDI form on page 120.

  Note: Using the traditional OpenEdge GUI, ABL applications implement toolbars using widgets displayed in the client area of an ABL window. Therefore, these ABL-implemented toolbars appear in the host form.

- **Border controls** — You must implement all border controls for the host form using .NET form properties and methods.

- **Message area** — Any messages that you currently display in the message area of the embedded window (using MESSAGE statements) are displayed in a separate ABL window or in ABL alert boxes, depending on `MESSAGE` statement options.
- **Status area** — Any ABL messages otherwise displayed in the status area of the embedded window are ignored.

After you embed an ABL window in a .NET form, the host form controls the basic presentation and availability of embedded frames on the screen. However, the ABL session controls most aspects of the appearance and behavior of these frames and all related widgets within the form client area, including frame scrolling. This means that once the host form is displayed, you can control and interact with the behavior of these widgets using ABL widget attributes, methods, and events.

This enhances the benefit of embedding ABL windows in forms because you can maintain all of your existing ABL code that controls and manages the embedded widgets. However, if you later decide to replace these widgets with .NET controls, you must entirely implement the .NET form using .NET controls.

In addition, when you embed an ABL window, the functions of its own attributes, methods, and events are retained, changed, or ignored by the ABL session, depending on the element. Basically, those elements related to window features that the ABL session still controls continue to provide a function, where as those related to window features replaced by form features either provide a modified function or are ignored by the ABL session.

Using the ABL features to embed a window

To embed ABL windows in .NET forms, ABL provides a mechanism based on two OpenEdge .NET classes and an associated property:

- **`Progress.Windows.MDIChildForm` class** — This is a form class based on `Progress.Windows.Form` that is specifically designed to embed the client area of one ABL window in an MDI child form.
- **`Progress.Windows.WindowContainer` class** — This is a control container class based on `System.Windows.Forms.UserControl` that can embed the client area of one ABL window in a .NET user control. You can then add this user control to a form in order to embed the ABL window in the .NET form. Using multiple `WindowContainer` objects, you can embed multiple ABL windows in a single .NET form.
- **`EmbeddedWindow` property** — This is a `HANDLE` property on both the `MDIChildForm` class and the `WindowContainer` class that contains the handle of the ABL window whose client area is embedded in the respective .NET control container.

To embed an ABL window in a .NET form, you begin by setting the `EmbeddedWindow` property of the appropriate object to the handle of the window whose client area you want to embed. Once you have set this property, the behavior of the ABL window you have embedded, as noted previously, changes. For more information on these changes, see the `EmbeddedWindow` property entry in OpenEdge Development: ABL Reference.

Each .NET class for embedding ABL windows (`MDIChildForm` and `WindowContainer`) requires a different procedure for embedding and displaying the client area of a window in a form. Once you have done this, you can work with the .NET form and its controls using .NET properties, methods, and events, and you can work with the embedded widgets using the same code you used before. The sections that follow describe how to use each of these classes to embed windows in a form.

Embedding a single window in an MDI child form

The `Progress.Windows.MDIChildForm` class is designed to embed one ABL window so that its client area becomes the client area of a .NET MDI child form, as described in the following general steps.

To embed an ABL window in a .NET MDI child form:

1. Ensure that the window object you want to embed has not been visualized or otherwise realized.
2. Create an MDI form object from the `Progress.Windows.Form` class, which becomes the MDI parent for your application. You can create the MDI parent directly with this class or you can create an ABL-derived MDI form from this class. For an example of an ABL-derived MDI form, see Sample ABL-derived .NET MDI form on page 120.
3. For each new instance of a `Progress.Windows.MDIChildForm` you want to create in the MDI parent (typically in response to a menu selection in the MDI parent, such as **New** > **File**), instantiate the class as in the following example, where `MdiChild` is an object reference to an `MDIChildForm`, `MdiContainer` is an object reference to the MDI parent form, and `hWin` is the handle to the current ABL window you want to embed. For example:

```
MdiChild = NEW Progress.Windows.MDIChildForm( MdiContainer, hWin ).
MdiChild:ClientSize = NEW System.Drawing.Size
                          ( hWin:WIDTH-PIXELS, hWin:HEIGHT-PIXELS ).
```

Note: If you are creating this `MDIChildForm` within an ABL-derived MDI form, `MdiContainer` would be replaced with `THIS-OBJECT`.

4. After all MDI initialization is complete, block on the form using a .NET `WAIT-FOR` statement, and in appropriate handlers on .NET events for the MDI, create MDI child forms as you have designed for Step 3. For example:

```
MdiContainer:Show( ).
MdiChild:Show( ).
hWin:VISIBLE = YES.
. . .
WAIT-FOR Application:Run( MdiContainer ).
```

Also, after you have first created an `MDIChildForm` instance, you can change the setting of its `EmbeddedWindow` property to the handle of a different ABL window. However, if it has already been realized, you must first delete the window to which the property was previously assigned before assigning the handle of another window to it.

The following procedure fragment creates a simple MDI containing one MDI child form containing a button from an embedded ABL window. The bold elements indicate the most critical and recommended code for making this work.

```
USING System.Windows.Forms.* FROM ASSEMBLY.
USING System.Drawing.* FROM ASSEMBLY.
USING Progress.Windows.* FROM ASSEMBLY.
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE hWin AS HANDLE NO-UNDO.
DEFINE VARIABLE MdiContainer AS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE MdiChild AS Progress.Windows.MDIChildForm NO-UNDO.

/* Create main menu for the MDI container (not shown for brevity). */
DEFINE VARIABLE MainMenu AS System.Windows.Forms.MainMenu NO-UNDO.
. . .

/* Define a frame with a button. */
DEFINE BUTTON b1 LABEL "Button 1" SIZE 18 BY 1.14.
DEFINE FRAME f1 b1 AT ROW 2 COL 22 WITH SIDE-LABELS THREE-D SIZE 60 BY 6.
```

```
/* Create the window. It will not be realized at this point. */
CREATE WINDOW hWin ASSIGN WIDTH = 60 HEIGHT = 6.
FRAME f1:PARENT = hWin.

/* A trigger to prove that clicking the button works. */
ON 'CHOOSE':U OF b1
DO:
    MESSAGE "Click of Button 1".
    RETURN.
END.

/* Create the MDI container form. */
MdiContainer = NEW Progress.Windows.Form( ).
MdiContainer:Text = "MDI Container Form".
MdiContainer:IsMdiContainer = TRUE.
MdiContainer:Menu = MainMenu.
MdiContainer:Show( ).

/* Create the MDI child form, embedding the window into it. */
MdiChild = NEW Progress.Windows.MDIChildForm( MdiContainer, hWin ).
MdiChild:Text = "MDI Child Form".
MdiChild:ClientSize = NEW Size( hWin:WIDTH-PIXELS, hWin:HEIGHT-PIXELS ).
MdiChild:FormClosed:Subscribe( "MdiChild_FormClosed" ).
MdiChild:Show( ).

/* Now make the window visible.
   It will be realized inside the MDI child form. */
hWin:VISIBLE = YES.
ENABLE ALL WITH FRAME f1.

WAIT-FOR Application:Run(MdiContainer).

/* Delete the embedded window after MDIChildForm closes. */
PROCEDURE MdiChild_FormClosed:

  DEFINE INPUT PARAMETER sender AS System.Object.
  DEFINE INPUT PARAMETER e      AS System.EventArgs.

  DELETE WIDGET hWin.

END PROCEDURE.
```

Recommended code in this example includes:

- Sizing the client area of the `MDIChildForm` object (`MdiChild`) to the client area of the ABL embedded window (`hWin`)
- Handling the `FormClosed` event on the MDI child form in order to delete the embedded window when it closes

Embedding one or more windows in a .NET form

`Progress.Windows.WindowContainer` is a control container class designed to embed the client area of one ABL window in any .NET form. You can do this in one of two basic ways:

- You can first embed a window in a `WindowContainer`, then add the `WindowContainer` to the control collection of any .NET form.
- You can add a `WindowContainer` to the control collection of any .NET form, then embed a window in the `WindowContainer`.

In this way, you can embed the client area of one or more windows in any single .NET form.

Note: If you attempt to add a `WindowContainer` to a `Progress.Windows.MDIChildForm`, the behavior of the MDI child form and its controls becomes unpredictable because an `MDIChildForm` object is designed to embed only the single window for which you create and initialize the form.

The following general steps describe how to use a `WindowContainer` to embed ABL windows in .NET forms.

To embed one or more ABL windows in any .NET form:

1. Ensure that any window object you want to embed has not been visualized or otherwise realized.
2. Instantiate the .NET form where you want to embed an ABL window. For example:

```
DEFINE VARIABLE MainForm AS Progress.Windows.Form NO-UNDO.
. . .
MainForm = NEW Progress.Windows.Form( ).
```

3. Instantiate a `Progress.Windows.WindowContainer`. For example:

```
DEFINE VARIABLE WinContainer AS Progress.Windows.WindowContainer
  NO-UNDO.
. . .
WinContainer = NEW Progress.Windows.WindowContainer( ).
```

4. Set `WindowContainer` properties to fit the contents of the embedded window in the container (`Size`) and to position the `WindowContainer` in the client area of its parent form (`Location`). (Note that the default `Location` property setting for a control within a form is `Point( 0, 0 )`, which you can also use to position the `WindowContainer`.) Then, set its `EmbeddedWindow` property to the handle of the window you want to embed (`hWin` in the following example) and set its `Parent` property to the object reference of its parent form in order to add the container to the form's control collection. For example:

```
WinContainer:Location = NEW Point( 10, 10 ).
WinContainer:Size = NEW Size( hWin:WIDTH-PIXELS, hWin:HEIGHT-PIXELS ).
WinContainer:EmbeddedWindow = hWin.
WinContainer:Parent = MainForm.
```

5. Repeat Step 3 and Step 4 appropriately for each additional ABL window you want to embed in the form.
6. Make both the .NET form and its embedded ABL windows visible and do any other initialization required, then block on the form using an appropriate .NET `WAIT-FOR` statement. For example:

```
WinContainer:Show( ).
hWin:VISIBLE = YES.
. . .
WAIT-FOR Application:Run( MainForm ).
```

Note that when you embed the client area of an ABL window in a form, the ABL window and its widgets do not interact directly with any other .NET controls that may be added to the form. This means, for example, the embedded client area does not participate in the tab order of the form. Thus, there is no way to tab into the embedded client area from another .NET control or another `WindowContainer`, and there is no way to tab out of the embedded client area into another .NET control or other `WindowContainer`. All tabbing within an embedded client area stays within the `WindowContainer` where it is embedded.

The following procedure fragment embeds a single ABL window in a .NET form using a `WindowContainer`. The bold elements indicate the most critical and recommended code for making this work:

```
USING System.Windows.Forms.* FROM ASSEMBLY.
USING System.Drawing.* FROM ASSEMBLY.
USING Progress.Windows.* FROM ASSEMBLY.
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE hWin AS HANDLE NO-UNDO.
DEFINE VARIABLE MainForm AS Progress.Windows.Form NO-UNDO.
DEFINE VARIABLE WinContainer AS Progress.Windows.WindowContainer NO-UNDO.

/* Create main menu for the form (not shown for brevity). */
DEFINE VARIABLE MainMenu AS System.Windows.Forms.MainMenu NO-UNDO.
. . .

/* Define a frame with a button. */
DEFINE BUTTON b1 LABEL "Button 1" SIZE 18 BY 1.14.
DEFINE FRAME f1 b1 AT ROW 2 COL 22 WITH SIDE-LABELS THREE-D SIZE 60 BY 6.

/* Create the window. It will not be realized at this point. */
CREATE WINDOW hWin ASSIGN WIDTH = 60 HEIGHT = 6.
FRAME f1:PARENT = hWin.

/* A trigger to prove that clicking the button works. */
ON 'CHOOSE':U OF b1
DO:
    MESSAGE "Click of Button 1".
    RETURN.
END.

/* Create the form. */
MainForm = NEW Progress.Windows.Form( ).
MainForm:Text = "Embedded Window Sample".
MainForm:Menu = MainMenu.
MainForm:ClientSize = NEW Size(hwin:WIDTH-PIXELS, hwin:HEIGHT-PIXELS).
MainForm:FormClosed:Subscribe( "Window_FormClosed" ).
MainForm:Show( ).

/* Create the WindowContainer, embedding the window into it. */
WinContainer = NEW Progress.Windows.WindowContainer( ).
WinContainer:Size = NEW Size( hWin:WIDTH-PIXELS, hWin:HEIGHT-PIXELS ).
WinContainer:EmbeddedWindow = hWin.WinContainer:Parent = MainForm.
WinContainer:Show( ).

/* Now make the window visible.
   It will be realized inside the WindowContainer. */
hWin:VISIBLE = YES.
ENABLE ALL WITH FRAME f1.

WAIT-FOR Application:Run(MainForm).

/* Delete the embedded window after the main form closes. */
PROCEDURE Window_FormClosed:

  DEFINE INPUT PARAMETER sender AS System.Object.
  DEFINE INPUT PARAMETER e      AS System.EventArgs.
```

```
  DELETE WIDGET hWin.

END PROCEDURE.
```

Recommended code in this example includes:

- Sizing the client area of the form object (`MainForm`) and the area of the `WindowContainer` object (`WinContainer`) to the client area of the ABL embedded window (`hWin`)
- Handling the `FormClosed` event on the main form in order to delete the embedded window when it closes

The `WindowContainer`, in this case, also relies on the default setting for its `Location` property.

Behavior of forms with embedded windows

The AVM passes all unhandled keystrokes in an embedded ABL client area to the form that contains it. This means that you can interact with form menus and use menu and toolbar accelerator keys even when focus is on the embedded client area.

If the physical client area of an embedded window is larger than the client area of its window container (`Progress.Windows.WindowContainer` or `Progress.Windows.MDIChildForm`), you can set the `SCROLL-BARS` window attribute to `TRUE`. This allows the AVM to scroll the embedded frames within the window container. (The `AutoScroll` property of the window container has no effect on embedded window scrolling.) You can also use the `VIRTUAL-HEIGHT-CHARS`, `VIRTUAL-HEIGHT-PIXELS`, `VIRTUAL-HEIGHT-CHARS`, and `VIRTUAL-HEIGHT-PIXELS` window attributes to specify the virtual window size used to scroll the client area.

For more information on other window attributes that work with embedded windows, see the reference entry for the `EmbeddedWindow` property in OpenEdge Development: ABL Reference.

Handling form and window input

To handle input for .NET forms and ABL windows together in an ABL application, ABL supports two basic mechanisms that work with both forms and windows in common:

- A common WAIT-FOR statement syntax that can handle general event processing for both forms and windows, as well as ABL non-GUI events
- An ACTIVE-FORM system reference to detect input focus for both forms and windows

Common event handling for forms and windows

As described previously (see Initializing and blocking on .NET forms on page 99), in order to handle input for a given .NET form, you must execute a single `WAIT-FOR` statement that calls a .NET input-blocking method that is appropriate for the type of form (non-modal or modal). Thus, for any number of non-modal forms for which you handle input simultaneously, you must use a single `WAIT-FOR` statement that calls the `System.Windows.Forms.Application:Run( )` method to handle events for all of them (see Blocking on non-modal forms on page 100). If you use non-modal .NET forms and ABL windows simultaneously, you can also use this single `WAIT-FOR` statement to handle input for all non-modal .NET forms and ABL windows.

For modal forms (dialog boxes), you must use a separate WAIT-FOR statement that calls the .NET ShowDialog( ) method on each modal form that you handle, similar to handling an ABL dialog box (see Blocking on modal dialog boxes on page 103). You can also handle input for modal ABL dialog boxes in the same session by blocking for each ABL dialog box using the same WAIT-FOR statement when not using .NET forms.

If you are adding .NET forms to an existing ABL application that already uses ABL windows, and this application simultaneously executes more than one WAIT-FOR statement or other ABL input-blocking statement (such as UPDATE or PROMPT-FOR) to simultaneously handle non-modal ABL windows and non-GUI ABL events (such as socket events), you must ensure that the one WAIT-FOR statement calling the .NET Application:Run( ) method is also executing in order to handle input for your new non-modal .NET forms.

Note: Progress Software Corporation recommends that you use a single WAIT-FOR statement to process all the non-modal events in your application, including both .NET and ABL events. However, if you choose to simultaneously execute multiple WAIT-FOR statements (stacked WAIT-FOR statements) to handle ABL non-modal windows in addition to the single WAIT-FOR statement handling .NET non-modal forms, you must ensure that all such WAIT-FOR statements terminate in reverse order of execution. Otherwise, your application can have unpredictable behavior.

Note: Also, if in the context of Application:Run( ) you directly or indirectly execute an additional ABL input-blocking or event-processing statement that blocks on a non-modal ABL window, in certain contexts (such as within a user-defined function or non-VOID method) the AVM raises the STOP condition.

.NET and ABL use somewhat different models for the initialization, display, and closing of forms and windows. For example, when a WAIT-FOR statement calling Application:Run(rForm1) on a main form (rForm1) returns from execution, all non-modal forms that were displayed and processed by that statement also close automatically. However, you must respond appropriately and explicitly close (hide) any open non-modal ABL windows that are handled by the same WAIT-FOR statement. In general, you must observe all the ABL requirements for managing elements of the traditional OpenEdge GUI whether or not you also use elements of the OpenEdge GUI for .NET.

Common focus detection for forms and windows

Using the traditional OpenEdge GUI, the ACTIVE-WINDOW system handle returns the handle of the last non-modal ABL window to generate an ENTRY event (that is, the last window to receive focus). If you use .NET forms, you can use the static ActiveForm property on System.Windows.Forms.Form to return the object reference to the last non-modal form to receive focus. However, when you use .NET forms and ABL windows together, you cannot use these two mechanisms to reliably determine the most recent non-modal form or window to receive focus. If a form is the last to receive focus, ACTIVE-WINDOW returns the Unknown value (?), and if a window is the last to receive focus, the static ActiveForm property returns an object reference with no meaning to the ABL session context.

Instead, to identify the last form or window to receive focus, ABL provides the ACTIVE-FORM system reference. ACTIVE-FORM returns a Progress.Windows.IForm interface reference to a form object on the form chain (see ABL session architecture for forms and windows on page 184). Thus, depending on the object that received the last focus, ACTIVE-FORM can return a reference to a .NET form object instantiated from Progress.Windows.Form (or a derived class)) or a reference to the Progress.Windows.FormProxy object associated with an ABL window that you have created in the session.

Note: For ACTIVE-FORM to reliably return the object reference to each .NET form that receives focus in an ABL session, you must create all .NET forms from Progress.Windows.Form (or a derived form, **never** System.Windows.Forms.Form).

If last focus is in the client area of an embedded ABL window, ACTIVE-FORM returns a Progress.Windows.IForm reference to the Progress.Windows.Form (or derived form) that embeds the ABL window. For more information, see Embedding ABL windows in .NET forms on page 189.

If ACTIVE-FORM returns a Progress.Windows.Form, you can then access the form class members by casting its Progress.Windows.IForm reference. If ACTIVE-FORM returns a Progress.Windows.FormProxy, you can access the ABL window it is associated with using the window handle returned by its ProWinHandle property.

Note: If the ABL default window is the last window to receive focus, ACTIVE-FORM returns the Unknown value (?), because ABL doesn't create a FormProxy object for the default window.

Managing form and window run-time behavior

In a session where you access both forms and windows, you must work with each individual form and window using the native features supported for it. In other words, ABL does not map window attributes to form properties using associated window and form objects. For example, there is no TITLE attribute on a shadow window that you can use to set its form's Text property, and there is no Text property on a Progress.Window.FormProxy object to set the associated window's TITLE attribute.

Also, aside from the features that ABL supports for working with forms and windows in common, such as parenting and detecting focus for forms and windows or processing form and window events, several ABL elements have modified behavior when accessing .NET forms and windows compared to when accessing ABL windows alone. The following table lists these ABL elements and how accessing .NET forms might affect their operation in your ABL session.

Table 9: ABL elements that change behavior with .NET forms

ABL element	Behavior with .NET forms
ACTIVE-WINDOW system handle	Returns a read-only handle to the last window to receive an ENTRY event (focus). If a form (not a window) is the last to receive focus, this handle returns the Unknown value (?). To obtain an indication of the last form or window to receive focus, use the ACTIVE-FORM system reference. For more information, see Common focus detection for forms and windows on page 196.
ALWAYS-ON-TOP handle attribute	Specifies that the window is a topmost window that remains on top of all non-topmost windows (ABL or non-ABL) on the Windows desktop. .NET provides the TopMost property to specify the same behavior for a given form. Windows maintains separate categories for topmost and non-topmost windows across applications. So, you can specify both ABL topmost windows using the ALWAYS-ON-TOP attribute and .NET topmost forms using the TopMost property. All these topmost windows and forms remain on top of all non-topmost windows and forms, and users can move any topmost window or form to the foreground.

ABL element	Behavior with .NET forms
CURRENT-WINDOW system handle	Returns the handle to the window to which an ABL frame is parented when there is no parent specified for the frame. ABL frames cannot be parented to .NET forms. So, any attempt to assign the handle of a form's shadow window (specified by the ProWinHandle property) to CURRENT-WINDOW raises a run-time error.
DEFAULT-WINDOW system handle	Returns the handle to the default window created by the AVM at startup. This window displays all ABL frames until the CURRENT-WINDOW system handle is set to the handle of another [dynamic] window or unless a displayed frame is explicitly parented to another window. Its message area also displays messages that have no other available display destination. ABL frames cannot be parented to .NET forms. So, ABL ignores any attempt to assign the handle of a form's shadow window (specified by the ProWinHandle property) to DEFAULT-WINDOW, leaving its previous value unchanged.
LOAD-ICON( ) and LOAD-SMALL-ICON( ) handle methods	These methods allow you to load a large and small icon (respectively) to display for a window during common window operations, such as application switching and displaying on the taskbar. .NET provides the Icon property to specify a default icon for a form to display for these common window operations. You can maintain uniform icons across .NET forms and ABL windows in an application by using these methods to load the same large and small icon you are using to set the Icon property for .NET forms.
SHOW-IN-TASKBAR handle attribute	Specifies whether an icon for the window appears on the taskbar. .NET also provides a ShowInTaskbar property to specify the same behavior for forms. Windows has a setting to group multiple window icons under one icon from the same application on the taskbar. ABL windows do not conform to this setting and always display individually on the taskbar. However, .NET form icons can and do group on the taskbar according to this setting, even for forms created in an ABL session with ABL windows.
TOP-ONLY handle attribute	Indicates whether another window in the ABL session can overlay a specified window. Setting this attribute to TRUE allows the specified window to overlay all other .NET forms and non-TOP-ONLY ABL windows in the session. .NET does not support a similar property to allow a form to overlay all other forms and windows in a session.

Configuring common session features

ABL and .NET rely on different mechanisms to determine some session-wide settings, but provide varying support for making some of these settings consistent between the ABL and .NET context. This support addresses:

- Palette management
- Font management
- Regional settings—localization

Palette management

On displays that support 256 colors (mainly clients connecting to a terminal server), ABL uses the Palette Manager to map application colors to the available 256 colors. For example, if an ABL application displays two 256-color bit maps, ABL allocates colors from a palette to the images. Because the palette is limited to 256 colors in total, ABL builds the palette so that it makes each 256-color bitmap look as good as it can while using only a portion of the available palette colors.

If a .NET form includes a bitmap image, ABL does not attempt to include the colors from that image in its palette. .NET handles its own palette management. Thus, windows and forms have separate palettes. This can result in unexpected changes in the appearance of bitmaps when focus changes between windows and forms. However, this behavior is typical of applications running on 256-color displays.

Font management

ABL reads its font settings from an initialization (`.ini`) file or from the registry, depending on the values of the Registry Base Key (`-basekey`) and Initialization File (`-ininame`) startup parameters. The ABL default font is 8-point MS Sans Serif. The default font for a form (as created in Visual Studio 2005) is 8.25-point Microsoft Sans Serif. In practice, these fonts are visually indistinguishable from each other. The main difference is in the font implementation—Microsoft Sans Serif is a Unicode-based TrueType font and MS Sans Serif is a raster (bitmapped) font and is limited to the Western character set.

If you want to use a font other than Sans Serif in an application that displays both forms and windows, you must set that font in your application initialization file or in the appropriate registry entry for OpenEdge windows, and you must also design your application forms with this font.

Regional settings: localization

ABL and .NET use different mechanisms to determine regional settings, such as numeric and date formats.

ABL, by default, ignores system-wide regional settings for numeric formats (decimal point and digit grouping characters) and date formats (the order of the day, month, and year). Instead, it defaults to American settings, and you must use startup parameters to modify this behavior. So, to use European number and date formats in an ABL session, you must start the AVM using the European Numeric Format (`-E`) and Date Format (`-d`) startup parameters. For example:

```
prowin32 -E -d dmy
```

.NET, by default, gets its regional settings from the Control Panel. You can also create custom regional settings for the current .NET context by instantiating and setting properties for a `System.Globalization.CultureInfo` object.

Thus, in order to ensure uniformity between .NET forms and ABL windows, Progress Software Corporation recommends that you start the ABL session with the Use OS Locale (`-useOsLocale`) startup parameter. This startup parameter tells the AVM to query the current Windows locale in order to determine what characters to use for the decimal point and digit grouping and in which order dates should be displayed. As long as you do not override these settings for forms (as explained in the next paragraph), the formatting of numeric and date values will be consistent between ABL windows and .NET forms in your application. The SESSION:NUMERIC-DECIMAL-POINT, SESSION:NUMERIC-SEPARATOR, and SESSION:DATE-FORMAT attributes will return the current regional settings.

If you choose to override the system regional settings (by using the `-E` and `-d` startup parameters or the corresponding `SESSION` handle attributes), you must propagate the settings to any .NET forms the application creates by instantiating a `System.Globalization.CultureInfo` object and setting its properties appropriately.

6

OpenEdge Installed .NET Controls

This appendix lists the controls that OpenEdge installs for access as visual design components using the Visual Designer of Progress Developer Studio for OpenEdge, and it describes where to find public third-party documentation on these controls and all other object types that OpenEdge installs to support the OpenEdge GUI for .NET.

Note: For information on the signatures of members provided by the classes described in this appendix, open the Class Browser view that is available in Progress Developer Studio for OpenEdge. For detailed documentation on each class, see the vendor documentation described in the following sections.

For details, see the following topics:

- OpenEdge Controls
- Microsoft .NET UI Controls
- OpenEdge Ultra Controls for .NET

OpenEdge Controls

In addition to support for the OpenEdge .NET form, `Progress.Windows.Form` (see Creating and Using Forms and Controls on page 89), the Visual Designer provides design support for the following OpenEdge .NET controls for access from the Toolbox as visual design components:

- **`Progress.Data.BindingSource`** — Allows you to bind ABL data to .NET controls, and is accessible in the Visual Designer Toolbox as the **ProBindingSource** control in the **OpenEdge** control group. For more

information on using the ProBindingSource to bind ABL data to .NET controls, see Binding ABL Data to .NET Controls on page 131.

- **`Progress.Windows.WindowContainer`** — A container control that allows you to embed the frames of an ABL window in the client area of a .NET form. You can then interact with the ABL widgets embedded in this window container using existing ABL triggers and data movement statements. For more information on using this `WindowContainer` control, see Using .NET Forms with ABL Windows on page 183.

Progress Developer Studio for OpenEdge also allows you to create a new user control by inheriting from the OpenEdge .NET class, `Progress.Windows.UserControl`. This ABL-derived user control can contain a customized set of .NET or ABL-derived controls, and you can add the new user control to the Visual Designer Toolbox as a custom design component, like any other third-party .NET control. For more information on coding definitions for ABL-derived .NET objects, see Deriving .NET classes in ABL on page 72. For more information on using this `UserControl` class in ABL, see Sample ABL-derived .NET user control on page 126.

Note: Unlike other OpenEdge-installed .NET controls, the OpenEdge `UserControl` class does not appear as a design component in the Visual Designer Toolbox. Instead, Progress Developer Studio for OpenEdge allows you to create a new ABL-derived user control in the Visual Designer using a wizard to generate the new class definition code, similar to creating a new ABL-derived form, by opening the **New** > **File** menu item and clicking the **ABL User Control** option.

Microsoft .NET UI Controls

The Microsoft .NET UI Controls represent a subset of controls available with.NET Framework 4.0.

The Visual Designer of Progress Developer Studio for OpenEdge also supports a subset of the installed Microsoft .NET UI Controls that you can access from the Toolbox as visual design components. Hover help is available that displays the class name and namespace for each control. Also, if you press **F1** after you place the control on a form within the Visual Designer, the online help opens on the Microsoft Web site.

For more information on these controls and all .NET object types available with .NET Framework 4.0, see the Class Library reference documentation at the following location:

```
http://msdn.microsoft.com/en-us/library/gg145045.aspx
```

OpenEdge Ultra Controls for .NET

The OpenEdge Ultra Controls for .NET provide extended features to the capabilities provided by controls in the Microsoft .NET Framework. They represent a subset of controls provided by Infragistics NetAdvantage for .NET 2013, Volume 1 (CLR 4.0).

The Visual Designer of Progress Developer Studio for OpenEdge also supports a subset of the installed OpenEdge Ultra Controls for .NET that you can access from the Toolbox as visual design components. Hover help is available that displays the class name and namespace for each control. Also, if you press **F1** after you place the control on a form within the Visual Designer, the online help opens on the Infragistics Web site.

For more information about these controls, see the API Reference Guide under Windows Forms at the following location:

```
http://help.infragistics.com/NetAdvantage/WinForms
```

7

Using .NET data types in ABL

The .NET type system represents a complex classification system with several different levels of sometimes interchangeable types that have varying degrees of compatibility. For more information, see the documentation for the .NET Framework SDK on MSDN. ABL supports many, but not all, of the types available in the .NET type system. For basic information on .NET types and the basis for them in ABL, see the Data types reference entry in OpenEdge Development: ABL Reference.

For details, see the following topics:

- Overview of .NET data types in ABL
- General ABL support for .NET types
- Implicit data type mappings
- Explicit data type mappings
- Assigning between ABL primitive or array types and System.Object
- Passing ABL data types to .NET constructor and method parameters
- Getting .NET data member, property, and method return values
- .NET boxing support
- .NET null values and the ABL Unknown value (?)
- Support for ADO.NET DataSets and DataTables
- Accessing and using .NET enumeration types
- Working with .NET generic types
- Accessing and using .NET arrays

Overview of .NET data types in ABL

The general categories of .NET types supported by ABL include:

- **All primitive data types** — These data types are defined by and used only by .NET languages, and represent atomic data types of the language—for example, the C# `bool` and `double` or the Visual Basic `Boolean` and `Double` data types. An ABL application accesses these primitive data types through public .NET method parameters, properties, or data members, using corresponding built-in ABL primitive types according to ABL-defined mappings—for example, the corresponding ABL `LOGICAL` and `DECIMAL`. ABL respects these mappings by doing appropriate compile-time type checking and run-time overflow checking.
- **Most object types** — These include most of the types that derive from the .NET class, `System.Object`. .NET supports two basic kinds of object types:
  - **Value types** — Objects that .NET assigns and passes by value. When .NET passes or assigns a value type object, the destination receives **a unique and separate copy of the objec**t.
  - **Reference types** — Objects that .NET assigns and passes by reference. When .NET passes or assigns a reference type object, the destination receives **a reference to the same copy of the object**.

The .NET CLR treats a small subset of value and reference object types as *aliases* for corresponding primitive data types of CLS-compliant languages. These languages can access the same values interchangeably as either the primitive data types that they define, or as the corresponding alias object types that .NET defines (for example in C#, as either `int` or `System.Int32`). ABL also interchangeably accesses (through assignment or parameter passing) either a given .NET primitive type or its alias object type as a given ABL primitive type by automatically *mapping* to that ABL type (for example, in this case, `INTEGER`). Thus, ABL views a .NET primitive type and its alias object type as a single ABL primitive type. This documentation therefore refers to both the .NET primitive data types and their corresponding alias object types as *.NETmapped data types*. Where applicable, it also refers to the object-type equivalents, themselves, as *.NETmapped object types*.

Note: .NET aliasing between primitive and object types differs from the mapping between ABL primitive and .NET types. In .NET, a given primitive type and its alias object type are treated as the same type. In ABL, the mapping between a given .NET type and its corresponding ABL primitive type involves a conversion between the two types. So, when a given value is converted between the corresponding ABL and .NET type, the precision, magnitude, or storage size might differ between the two.

.NET also supports automatic assignment and parameter passing between .NET primitive types (including their alias object types) and the .NET root object type, `System.Object`. This automatic assignment is done using a mechanism referred to as *boxing* (when the primitive type is assigned to a `System.Object`) and *unboxing* (when an appropriate `System.Object` is assigned to its corresponding primitive type). ABL also supports a similar form of boxing and unboxing for assignments and parameter passing between ABL primitive or array types and a .NET `System.Object` or .NET array object type. For more information, see .NET boxing support on page 223.

ABL treats all .NET value and reference types, other than the mapped data types, as objects similar to ABL class and interface types. The following sections describe these two kinds of object types and their basic function in ABL. For more information on ABL support for .NET types, see General ABL support for .NET types on page 207.

Value types in ABL

These are all the types that derive from the .NET `System.ValueType` class. They include a specific set of classes—all structures and enumerations. Value types differ from all other .NET object types in that they are passed to or returned from .NET by value. This has implications for managing value type objects in ABL. For more information, see Support for .NET object types on page 208.

.NET actually implements most primitive data types of .NET languages as their interchangeable subset of value types in the CLR—for example, `System.Boolean`, which implements the C# `bool`, and `System.Double`, which implements the C# `double`. An ABL application, then, accesses these .NET mapped data types through public .NET method parameters, properties, or data members using corresponding built-in ABL primitive types. For example, ABL maps its `LOGICAL` to `System.Boolean` and maps its `DECIMAL` to `System.Double`. However, unlike CLS-compliant languages whose primitive types mostly map one-to-one with their interchangeable object-type equivalents, ABL has fewer corresponding primitive types and supports its own mappings by doing appropriate compile-time and run-time type compatibility checking.

Note: The Microsoft .NET Framework documentation specifies object-type equivalents when describing data items in a language-independent manner. Where this same documentation shows language-dependent code and signatures (declarations) for .NET methods, properties, and data members, it specifies the primitive data types using the syntax of each language in order to describe the same data items.

Reference types in ABL

These include all object types that derive from `System.Object`, other than structures and enumerations, including both class and interface types. ABL supports a large subset of the available .NET reference types, as described in the following section. These include one reference type (`System.String`) that is also a .NET mapped object type, because it is also interchangeable with the CLS-compliant language primitive data types for character strings. Unlike value types, reference types are passed or returned from .NET by reference in a manner very similar to how ABL passes or returns its own class-based objects.

General ABL support for .NET types

As already noted, ABL supports access to .NET mapped data types by using corresponding ABL built-in primitive types. You can access all other supported .NET object types directly as objects, including all value types that do not map to ABL built-in primitive types. The following sections provide more information on how ABL supports these .NET types.

Support for .NET mapped object and primitive data types

ABL defines implicit mappings between .NET mapped data types and appropriate ABL built-in primitive types. For a .NET data member, property, or method return value that you assign, or a .NET method parameter that you pass, you can assign or pass an ABL primitive type that corresponds to the .NET mapped data type of the particular data member, property, method return value, or method parameter. In fact, you cannot define an object reference to or instantiate any .NET mapped object type in ABL as an object using the NEW function (classes). This means that, in ABL, you cannot directly assign or pass between ABL and the .NET context any instance of a .NET mapped data type as an object.

For example, you can never have an object reference to a `System.Boolean` or `System.Double` class instance. So, to set a `System.Boolean` or `System.Double` property, you must use a corresponding ABL `LOGICAL` or `DECIMAL` value. Note that, in ABL, you **can** assign .NET properties and data members defined as .NET mapped data types to other .NET properties and data members, and you can pass .NET properties, data members, and method return values defined as .NET mapped data types to .NET method parameters. However, ABL evaluates the assigned or passed .NET mapped data types as ABL primitive values before completing the assignments or passing the parameters.

Note: In ABL, you **can** reference .NET mapped object type names in order to access their static members, you can refer to their type names as character strings in order to create .NET arrays of mapped types, and you can define an object reference to an array of mapped types.

You can thus use ABL primitive types according to implicit data type mappings to access all supported .NET class members, including:

- Passing .NET constructor or method parameters
- Getting method return values
- Setting and getting .NET property values
- Assigning values to and from .NET data members

For more information, see Implicit data type mappings on page 210.

ABL also defines widening relationships that allow you to specify additional ABL data types for some of these implicit mappings when passing constructor or method parameters. And, because ABL does not support a unique primitive type that corresponds to every .NET mapped data type, some ABL primitive types map to multiple .NET data types. For constructor or method parameters, ABL also allows you to specify the exact .NET data type you want to map when more than one mapping is possible, especially to identify a particular method overloading to use. For more information, see Passing ABL data types to .NET constructor and method parameters on page 218.

Support for .NET object types

Other than the small set of object types that ABL does not allow you to use, or the mapped object types that ABL accesses only as ABL primitive types, you can access all .NET object types directly in ABL. For more information on those object types you cannot use, see Limitations of support for .NET classes on page 31.

ABL thus supports direct access to the following .NET object types:

- **Classes** — Reference types that ABL views and manages like ABL classes, with support for additional features that are unique to .NET classes, such as inner (nested) classes (see Referencing and instantiating .NET classes on page 34 and Accessing .NET class members on page 45). You can also define ABL classes that inherit from .NET classes (see Defining ABL-extended .NET objects on page 70).
- **Interfaces** — Reference types that ABL views and manages like ABL interfaces, with support for additional features that are unique to .NET interfaces, such as inner (nested) interfaces (see Referencing .NET class and interface types on page 37 and Accessing members of .NET interfaces on page 47). You can also define ABL classes that implement .NET interfaces (see Defining ABL-extended .NET objects on page 70).
- **Structures** — Value types that ABL views and manages similar to .NET classes. Structure types correspond to certain user-defined aggregate types in .NET languages that are supported using syntax native to each .NET language. For example, C# and C++ use the `struct` keyword to begin a user-defined type definition that .NET implements as a structure type. All such native-language user-defined types are implemented as .NET structure types in the CLR, and all .NET structure types inherit from the .NET class, `System.ValueType`. Although .NET passes or assigns structure types by value, once you reference a

structure in ABL, you can pass and assign object references to it like any other class instance. The only time a structure is passed or assigned by value within ABL is when you pass the object reference to a .NET method parameter or assign the object reference to or from a .NET object property, data member, or method return value. This can create some unique situations when exchanging structure values between the ABL and .NET context. For more information, see Support for .NET value types as objects on page 209.

- **Enumerations** — Value types that ABL views and manages similar to .NET classes, and that can represent any one of a particular enumerated subset of values from an underlying .NET mapped data type. Each value of that mapped data type subset is represented as a named member of an enumeration type. Each .NET language provides its own syntax to define and reference enumerations. All .NET enumerations inherit from the `System.Enum` structure, which inherits from the `System.ValueType` class. Like structures, ABL allows you to reference .NET enumerations as objects that .NET passes by value. For more information, see Accessing and using .NET enumeration types on page 228. Also like structures, you can encounter unique situations when exchanging enumeration object values between the ABL and .NET context. For more information, see Support for .NET value types as objects on page 209.

Note: ABL does not support a native ABL type that is similar to a .NET enumeration.

Support for .NET value types as objects

Many .NET value types map directly to ABL primitive types with no object instance used to represent them (for example, `System.Int32` and `System.Boolean`). You work directly with these .NET types using their ABL data type equivalents (in this case, `INTEGER` and `LOGICAL`, respectively). However, .NET has some value types that ABL treats as objects (for example, all structures, such as `System.Drawing.Size`, and all enumerations). .NET always passes or assigns these value type objects by value. So, you must work with these value type objects differently from reference type objects.

Note: .NET does not allow inheritance from value type objects. Therefore, you cannot define an ABL class that inherits from any .NET value type object (structure or enumeration).

Thus, when .NET passes or returns a value type object to ABL, again, it is passed by value. However within ABL, you refer to the object by reference like any other class-based object. However, any particular instance of a .NET value type object that ABL references is always a new copy of the original instance passed or returned by .NET. This means that the object reference in ABL does not point to the same copy of the value type instance that .NET has. For example:

```
USING System.Windows.Forms.* FROM ASSEMBLY.

DEFINE VARIABLE rSize1      AS CLASS System.Drawing.Size NO-UNDO.
DEFINE VARIABLE rSize2      AS CLASS System.Drawing.Size NO-UNDO.
DEFINE VARIABLE myObjectRef AS CLASS Button              NO-UNDO.

ASSIGN
  myObjectRef = NEW Button( )
  rSize1      = myObjectRef:Size
  rSize2      = myObjectRef:Size.

MESSAGE
  INTEGER(rSize1) SKIP
  INTEGER(rSize2) VIEW-AS ALERT-BOX.
```

This example assigns the object reference value of the `Size` property on `myObjectRef` to two different object reference variables (`rSize1` and `rSize2`). If `System.Drawing.Size` were a reference type, both of these variables would point to the same object instance. But since this is a value type object, each access to the `Size` property returns its value, and ABL creates a new instance for the object in the session. In contrast, when .NET returns a reference type object to ABL, no matter how many times it returns the same object, all ABL object references point to that same reference type instance.

As a result, .NET value type objects have some different usage requirements from reference type objects in ABL, as follows:

- Each time you send or receive a value type object between ABL and .NET, you cause a new instance to be created. Depending on the frequency of these assignments and the content of the value type, this can impact performance. Do as much work on the ABL side as possible before assigning a value type back to .NET. Note once you have a given .NET value type in the ABL context, you can assign affected ABL object reference variables to each other with minimal impact because they all reference the same instance.
- You cannot use value type objects in chained references. For example:

```
ASSIGN
  myObjectRef:Size:Width  = 80
  myObjectRef:Size:Height = 40.
```

This code does not set the Width and Height properties on the System.Drawing.Size object referenced by the myObjectRef:Size property. Instead, it sets the Width and Height properties on a copied instance of System.Drawing.Size that is created when the property is referenced. This is not what is intended. Therefore, the compiler generates an error or a warning on the code construct. If the value type in the middle of the chain is returned from a property or method reference, (i.e. if Size is property), the compiler generates an error. If the value type is obtained from a field reference the compiler generates a warning. In order to modify the properties (or data members) of a value type, you must reset the value type itself to a new instance. For example:

```
myObjectRef:Size = NEW System.Drawing.Size(80, 40).
```

For an example of managing .NET value type object references in ABL, see the `PointArray.p` procedure in the v

Implicit data type mappings

The following table shows the implicit mappings between .NET mapped data types and ABL primitive types. These .NET mapped data types are shown in two forms—as object types and as C# primitive data types. .NET has more mapped data types than the ABL has primitive types. So, more than one .NET type often maps to a single ABL primitive type. Where an exact mapping is not possible, ABL uses the primitive type that most completely holds the value. For these cases, and a few others, run-time errors can result if a provided value does not fit into the destination data type. Entries in the table have footnotes to indicate where any special conditions can or do occur. For more information on these conditions, see the sections following the table.

Note also that ABL supports similar implicit mappings between ABL primitive arrays and .NET arrays of mapped data types. For more information, see Implicit array mappings on page 212.

Table 10: Implicit mappings between .NET and ABL data types

Implicit .NETobject type	Implicit C#primitive data type	Implicit ABLprimitive type
System.Boolean	bool	LOGICAL
System.Byte	byte	INTEGER[1,2]
System.SByte	sbyte	INTEGER[1]
System.DateTime	–	DATETIME
System.Decimal	decimal	DECIMAL[3,4]
System.Int16	short	INTEGER[1]
System.UInt16	ushort	INTEGER[1,2]
System.Int32	int	INTEGER[4]
System.UInt32	uint	INT64[5,2]
System.Int64	long	INT64[4]
System.UInt64	ulong	DECIMAL[2,6]
System.Double	double	DECIMAL[7]
System.Single	float	DECIMAL[7]
System.Char	char	CHARACTER[8]
System.String	string	CHARACTER[4] LONGCHAR[9]

Implicit array mappings

Table 10: Implicit mappings between .NET and ABL data types on page 211 shows the implicit mappings between ABL primitive types and .NET mapped data types. Note that all of the listed ABL primitive types, except `BLOB` and `CLOB`, can also be defined as elements of an ABL array (using the `EXTENT` option). ABL similarly supports implicit mappings between ABL arrays of these primitive types and .NET one-dimensional arrays of the corresponding .NET mapped data types (*.NET arrays of mapped types*). Note that unlike ABL arrays, all .NET arrays are objects—classes whose object type names consist of the .NET type name of the array element following by a pair of square brackets (`[]`) containing a comma for each additional dimension of the array beyond one (for example, `[,]` specifies two dimensions). In ABL, you must surround the entire .NET array type name with double-quotes because the brackets and commas represent special characters in ABL names.

So, for example, you can assign an ABL `INTEGER` array to a writable .NET property defined as a `"System.Int32[]"` (or C# `int[]`), and you can assign a readable .NET property defined as a `"System.Int32[]"` (or C# `int[]`) to an ABL `INTEGER` array. In each case, ABL converts the source array type to the target array type. This same implicit array mapping also works for passing .NET method parameters.

For more information on working with and mapping .NET arrays in ABL, see Accessing and using .NET arrays on page 235.

An ABL INTEGER larger than its .NET destination

The following example raises a run-time error because the size of an ABL `INTEGER` value is incompatible with the .NET data type of a method parameter:

```
DEFINE VARIABLE iResult AS INTEGER NO-UNDO.
iResult = System.Math:Min
            (30000 AS SHORT, 60000 AS SHORT). /* Run-time error */
```

[1] An ABL INTEGER is a 32-bit number. Thus, it can hold values that are too big to store in a .NET System.Byte, System.SByte, System.Int16, or System.UInt16. Therefore, AVM raises a run-time error if an incompatible value is assigned.

[2] If you pass a negative ABL data type to an unsigned data type, the ABL virtual machine (AVM) raises a run-time error.

[3] The range of values for a .NET System.Decimal and the range of values for an ABL DECIMAL are not equivalent. In particular, an ABL DECIMAL can be a much larger positive number or a much smaller negative number than a .NET System.Decimal can represent, and a .NET System.Decimal can represent a positive or negative number with much higher precision (with more significant digits to the right of the decimal point) than an ABL DECIMAL can represent. Therefore, the AVM raises a run-time error if you assign too large or too small of an ABL DECIMAL value to a .NET System.Decimal. If you assign too precise a .NET System.Decimal to an ABL DECIMAL, with too many significant digits to the right of the decimal point, ABL truncates the least significant digits necessary to represent the value as an ABL DECIMAL.

[4] If you specify this ABL data type for a parameter of an overloaded .NET method and some overloads are distinguished by multiple .NET data types that match this same ABL data type, ABL automatically maps the corresponding .NET data type as the default match for the specified ABL data type. For example, if the method is overloaded by a parameter for both System.Decimal and System.Double, ABL automatically maps a specified ABL DECIMAL to System.Decimal, unless you indicate otherwise. For information on indicating a particular .NET data type mapping for a given ABL data type, see Indicating explicit .NET data types on page 218.

[5] An ABL INT64 is a 64-bit number. Thus, it can hold values that are too big to store in a .NET System.UInt32. Therefore, AVM raises a run-time error if an incompatible value is assigned.

[6] An ABL DECIMAL can represent a much larger number than a System.UInt64. Therefore, AVM raises a run-time error if an incompatible value is assigned.

[7] An ABL DECIMAL represents numbers up to 50 digits long. As a result, an ABL DECIMAL value cannot represent the full range of values for a .NET System.Double or System.Single. Therefore, AVM raises a run-time error if an incompatible value is assigned. Also, an ABL DECIMAL can lose precision when it is represented by a .NET System.Double or System.Single.

[8] This ABL CHARACTER mapping supports a single Unicode character.

[9] When the value comes from .NET, ABL converts from `System.String` to either CHARACTER or LONGCHAR, depending on the length of the character string. When the value comes from ABL, System.String accepts values from either CHARACTER or LONGCHAR.

In this case, the .NET static `Min( )` method compares two `System.Int16` parameters and returns the smaller value. However, an ABL `INTEGER` value of 60000 is too large for a signed 16-bit number.

Note: The `AS` option in a method parameter identifies the specific .NET data type that the method parameter takes. For more information, see Passing ABL data types to .NET constructor and method parameters on page 218.

Negative ABL values and unsigned .NET destinations

The following example raises a run-time error because of a negative ABL value passed to an unsigned .NET method parameter:

```
DEFINE VARIABLE dResult AS DECIMAL NO-UNDO.

dResult = System.Math:Max
  (75.0 AS UNSIGNED-INT64, -75.0 AS UNSIGNED-INT64). /* Run-time error */
```

In this case, the .NET static `Max( )` method compares two `System.UInt64` parameters and returns the larger value. However, an ABL `DECIMAL` value of -75.0 is incompatible with an unsigned number.

ABL DECIMAL and .NET System.Decimal

The following example raises a run-time error because a large ABL `DECIMAL` is passed to a .NET `System.Decimal` method parameter:

```
DEFINE VARIABLE dResult AS DECIMAL NO-UNDO.

dResult = System.Math:Abs
  (-22123456789012345678901234584575656.67). /* Run-time error */
```

A .NET `System.Decimal` can hold a positive or negative value with a maximum of 29 digits (`79,228,162,514,264,337,593,543,950,335`), and the specified ABL `DECIMAL` value passed to the `System.Decimal` parameter has 37 digits, including 2 digits to the right of the decimal point.

The following example truncates the least significant digits of the result from dividing one .NET `System.Decimal` value by another and storing it in the ABL `DECIMAL` variable, `dResult`:

```
DEFINE VARIABLE dResult AS DECIMAL NO-UNDO.

dResult = System.Decimal:MaxValue / System.Math:Pow(10, 28).
```

In this case, the maximum `System.Decimal` value is divided by 10^{28}. In a CLS-compliant language, the result of this calculation is a `System.Decimal` with the value of `7.922816251426433759354395033`5. However, an ABL `DECIMAL` cannot represent a value with more than 10 digits to the right of the decimal point. So, the value of the `dResult` variable is `7.9228162514`, truncating the least significant 18 digits.

Default matching ABL and .NET data types

In the implicit mappings shown in Table 10: Implicit mappings between .NET and ABL data types on page 211, four of the listed ABL primitive types can each represent two or more .NET data types. One of the .NET data types mapped to each of these four ABL data types is a *default matching* .NET data type. If you pass one of these ABL data types to a .NET method that is overloaded by that parameter using more than one of its corresponding mapped .NET data types, and one these mapped .NET data types is the default match for that ABL data type, ABL chooses this default matching .NET data type to map the specified ABL data type.

The following table shows the six basic ABL data types, each listed with its default matching .NET data type (with reference to the .NET mapped object and C# primitive data types).

ABLprimitive type	Default match(.NET object type)	Default match(C# primitive type)
CHARACTER	System.String	string
DATETIME	System.DateTime	–
DECIMAL	System.Decimal	decimal
INT64	System.Int64	long
INTEGER	System.Int32	int
LOGICAL	System.Boolean	bool

For example, if you pass an ABL INTEGER to a .NET method that is overloaded three times by one parameter with the System.Int32, System.Byte, and System.UInt16 data types, ABL calls the method that maps the ABL INTEGER to the .NET System.Int32 parameter.

Note that if the parameter is overloaded only by data types other than the default matching data type, for example System.Byte and System.UInt16, ABL raises a compile-time ambiguity error unless you explicitly indicate the .NET parameter data type you want to map. For more information on indicating the exact .NET data type for mapping an ABL data type passed to a .NET method parameter, see Indicating explicit .NET data types on page 218.

Note that this default matching applies also to mapping ABL primitive array types to .NET arrays of mapped types. For example, if you pass an ABL INTEGER array to a .NET method that is overloaded three times on one parameter by the "System.Int32[]", "System.Byte[]", and "System.UInt16[]" array types, ABL implicitly calls the method that maps the ABL INTEGER array to the .NET "System.Int32[]" parameter.

Note, also, that if the parameter is overloaded only by data types other than the default matching data type, for example System.Byte and System.UInt16, ABL raises a compile-time ambiguity error unless you explicitly indicate the .NET parameter data type you want to map using an AS data type. You can do this for mapping both ABL primitive data type and ABL primitive array type arguments. For more information on indicating the exact .NET data type for mapping an ABL type passed to a .NET method parameter, see Indicating explicit .NET data types on page 218.

An ABL INT64 larger than its .NET System.UInt32 destination

ABL maps the ABL INT64 to the .NET System.UInt32 because the unsigned value of System.UInt32 can be twice as large as the maximum signed integer value that an ABL INTEGER can represent. However, the ABL INT64 can hold values many times larger than a .NET System.UInt32. The following example raises a run-time error because iResult is too large for the System.Math.Max( ) method to compare as a System.UInt32:

```
DEFINE VARIABLE iResult AS INT64 NO-UNDO.
DEFINE VARIABLE iTest   AS INT64 NO-UNDO.
ASSIGN
  iResult = 90000000000
  iTest   = System.UInt32:MaxValue
  iResult = System.Math:Max(iTest AS UNSIGNED-INTEGER,
              iResult AS UNSIGNED-INTEGER). /* Run-time error */
```

An ABL DECIMAL larger than its .NET System.UInt64 destination

ABL maps the ABL DECIMAL to the .NET System.UInt64 because the unsigned value of System.UInt64 can be twice as large as the maximum signed integer value that an ABL INT64 can represent. However, the ABL DECIMAL can hold values many times larger than a .NET System.UInt64. The following example raises a run-time error because dResult is too large for the System.Math.Max( ) method to compare as a System.UInt64:

```
DEFINE VARIABLE dResult AS DECIMAL NO-UNDO.
DEFINE VARIABLE dTest AS DECIMAL   NO-UNDO.
ASSIGN
  dResult = 200000000000000000000
  dTest   = System.UInt64:MaxValue
  dResult = System.Math:Max(dTest AS UNSIGNED-INT64,
                            dResult AS UNSIGNED-INT64). /* Run-time error */
```

ABL DECIMAL and .NET System.Double or System.Single

The ABL DECIMAL is the ABL data type that can represent the widest range of .NET System.Double and System.Single values, and the reverse is also true. However, the ABL DECIMAL cannot represent all values of the .NET System.Double, and the .NET System.Single cannot represent all values of the ABL DECIMAL, as the examples in this section show.

The following example raises a run-time error because the maximum System.Double value represents a whole number of over 300 digits that is too large to assign to an ABL DECIMAL that can hold a maximum of 50 digits:

```
DEFINE VARIABLE dTest AS DECIMAL NO-UNDO.
dTest = System.Double:MaxValue. /* Run-time error */
```

The following example raises a run-time error because the negative value of the ABL DECIMAL (50 whole-number digits), is too large to pass to a System.Single method parameter, which can represent whole-number values of no more than 38 digits:

```
DEFINE VARIABLE dResult AS DECIMAL NO-UNDO.
DEFINE VARIABLE dTest   AS DECIMAL NO-UNDO.
ASSIGN
  dResult = -99999999999999999999999999999999999999999999999999
  dTest   = System.Math:Abs(dResult AS FLOAT).
 /* Run-time error */
```

The following example returns a value for dResult of 0.0000003469 (with rounding), because an ABL DECIMAL can only represent 10 digits to the right of the decimal point. Any remaining digits are truncated:

```
DEFINE VARIABLE dResult AS DECIMAL NO-UNDO.
dResult = System.Math:Abs
          ((34687984589034945 / System.Math:Pow(10, 23)) AS DOUBLE).
```

Note that If the ABL calculation and DECIMAL data type maintained the full precision of the .NET System.Double, the value of dResult would be 0.00000034687985 (with rounding).

Explicit data type mappings

In many, if not most, instances of access to .NET data types, ABL allows you to specify an ABL data type that implicitly maps to the specified .NET mapped data type (see General ABL support for .NET types on page 207). This is true, for example, for all direct assignments between ABL primitive and corresponding .NET mapped data types and most instances of method parameter passing between these same ABL and .NET data types (see Implicit data type mappings on page 210). However, there are certain circumstances in which you need to explicitly indicate the .NET data type to which you want to map an ABL primitive type, as shown in the following table. These circumstances include:

- When you pass ABL primitive or primitive array types to overloaded .NET method and constructor parameters where you do not want the default match (see Default matching ABL and .NET data types on page 214). For more information, see Passing ABL data types to .NET constructor and method parameters on page 218.
- When you need to manually box an ABL primitive or primitive array type to a System.Object as a .NET mapped data type or a .NET array of mapped type elements other than the default match. For more information, see .NET boxing support on page 223 and Accessing and using .NET arrays on page 235.
- When you override an inherited .NET method; when you override an inherited abstract .NET method or property; or when you implement a property or method from a .NET interface. For more information, see Defining ABL-extended .NET objects on page 70.

  Note: There is no data type mapping when you override a .NET abstract event or implement a .NET interface event because you must define the event signature with reference to a .NET delegate.

- When you substitute actual data types for type parameters in the constructed type name of a .NET generic type. For more information, see Referencing .NET generic types on page 39.

The following table shows how to specify the mapping from ABL primitive or primitive array types to .NET mapped data types or arrays of mapped type elements in a way that explicitly indicates the matching .NET data type. In general, if you want to match a .NET mapped type that is the default match for a given ABL primitive data type, you specify the ABL primitive data type as required. Otherwise, ABL provides a unique data type identifier (*AS data type*) that corresponds to a specific .NET mapped data type other than the default match. Note that in some rows, the **ABL AS data type** column is empty because the explicit .NET data type in these rows is the default (and, sometimes, the only) match for the corresponding ABL primitive (or primitive array) type. Also note that the mechanism for specifying the `AS` data type differs depending on the usage context and the data types involved.

Table 11: Explicit mappings from ABL primitive to .NET data types

Explicit .NET object type	Explicit C# primitive type	ABL primitive type	ABL AS data type
System.Boolean	bool	LOGICAL	–
System.Byte	byte	INTEGER	UNSIGNED-BYTE
System.SByte	sbyte	INTEGER	BYTE
System.DateTime	–	DATETIME[10]	–
System.Decimal	decimal	DECIMAL[10]	–
System.Int16	short	INTEGER	SHORT
System.UInt16	ushort	INTEGER	UNSIGNED-SHORT
System.Int32	int	INTEGER[10]	–
System.UInt32	uint	INT64	UNSIGNED-INTEGER
System.Int64	long	INT64[10]	–
System.UInt64	ulong	DECIMAL	UNSIGNED-INT64
System.Double	double	DECIMAL	DOUBLE
System.Single	float	DECIMAL	FLOAT
System.Char	char	CHARACTER	SINGLE-CHARACTER
System.String	string	CHARACTER[10] /LONGCHAR	–

[10] Use this ABL primitive data type to explicitly indicate a default match, with no need for an AS data type.

Assigning between ABL primitive or array types and System.Object

If you assign an ABL primitive type to a `System.Object` type reference, ABL automatically boxes the ABL data type into the `System.Object` as its default matching .NET object type. On the other hand, if you assign a `System.Object` type reference that represents a .NET mapped type to an ABL primitive type, ABL automatically unboxes the mapped type to its corresponding ABL primitive value, and attempts to assign that unboxed ABL value to the specified ABL primitive type. For more information on boxing and unboxing, see .NET boxing support on page 223. The same process works for passing ABL primitive types to method `INPUT` parameters, but **only** for parameters of .NET methods and constructors. For more information, see Passing ABL data types to .NET constructor and method parameters on page 218.

Similar boxing and unboxing rules for assignment and parameter passing apply between an ABL array and a `System.Object` type reference, including the mapping of an ABL primitive array (such as an ABL `INTEGER EXTENT`) to a .NET `System.Object` reference that represents a .NET array of mapped type elements (such as a `"System.Int32[]"`). In addition, ABL has rules for boxing and unboxing during assignment and parameter passing between an ABL array of any .NET object type and a compatible .NET array object type reference. For more information on mapping between ABL and .NET array types, see Accessing and using .NET arrays on page 235.

Note that when passing parameters to ABL routines (methods, constructors, procedures, and user-defined functions), ABL does **no** automatic boxing or unboxing. In such cases, ABL raises a compile-time error. To make these parameter passing cases work for ABL routines, you need to use the ABL BOX or UNBOX function, as appropriate. For more information, see Manual boxing on page 225.

Passing ABL data types to .NET constructor and method parameters

ABL supports two special cases for handling data types when passing ABL primitive or primitive array arguments to parameters of .NET constructors and methods:

- Syntax to indicate an explicit .NET data type when the passed ABL primitive or primitive array type can represent multiple .NET mapped data types or arrays of mapped type elements
- Data type widening between a parameter defined as a .NET mapped data type and its implicitly mapped ABL primitive type

Indicating explicit .NET data types

In Table 10: Implicit mappings between .NET and ABL data types on page 211, note that some ABL primitive types map to two or more alternative .NET data types, and one of these alternative mappings represents a default match for the given ABL primitive type (see Default matching ABL and .NET data types on page 214).

Two cases exist where the default matching data type is insufficient and you must identify the explicit .NET data type you are passing to a parameter in order for ABL to call the method as you expect:

- For .NET constructors or methods whose signatures are overloaded by alternative .NET types or arrays of mapped type elements that map to the same ABL primitive or primitive array type, you need to explicitly identify the .NET mapped data type or mapped type array element to ABL so it can invoke the correct

constructor or method overloading. For more information on constructor and method overloading in ABL, see OpenEdge Development: Object-oriented Programming.

- For a `System.Object INPUT` parameter of a .NET constructor or method, ABL automatically boxes the passed ABL primitive or primitive array type into its default matching .NET mapped type or array of mapped type elements. However, if you need the target type to be something other than the default match, you must explicitly indicate the .NET mapped data type to use for the `INPUT` ABL primitive or primitive array elements. A common example of this is the `SetValue( )` method on `System.Array`.

For ABL primitive arguments, ABL provides an `AS` option on passed .NET constructor and method parameters to indicate an explicit .NET data type for each ABL primitive type that has alternative .NET data type mappings. You indicate the explicit .NET data type by specifying a corresponding `AS` data type keyword for the `AS` option (see Table 11: Explicit mappings from ABL primitive to .NET data types on page 217). For more information on the syntax of the `AS` option for passing .NET mapped data type parameters, see Specifying .NET constructor and method parameters on page 50. Note that if the .NET data type you want to pass is the default match for the ABL primitive type you are passing, you simply pass the ABL value without the `AS` option.

For ABL primitive arrays, ABL provides a BOX function that you can invoke on an appropriate `INPUT` parameter of a .NET constructor or method for the ABL primitive array argument, and in this `BOX` function invocation, you can indicate the target .NET array of mapped types by specifying the corresponding `AS` data type keyword (see Table 11: Explicit mappings from ABL primitive to .NET data types on page 217) as a character expression. For more information on using the `BOX` function, see .NET boxing support on page 223. For `OUTPUT` parameters, you must pass an argument defined as the indicated .NET array of mapped types and assign the .NET array output parameter result to a matching ABL primitive array. However, for performance reasons, you might want to work with the output .NET array directly instead of converting it to an ABL array. For more information on assigning between ABL and .NET arrays, see Accessing and using .NET arrays on page 235.

Note, again, that if there is no overloading of a parameter, you do not have to specify an `AS` option, and the ABL data type of the argument will match any of the .NET data types supported by the implicit mappings. If you do not specify the `AS` option for an overloaded parameter, and none of the available overloadings represents the default match for the ABL primitive type you pass, ABL raises a compile-time ambiguity error.

The following example shows three different overloadings of the static `Max( )` method called on the `System.Math` class from ABL. The first overloading maps the parameters to the `INTEGER` default match, `System.Int32`. The remaining overloadings map the parameters to the .NET data type specified by the given `AS` data type:

```
DEFINE VARIABLE iResult AS INTEGER NO-UNDO.

ASSIGN
  iResult = System.Math:Max(-2147483647, 2147483647)
  iResult = System.Math:Max(65535 AS UNSIGNED-SHORT, 65 AS UNSIGNED-SHORT)
  iResult = System.Math:Max(127 AS BYTE, 12 AS BYTE).
```

Data type widening

Object-oriented programming, in support of strong typing, typically requires any argument that you pass to a constructor or method parameter to have a data type that exactly matches the data type used to define the parameter. *Widening* is a feature of many object-oriented programming languages that allows you to pass a different data type to a constructor or method than the data type that is defined for the parameter itself. Widening typically allows this, as long as the passed data type is structurally capable of handling the data that you are passing through the parameter to the data type of the destination. ABL supports widening for passing parameters to ABL class-based constructors and methods. For more information, see OpenEdge Development: Object-oriented Programming.

ABL also supports a form of widening for passing parameters to .NET constructors and methods. In this case, there are more .NET data types that can be involved in passing data to a widened data type than ABL data types in an ABL user-defined constructor or method. Thus, ABL provides special support for data type widening when passing .NET parameters involving ABL primitive and .NET mapped data types.

Note: ABL does not support widening when passing parameters between ABL primitive array types and .NET arrays of mapped types. For ABL and .NET array mappings, the element types of the source and target arrays must match exactly. For more information, see Accessing and using .NET arrays on page 235.

The following table lists .NET data types with default ABL data type mappings that support other ABL widening data types for passing primitive data to a .NET `INPUT` parameter. In general, for each ABL data type passed as an `INPUT` parameter, if another ABL data type exists that can hold a smaller value acceptable to the corresponding .NET data type, you can use that other ABL data type to pass the value. For example, where the default ABL data type mapping is `DECIMAL`, the `INTEGER` and `INT64` data types can also hold values that the .NET data type can accept for `INPUT`. For all other implicit data type mappings (see Table 10: Implicit mappings between .NET and ABL data types on page 211), you can use only the specified ABL data type to pass a .NET parameter for `INPUT`.

.NET parameter object type	C# parameter primitive type	ABL implicit mapping data type	ABL INPUT widening data type
System.DateTime	–	DATETIME	DATE
System.Decimal	decimal	DECIMAL	INTEGER, INT64
System.UInt32	uint	INT64	INTEGER[11]
System.Int64	long	INT64	INTEGER
System.UInt64	ulong	DECIMAL	INTEGER[11], INT64 [11]
System.Double	double	DECIMAL	INTEGER, INT64
System.Single	float	DECIMAL	INTEGER[12], INT64[12]

The following table lists .NET data types with default ABL data type mappings that support other ABL widening data types for passing primitive data to a .NET `OUTPUT` parameter. In general, for each ABL data type passed as an `OUTPUT` parameter, if another ABL data type exists that can accept a larger value that is passed from the corresponding .NET data type, you can use that other ABL data type to accept the value. For example, where the default ABL data type mapping is `INTEGER`, the `INT64` and `DECIMAL` data types can also accept values passed from the .NET data type for `OUTPUT`. For all other implicit data type mappings (see Table 10: Implicit mappings between .NET and ABL data types on page 211), you can use only the specified ABL data type to pass a .NET parameter for `OUTPUT`.

.NET parameter object type	C# parameter primitive type	ABL implicit mapping data type	ABL OUTPUT widening data type
System.Byte	byte	INTEGER	INT64, DECIMAL
System.SByte	sbyte	INTEGER	INT64, DECIMAL

[11] If you pass a negative ABL data type to an unsigned data type, the AVM raises a run-time error.
[12] You can lose precision if you pass an ABL `INTEGER` or `INT64` to a System.Single parameter.

.NET parameter object type	C# parameter primitive type	ABL implicit mapping data type	ABL OUTPUT widening data type
System.Char	char	CHARACTER	LONGCHAR
System.DateTime	–	DATETIME	DATETIME-TZ
System.Int16	short	INTEGER	INT64, DECIMAL
System.UInt16	ushort	INTEGER	INT64, DECIMAL
System.Int32	int	INTEGER	INT64, DECIMAL
System.UInt32	uint	INT64	DECIMAL
System.Int64	long	INT64	DECIMAL

Note that similar to passing ABL parameters to ABL class-based methods, data widening is not supported for passing INPUT-OUTPUT parameters to .NET methods. Because the data is moving in both directions, no single widening choice can work for both directions. Thus, for INPUT-OUTPUT parameters that you pass to a .NET method, you must use the appropriate ABL implicit mapping data type.

Getting .NET data member, property, and method return values

When returning an OUTPUT parameter from a .NET constructor or method, ABL always has a destination variable in which to store the value. Therefore at compile time, ABL ensures that the .NET parameter and ABL variable are type-compatible according to the OUTPUT parameter mapping tables (see Passing ABL data types to .NET constructor and method parameters on page 218). At run time, ABL then converts the .NET output data to the ABL data type of the variable.

However, ABL gets .NET data member (field), property, and method return values differently than for OUTPUT parameters, because they can appear in an expression with no specified return variable. In these cases, ABL always converts the .NET output data to the data type specified in the implicit mapping table (see Table 10: Implicit mappings between .NET and ABL data types on page 211).

The following example gets a .NET data member value:

```
DEFINE VARIABLE iResult AS INTEGER NO-UNDO.

iResult = System.Math:PI.
```

In this case, ABL converts the .NET System.Double value of System.Math:PI according to the implicit mapping to an ABL DECIMAL with the value 3.1415926535. It then attempts to assign this value to the INTEGER, iResult, which ABL allows, performing the conversion with rounding and assigning a value of 3. If iResult, instead, was attempting to receive the same value passed in an OUTPUT parameter of a .NET method, ABL would raise a compile-time type compatibility error.

The same conversion occurs when getting the method return value in the following example:

```
DEFINE VARIABLE iResult AS INTEGER NO-UNDO.

iResult = System.Math:Abs(-(System.Math:PI) AS DOUBLE).
```

Again, the following example is a similar case, where you "assign" the results of the same .NET data member and method return values passed to ABL `INPUT` parameters. So, this displays "`piResult1 = 3 and piResult2 = 3`":

```
PROCEDURE viewResults:
  DEFINE INPUT PARAMETER piResult1 AS INTEGER NO-UNDO.
  DEFINE INPUT PARAMETER piResult2 AS INTEGER NO-UNDO.

  MESSAGE "piResult1 = " piResult1 " and piResult2 = " piResult2
    VIEW-AS ALERT-BOX INFORMATION.
END.

RUN viewResults
  (System.Math:PI, System.Math:Abs(-(System.Math:PI) AS DOUBLE)).
```

Because ABL performs conversion for .NET data member, property, and method return values according to the implicit data type mappings before evaluating these return values in expressions and assignments, data precision and values can be lost consistent with the implicit data type mapping rules. For more information, see Implicit data type mappings on page 210.

If you assign a .NET property or method return value to another .NET property or data member in ABL, you have the same potential for loss of data precision or values because ABL performs the appropriate ABL data conversion before assigning the original .NET value on the right-hand side to the .NET data item on the left-hand side of the assignment. For example, this code fragment assigns the static `MaxValue` data member of the `System.Single` class to the `System.Single X` property of a `System.Drawing.PointF` object:

```
DEFINE VARIABLE rPointF AS CLASS System.Drawing.PointF NO-UNDO.

ASSIGN
  rPointF   = NEW System.Drawing.PointF( 1.0, 2.0)
  rPointF:X = System.Single:MaxValue.
```

ABL evaluates the .NET data member, `MaxValue`, as an ABL `DECIMAL` before assigning it to the .NET `X` property.

If you assign a .NET property, data member, or method return value defined as a `System.Object` to an ABL data element defined as a primitive type, the assigned `System.Object` value must be a .NET mapped object type. If the value is a .NET mapped object type, ABL unboxes it from the `System.Object` and converts it to its matching ABL primitive type before attempting to assign the value to the target ABL data element. For more information on unboxing, see .NET boxing support on page 223.

.NET boxing support

In a .NET language, like C#, you can automatically assign between a primitive data type, like `int`, and a `System.Object`. For example, .NET automatically converts (*boxes*) an `int` into a `System.Object` that contains the same `System.Int32` value. .NET also automatically converts (*unboxes*) a `System.Object` that contains a `System.Int32` into the same `int` value whenever assigning or passing method parameters between the two. ABL provides a similar form of boxing and unboxing support when assigning values or passing method parameters between an ABL primitive or array type and a corresponding .NET `System.Object` or array object type. ABL provides this boxing and unboxing support automatically for assignments, and as appropriate when passing parameters to .NET methods. However, when passing parameters to ABL routines, you must manually box and unbox as needed using built-in ABL functions. The following sections describe this ABL boxing support:

- Automatic boxing on page 223
- Manual boxing on page 225

Automatic boxing

ABL automatically performs the required boxing or unboxing operation in the following situations:

- When you assign a value between an ABL primitive and a .NET property, data member, or method return value defined as an appropriate `System.Object`, the appropriate boxing (to the `System.Object`) or unboxing (from the `System.Object`) occurs.

  Note: Assigning a `System.Object` to a .NET primitive (or equivalent object) type is invalid because of type narrowing, but ABL makes an exception when assigning a `System.Object` to an ABL primitive by automatically converting the types.

- When you assign a compatible ABL array type to a .NET property or data member defined as a `System.Object`.

  Note: ABL does not automatically convert types when assigning a `System.Object` to an ABL array.

- For a .NET method or constructor, when you pass an ABL primitive or compatible array type to an `INPUT` parameter defined as a `System.Object`.

  Note: ABL does not automatically convert types when passing these ABL types to an `OUTPUT System.Object` parameter.

- When you assign between a compatible ABL array type and a .NET array object type, or when you pass a compatible ABL array type to a .NET method or constructor parameter defined as a .NET array object type.

For example, when you assign an ABL INTEGER to a .NET data element defined as a System.Object, ABL automatically boxes the INTEGER into to the System.Object as the default matching System.Int32. However, if the .NET property or data member requires a conversion to a .NET mapped data type other than the default match, you must use the ABL BOX function (see the). The same is true of assigning an ABL INTEGER array to a System.Object: ABL automatically boxes and stores the ABL array as a one-dimensional .NET array of System.Int32 elements ("System.Int32[]"), and you can also manually box the ABL array as a .NET array of a non-default matching element type, such as an "System.Byte[]", using the BOX function. For an ABL array of any other (non-mapped) .NET object type, such as of System.Drawing.Point or Progress.Windows.Form elements, ABL boxes the ABL array to a System.Object as the specified one-dimensional .NET array, for example "System.Drawing.Point[]" or "Progress.Windows.Form[]".

Similarly, if you assign a System.Object to an ABL INTEGER data element, ABL automatically unboxes the System.Object and the result depends on the content of the System.Object (see Getting .NET data member, property, and method return values on page 221). If the System.Object is a .NET mapped data type, ABL unboxes the .NET mapped data type from the System.Object and attempts to assign the default matching ABL primitive type that results to the ABL INTEGER data element. If the ABL primitive type unboxed from the System.Object is compatible, in this case, with INTEGER, the assignment works. Otherwise, the AVM raises a run-time error.

If you assign a compatible ABL array, such as an INTEGER EXTENT or a System.Drawing.Point EXTENT, to a System.Object, ABL boxes the ABL array, in this case, as the default matching .NET array object, "System.Int32[]", or the equivalent one-dimensional .NET array object, "System.Drawing.Point[]", respectively. However, you cannot assign a similarly boxed System.Object to the equivalent ABL array, without manually unboxing the System.Object using the UNBOX function (see Manual boxing on page 225). If you want to assign an ABL primitive array and box it as a .NET array of elements other than the default match, you must assign the result of executing the BOX function for the ABL array, indicating the AS data type to the function that matches the target element type (see Manual boxing on page 225).

When passing a ABL primitive or compatible array type to a .NET method or constructor INPUT parameter defined as System.Object, ABL automatically boxes the ABL primitive or array type into the System.Object parameter (on INPUT) or unboxes the System.Object parameter as the default matching .NET type or the equivalent .NET array object type, respectively. In addition for an ABL primitive, if the System.Object parameter requires a .NET mapped data type other than the default match, you can specify the AS data type option on the ABL primitive argument in order to box the value to a specified .NET mapped type. However, for an ABL primitive array argument, if you want to box it as a .NET array of elements other than the default match, you must execute the BOX function on the parameter for the ABL array argument, indicating the AS data type to the function that matches the target element type.

For example, if you pass an ABL INTEGER to a System.Object INPUT parameter of a .NET method, by default, ABL boxes the INTEGER value as a System.Int32. If you specify the AS UNSIGNED-BYTE option when passing the INTEGER value, ABL boxes the value as a System.Byte. For more information on using the AS data type option, see Specifying .NET constructor and method parameters on page 50 and the Passing ABL data types to .NET constructor and method parameters on page 218. If you pass an ABL INTEGER array to a System.Object INPUT parameter of a .NET method, ABL boxes the ABL array as a "System.Int32[]". However, if you want to box the ABL INTEGER array as another compatible .NET array of mapped type elements (such as a "System.Byte[]"), you must execute the BOX function on the parameter for the INTEGER array, indicating the AS data type to the function. If you pass any other compatible ABL array (such as System.Drawing.Point EXTENT), ABL boxes it the equivalent one-dimensional .NET array (such as "System.Drawing.Point[]").

When you assign, or pass .NET method parameters, between compatible ABL arrays and .NET arrays, ABL is also doing an automatic boxing (from ABL non-object to .NET object type) or unboxing (from .NET object to ABL non-object type) operation. In this case, the operation follows array mapping rules that determine how an ABL array is converted to a .NET array object type (boxing) and how a .NET array object is converted to an ABL array type (unboxing). For more information on the rules for mapping between ABL and .NET arrays, see Accessing and using .NET arrays on page 235.

Note that for ABL routines of any kind (class-based methods, constructors, user-defined functions, or procedures), ABL does no automatic boxing or unboxing between ABL non-object types and .NET object types involved in parameter passing. Instead, you must either use the ABL BOX or UNBOX function (as appropriate) or assign arguments to box or unbox them appropriately before or after calling the ABL routines, as described in the following section.

Manual boxing

ABL provides a BOX function to manually convert an ABL primitive value or array to a particular .NET mapped data type or .NET array object type, and return the result as a .NET `System.Object` reference. ABL also provides an UNBOX function to manually unbox a .NET mapped data type or array object type from a `System.Object` or array object and return it as a matching ABL primitive or array type. This is the syntax for the `BOX` and `UNBOX` functions:

Syntax

```
BOX ( ABL-expression [ , AS-data-type-expression] )
UNBOX ( object-reference )
```

ABL-expression is the primitive value or array that you want to box as a .NET mapped data type or array object type. You can use *AS-data-type-expression* only for a primitive value or primitive array. This character expression indicates the .NET mapped data type or the element type of the primitive array that you want to box for *ABL-expression*. This character expression contains the `AS` data type keyword in Table 11: Explicit mappings from ABL primitive to .NET data types on page 217 that matches the .NET type you want to specify. Without *AS-data-type-expression*, the function boxes *ABL-expression* to its default matching .NET mapped data type or array of mapped type elements. The *object-reference* is a reference to a `System.Object` or .NET array object from which you want to unbox the .NET mapped data type or array object type.

You can optionally use these functions appropriately where ever ABL does automatic boxing and unboxing (see Automatic boxing on page 223). However, in certain cases, you must use these functions to box or unbox an ABL data type.

You must use the `BOX` function to manually box an ABL value or array into a .NET `System.Object` or .NET array object:

- When you assign an ABL primitive or primitive array type to a `System.Object` and you want the ABL value or array boxed as a .NET mapped data type or .NET array of mapped type elements other than the default match—for example, in order to assign an ABL `INTEGER` or `INTEGER` array to the target .NET data element as a `System.Byte` or `"System.Byte[]"`.
- When you pass an ABL primitive array type to a `System.Object INPUT` parameter of a .NET method or constructor and you want to box the ABL array as a .NET array of mapped type elements other than the default match.
- When you pass an ABL primitive or array type to a `System.Object` or .NET array object `INPUT` parameter of any ABL routine (method, constructor, procedure, or user-defined function). You must invoke the `BOX` function for the ABL data type directly on the `INPUT` parameter, casting as necessary using the CAST function. The passing of ABL array to .NET array parameters represents an array mapping. For more information, see Accessing and using .NET arrays on page 235.

You must use the `UNBOX` function to manually unbox the ABL value or array from a .NET `System.Object` or .NET array object:

- When you use a `System.Object` in an ABL expression, such as a numeric or date expression.

- When you want to assign a System.Object to a compatible ABL array.
- When you pass a System.Object or .NET array object to a compatible ABL primitive or array INPUT parameter of any ABL routine (method, constructor, procedure, or user-defined function). You must invoke the UNBOX function for the System.Object or array object directly on the ABL INPUT parameter. The passing of .NET array to ABL array parameters represents an array mapping. For more information, see Accessing and using .NET arrays on page 235.

As an example of needing the BOX function, the following code fragment raises a run-time error on an assignment statement (rcList:Item[0] = 5) because the .NET indexed property to which an ABL INTEGER is being assigned is a System.Object that requires the data type, System.Byte, which is different from the default match for an ABL INTEGER (System.Int32). In this case, the property represents the elements of a .NET array object defined to hold System.Byte element values. However, .NET does not allow you to store the System.Int32 to which ABL automatically converts the value. So, .NET raises an error when ABL attempts to complete the assignment, as shown:

```
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE rcArray AS CLASS System.Array             NO-UNDO.
DEFINE VARIABLE rcList  AS CLASS System.Collections.IList NO-UNDO.
DEFINE VARIABLE iValue  AS INTEGER                        NO-UNDO.

ASSIGN
  rcArray        = System.Array:CreateInstance
                     (TypeHelper:GetType("System.Byte"), 1)
  rcList         = rcArray

  /* This line raises a run-time error that the source type (System.Int32)
     cannot narrow to the target type (System.Byte) */
  rcList:Item[0] = 5 /* Run-time ERROR */
  iValue         = rcList:Item[0].

MESSAGE "rcList:Item[0] = " iValue VIEW-AS ALERT-BOX.
```

Note: For more information on using .NET array objects (based on the System.Array class), see Accessing and using .NET arrays on page 235. The Item property is actually an explicit member of an interface (System.Collections.IList) that System.Array implements. For more information on explicit interface members, see Accessing members of .NET interfaces on page 47.

However, the following code fragment runs to completion using the BOX function to convert the INTEGER value to a System.Byte before boxing the value in the System.Object array element:

```
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE rcArray AS CLASS System.Array             NO-UNDO.
DEFINE VARIABLE rcList  AS CLASS System.Collections.IList NO-UNDO.
DEFINE VARIABLE iValue  AS INTEGER                        NO-UNDO.

ASSIGN
  rcArray        = System.Array:CreateInstance
                     (TypeHelper:GetType("System.Byte"), 1)
  rcList         = rcArray
  rcList:Item[0] = BOX(5, "UNSIGNED-BYTE")
  iValue         = rcList:Item[0].

MESSAGE "rcList:Item[0] = " iValue VIEW-AS ALERT-BOX.
```

The second parameter of the BOX function specifies an AS data type that corresponds to the .NET mapped data type into which you want to box the value (5) specified by the first parameter. In this case, the AS data type, UNSIGNED-BYTE (specified for BOX as a character string) corresponds to a .NET System.Byte. For more information on the AS data types that you can use to specify the explicit .NET conversion for each ABL primitive type passed to the BOX function, see Explicit data type mappings on page 216.

As an example of needing the UNBOX function, the following code fragment raises a compile-time error on the MESSAGE statement because it is attempting to use a .NET System.Object directly in expressions:

```
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE rcArray AS CLASS System.Array             NO-UNDO.
DEFINE VARIABLE rcList  AS CLASS System.Collections.IList NO-UNDO.

ASSIGN
  rcArray        = System.Array:CreateInstance
                     (TypeHelper:GetType("System.Int32"), 1)
  rcList         = rcArray
  rcList:Item[0] = 10. /* Automatic boxing */

MESSAGE
  "A circle with a diameter of "             /*Compile-time Error*/
  rcList:Item[0] " has an area = "
  System.Math:PI * EXP(rcList:Item[0] / 2 , 2)
  VIEW-AS ALERT-BOX.
```

In this case, the first element of a System.Int32[] array is set to an ABL INTEGER value (10) through the default indexed property (Item) of the System.Collections.IList interface that the .NET array object implements. (Note that the assignment performs automatic boxing to assign the value to the System.Object property as a System.Int32.) When the value is accessed from the indexed property (rcList:Item[0]) directly in an expression, ABL does not automatically recognize that it needs to unbox a System.Object value in order for it to work in an expression. Thus, this code generates an incompatible data type error at compile time.

By using the UNBOX function, you tell ABL that the specified .NET expression (rcList:Item[0]) represents a System.Object that it needs to unbox and convert to an ABL data type, as in the following update to the same code fragment:

```
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE rcArray AS CLASS System.Array             NO-UNDO.
DEFINE VARIABLE rcList  AS CLASS System.Collections.IList NO-UNDO.

ASSIGN
  rcArray        = System.Array:CreateInstance
                     (TypeHelper:GetType("System.Int32"), 1)
  rcList         = rcArray
  rcList:Item[0] = 10. /* Automatic boxing */

MESSAGE
  "A circle with a diamter of "
  UNBOX(rcList:Item[0]) " has an area = "
  System.Math:PI * EXP(UNBOX(rcList:Item[0]) / 2 , 2)
  VIEW-AS ALERT-BOX.
```

This code then compiles and runs to completion, displaying a message with the area of the circle represented by the diameter stored in the .NET array. For more examples using the BOX and UNBOX functions, see the reference entry for each ABL function in OpenEdge Development: ABL Reference.

.NET null values and the ABL Unknown value (?)

In general, ABL maps the `Unknown value (?)` to the .NET `null` value. Thus, if you pass the `Unknown value (?)` as a .NET method parameter or assign it to a .NET data member or property, ABL converts it to `null`. However, where .NET defines a default value for a data type, setting a .NET data item to the `Unknown value (?)` actually sets the data item to the default value specified for the data type. This means that if you set a .NET data type that has a default value to the `Unknown value (?)`, retrieving the value of that data type returns its default value, not the `Unknown value (?)` that you assigned.

The following table lists the general classes of .NET data types that have default values and the default values that they support.

.NET data types	Default value
Integer numeric types (such as `System.Int32` or `System.Byte`)	`0`
Decimal or floating point numeric types (such as `System.Decimal` or `System.Single`)	`0.0`
Logical types (`System.Boolean`)	`no`

Support for ADO.NET DataSets and DataTables

ABL does no mapping between ADO.NET DataSets (`System.Data.DataSet`) and DataTables (`System.Data.DataTable`) and their corresponding ABL data objects, ProDataSets and temp-tables. In ABL, these .NET data objects look like any other .NET object, and you can access them accordingly.

However, ABL supports the ability to map ABL ProDataSets, temp-tables, and other forms of ABL data directly to .NET controls using an OpenEdge extension of the .NET `System.Windows.Forms.BindingSource` class, Progress.Data.BindingSource (ProBindingSource). For more information on using the ProBindingSource, see Binding ABL Data to .NET Controls on page 131.

Accessing and using .NET enumeration types

In general, an enumeration is a type whose members consist of a defined set of named constant values with a common underlying data type. For example, you might have an enumeration type `Color` defined with an underlying integer data type, and this enumeration has a member, `Blue`, with the underlying integer value of `5`. However, where .NET structures its enumerations to interoperate interchangeably as both values and as value type objects, ABL structures and manages .NET enumerations, like all class instances that it references, only as objects.

Enumerations in .NET

In .NET, an enumeration is a special kind of value type object that inherits from `System.Enum` and whose members correspond to a set of constant values with a common underlying primitive data type. Each member is denoted by a unique name, analogous to a property of a class. For example, in .NET you might have a `Color` enumeration defined in a `MyGraphics` namespace with a common underlying integer data type, where the value, `5`, is denoted by the enumeration member, `Blue`. In .NET, you reference members of an enumeration similar to members of any other .NET class, for example (in .NET notation), `MyGraphics.Color.Blue`.

Each .NET language provides its own syntax for defining enumeration types and accessing their values (members). This includes operator overloading that allows enumeration data items to be manipulated like any other data item with the same underlying primitive data type. Typically, this means that, in .NET, if you define a variable with an enumeration type, you can assign and evaluate (with casting) a compatible value of the same primitive data type to the enumeration variable.

For example, in .NET, if you define a variable with the `Color` enumeration type, you can assign the enumeration member, `Blue` to that variable, or you can cast and assign the integer value, `5`, to the same variable, both of which are equivalent. You can also perform binary operations, such as addition and subtraction, as defined for the enumeration and its underlying data type. For example, you might be able to add two enumeration members together to obtain the value of a third, as in the expression `MyGraphics.Color.Blue + MyGraphics.Color.Yellow`.

Therefore, as a value type in .NET, an enumeration type can function interchangeably with its underlying primitive data type.

.NET enumerations in ABL

ABL also views a .NET enumeration as a special kind of class, and you can reference .NET enumeration types and their members like any other .NET class—for example, `MyGraphics.Color:Blue` (see the Enumeration member access reference entry in OpenEdge Development: ABL Reference). However, unlike .NET, which manages enumerations as values, ABL only references and manages .NET enumeration types and their values as objects.

So, where .NET languages can view enumeration members as named constants of a common underlying primitive data type, ABL views the members of an enumeration class as objects that **refer to but are not equivalent to the underlying primitive type**. This means that when you define an ABL variable as a .NET enumeration type, it functions as an object reference, and you can assign it a reference to a .NET enumeration value (member). However, you cannot assign an ABL primitive value to an enumeration variable even if it is equivalent to the underlying primitive value of an enumeration member. In other words, ABL does not map between primitive values and their equivalent enumeration members, as it does with .NET mapped data types (see Implicit data type mappings on page 210).

For example (using our previously defined .NET `Color` enumeration example), the following code fragment shows an invalid assignment to an enumeration variable in ABL:

```
DEFINE VARIABLE rColor AS CLASS MyGraphics.Color NO-UNDO.

rColor = 5. /* Compile-time error */
```

Such an assignment raises a compile-time error, and ABL (unlike .NET languages) provides no way to cast such an assignment.

The following code fragment shows a valid .NET enumeration type assignment to the same `MyGraphics.Color` enumeration variable in ABL:

```
DEFINE VARIABLE rColor AS CLASS MyGraphics.Color NO-UNDO.

rColor = MyGraphics.Color:Blue.
```

Also, ABL does not support operator overloading for classes the way .NET languages do. So, you cannot directly perform operations on .NET enumeration members using the same operations supported for the underlaying data type of an enumeration.

For example, an expression like the one shown in the following ABL assignment is invalid:

```
DEFINE VARIABLE rColor AS CLASS MyGraphics.Color NO-UNDO.

rColor = MyGraphics.Color:Blue + MyGraphics.Color:Yellow.
```

However to support such operations on enumeration members, ABL supports a helper class, Progress.Util.EnumHelper, which provides methods that perform common operations on .NET enumerations.

Using the Progress.Util.EnumHelper class

The `Progress.Util.EnumHelper` class provides public static methods that accept .NET enumeration members as `INPUT` parameters in the form of `System.Enum` objects. These methods then perform common operations on the `INPUT` members and return either a `LOGICAL` value or a new instance of `System.Enum`, depending on the operation. Together, these methods support the basic operations that you can perform on a primitive type in .NET. Note that the available operations on a given .NET enumeration type depend both on its underlying primitive data type and whether it is defined with the `System.FlagsAttribute` class to allow bit-wise operations.

The following table lists these methods showing a brief signature and description of the operation that each method performs. For more information on this class and its methods, see the Progress.Util.EnumHelper class reference entry and its respective method entries in OpenEdge Development: ABL Reference.

Method	Description
`Add( enum1, enum2 )`	Adds (+) the underlying values of *enum1* and *enum2* and returns the sum as a new instance
`Subtract( enum1, enum2 )`	Subtracts (−) the underlying value of *enum1* from *enum2* and returns the difference as a new instance
`AreEqual( enum1, enum2 )`	Returns `TRUE` if the underlying values of *enum1* and *enum2* are equal (=); otherwise, returns `FALSE`
`AreNotEqual( enum1, enum2 )`	Returns `TRUE` if the underlying values of *enum1* and *enum2* are not equal (<>); otherwise, returns `FALSE`

Method	Description
`IsGreater( enum1, enum2 )`	Returns `TRUE` if the underlying value of *enum1* is greater than (>) *enum2*; otherwise, returns `FALSE`
`IsLess( enum1, enum2 )`	Returns `TRUE` if the underlying value of *enum1* is less than (<) *enum2*; otherwise, returns `FALSE`
`IsGreaterOrEqual( enum1, enum2 )`	Returns `TRUE` if the underlying value of *enum1* is greater than or equal to (>=) *enum2*; otherwise, returns `FALSE`
`IsLessOrEqual( enum1, enum2 )`	Returns `TRUE` if the underlying value of *enum1* is less than or equal to (<=) *enum2*; otherwise, returns `FALSE`
`And( enum1, enum2 )`	Applies a bit-wise AND operation between the underlying values of *enum1* and *enum2*, and returns the result as a new instance[13]
`Or( enum1, enum2 )`	Applies a bit-wise OR operation between the underlying values of *enum1* and *enum2*, and returns the result as a new instance[13]
`Xor( enum1, enum2 )`	Applies a bit-wise exclusive OR operation between the underlying values of *enum1* and *enum2*, and returns the result as a new instance[13]
`Complement( enum )`	Applies a bit-wise complement operation to the underlying value of *enum*, and returns the result as a new instance[13]

The following example performs a bit-wise OR of two `AnchorStyles` enumeration members, casts and assigns the result to the `Anchor` property, and displays that result:

```
USING System.Windows.Forms.* FROM ASSEMBLY.
USING Progress.Util.*        FROM ASSEMBLY.

DEFINE VARIABLE rButton AS Button      NO-UNDO.
DEFINE VARIABLE rEnum   AS System.Enum NO-UNDO.

ASSIGN
  rButton        = NEW Button( )
  rEnum          = EnumHelper:Or(AnchorStyles:Bottom, AnchorStyles:Right)
  rButton:Anchor = CAST(rEnum, AnchorStyles).
MESSAGE rButton:Anchor VIEW-AS ALERT-BOX.
```

[13] The enumeration type specified by the System.Enum arguments to this method must be defined using the System.FlagsAttribute class. For more information, see the .NET documentation for the specified enumeration type.

Working with .NET generic types

A previous section in this book introduces .NET generic types and describes how to reference them as constructed types in ABL by substituting the type parameters (see Referencing .NET generic types on page 39). For example, .NET supports a generic sorted list object that allows you to create a sorted list of key/value pairs, where the value of a specified type is sorted by the key of a specified type. So, to define an object reference to a .NET sorted list consisting of `System.String` keys and `System.Int16` values, you can code the following ABL statement:

```
DEFINE VARIABLE rKeyValueList AS CLASS
  "System.Collections.Generic.SortedList<CHARACTER, SHORT>" NO-UNDO.
```

In this example, the generic type name is constructed by substituting the ABL primitive type, `CHARACTER`, for the key type parameter (first *`Tparm`*) and the ABL AS data type, `SHORT`, for the value type parameter (second *`Tparm`*).

Identifying generic type parameters

To identify what type parameter or parameters are required to construct a given .NET generic type, you can begin by using the Class Browser in Progress Developer Studio for OpenEdge look up the open type name, then review the complete description for the generic type in the .NET documentation.

In .NET Framework class library documentation, you can identify a generic type by how it is listed under its namespace. Depending on the .NET Framework version, the listing for the object type might include the word "Generic" or the open type parameters in parentheses. For example, the .NET Framework 4.0 listing for the `Stack` generic type appears under the `System.Collections.Generic` namespace as follows:

```
Stack(T) or Stack<T> Class
```

See the class Library reference documentation at the following location:

```
http://msdn.microsoft.com/en-us/library/gg145045.aspx
```

Identifying constraints on generic type parameters

The complete formal declaration for a generic type can also include constraints on the .NET data types that are allowed for each of its type parameters. These constraints place limits on the choice of data types that you can substitute for a given type parameter.

If there are no constraints on a type parameter, when a generic class is compiled, the .NET compiler treats the type parameter as a `System.Object`. Therefore, the methods and properties defined in the generic class can operate on the type parameter only in ways that are allowed for a `System.Object`, which is a very limited set of operations. By constraining the type parameter, it increases the number of allowable operations to what are supported by the constraining types and all types in their inheritance hierarchy.

Therefore, generic classes and interfaces often have constraints on their type parameters. It gives the coder of the generic type more flexibility and some type checking even before the actual type is supplied in a constructed type name.

A given constraint might allow you to substitute only .NET value types for a given type parameter, which would allow you to substitute the ABL `INTEGER` or .NET `"System.Drawing.Size"` type. A type with this constraint is the `SettablePropertyValue<T>` generic class in the `Microsoft.EnterpriseManagement.Administration` namespace. Note that constraints on a generic type parameter are introduced in C# by the `where` keyword (shown in bold):

```
public sealed class SettablePropertyValue<T> where T : struct
```

For another example, this is a C# generic class type that constrains the `TEmployee` type parameter to be replaced by an `Employee` object or any object that inherits from `Employee`:

```
public class GenericList<TEmployee> where TEmployee : Employee
```

A .NET generic type parameter can also be defined with multiple constraints to identify a particular subset of .NET types that you can substitute for a given type parameter in a constructed type name. This example shows the C# definition for the `System.Nullable<T>` generic structure type:

```
public struct Nullable<T> where T : struct, new()
```

Thus, `System.Nullable<T>` defines two constraints on the type parameter `T`, where the .NET type you substitute for `T` must be a value type with a default constructor (a constructor with no parameters).

The following table lists every possible type parameter constraint using C# syntax, where the `where` keyword identifies one or more constraints on some type parameter, *`Tparm`*, and the description shows one or more ABL constructed type examples that satisfy each constraint.

Constraint	Description

`where` *`Tparm`* `: struct`	*`Tparm`* must be a value type, and can be any value type except a nullable type (`System.Nullable`).
	For example, valid ABL references to a generic type, `SomeGeneric<`*`Tparm`*`>`, with this constraint can be:
	`"SomeGeneric<SHORT>"`
	`"SomeGeneric<System.Drawing.Point>"`

`where` *`Tparm`* `: class`	*`Tparm`* must be a reference type, which can be any class, interface, delegate, or array type. For example, a valid ABL reference to a generic type, `SomeGeneric<`*`Tparm`*`>`, with this constraint can be: `"SomeGeneric<System.Windows.Forms.Button>"`

Constraint	Description
`where Tparm : new()`	*Tparm* must have a default public constructor (without parameters) For example, a valid ABL reference to a generic type, `SomeGeneric<Tparm>`, with this constraint can be: `"SomeGeneric<System.Windows.Forms.Label>"`
`where Tparm : BaseClassName`	*Tparm* must be, or derive from, the class specified by *BaseClassName*. For example, a valid ABL reference to a generic type defined as `SomeGeneric<TControl> where TControl : System.Windows.Forms.Control` can be: `"SomeGeneric<System.Windows.Forms.ButtonBase>"`
`where Tparm : InterfaceName`	*Tparm* must be, or must implement, the interface specified by *InterfaceName*. The constraining interface can also be generic. For example, a valid ABL reference to a generic type defined as `SomeGeneric<TICollection> where TICollection : System.Collections.ICollection` can be: `"SomeGeneric<System.Collections.Queue>"`
`where Tparm : Uparm`	The type argument that you substitute for *Tparm* must be, or must derive from, the type argument that you substitute for another type parameter, *Uparm*. This is called a naked type constraint. For example, a valid ABL reference to a generic type defined as `SomeGeneric<TValue1, TValue2> where TValue1 : TValue2` can be: `"SomeGeneric<System.Windows.Forms.ButtonBase, System.Windows.Forms.Control>"`
`where Tparm : constraint [ , constraint ] ...`	*Tparm* can by constrained by one or more of the previously listed constraints (*constraint*), as long as the `new()` constraint is listed last. For example, a valid ABL reference to a generic type defined as `SomeGeneric<TValue> where TValue : class, new()` can be: `"SomeGeneric<System.Windows.Forms.Label>"`

Accessing and using .NET arrays

All .NET arrays are reference type objects that inherit from the .NET `abstract` class, `System.Array`, as shown in the following ABL class hierarchy:

```
Progess.Lang.Object   <------------------------------ ABL root class
  System.Object       <------------------------------ .NET root class
    System.Array      <------------------------------ .NET array base class
      "ElementType[]" <------------------------------ .NET array class
```

A .NET array object can represent an array of any .NET value or reference type, including another .NET array type, and with any number of dimensions (starting at one, of course).

The type name of any .NET array type, regardless of extent, is the type name of its elements (*`ElementType`*) appended with a pair of square brackets (`[]`). If the brackets are empty, the type name specifies a one-dimensional array. If the brackets contain commas (for example, `[,,]`), the type name specifies an additional dimension for each comma. In ABL, you must also surround this type name within quotes to allow the special characters (`[,]`) in the name, as in the example class hierarchy (`"`*`ElementType`*`[]"`).

For .NET arrays of a primitive type (for example, C# `int`), the same array type can be represented using the primitive type in a given .NET language (for example, C# `int[]`) or the equivalent object type alias (for example, `System.Int32[]`). (For information on the .NET object type aliases for .NET primitive types, see Table 10: Implicit mappings between .NET and ABL data types on page 211.) However, where in C# you can reference the same array type as `int[]` or as `System.Int32[]`, in ABL you can only reference this array type as `"System.Int32[]"`, using the alias object type name. In fact, where in ABL you **must** access any .NET scalar primitive type (for example, `System.Int32`) as its equivalent ABL primitive type (for example, `INTEGER`), you can **only** reference a .NET primitive type array as a .NET array object in ABL. For example, you **cannot** reference a `"System.Int32[]"` array type in ABL as `"INTEGER[]"`, but only as `"System.Int32[]"`.

You can similarly reference the type of any .NET array object, such as `"System.Drawing.Point[]"`, which is a one-dimensional array of .NET graphical point objects, or `"System.Drawing.Point[,,]"`, which is a three-dimensional array of the same object type. .NET actually supports a variety of different kinds of array object types, including jagged arrays, that you can access in ABL. For more information, see the "Arrays Tutorial" in the *C# Programmer's Reference* on MSDN.

Note that you do not specify the extent (size) for each dimension of a .NET array object as part of its type name. Instead, these values are set when the array object, itself, is created, whether in the .NET or ABL context.

The following sections provide more information on accessing and using .NET arrays in ABL:

- Accessing .NET arrays on page 235
- Array mapping: conversion between ABL arrays and .NET arrays on page 238
- Array assignment on page 239
- Example: Mapping an ABL array to a .NET array on page 245
- Example: Accessing a .NET array on page 246

Accessing .NET arrays

The first thing to note about accessing .NET arrays is that, by default, .NET array dimensions use zero (0)-based indexing as compared to ABL arrays, which always use one (1)-based indexing.

Caution: This means that when working together with ABL and .NET arrays, ABL arrays are always indexed from 1 to the array size while .NET array dimensions are typically indexed from 0 to the dimension size minus 1.

Note: .NET allows you to specify the base index value for dimensions of any .NET array that you create. However, in practice, almost all .NET arrays are created to use the zero-based index default.

You can access a .NET array created in the .NET context, like any other .NET object, by using its object reference directly from a .NET class member, or by defining an ABL object reference variable or property and assigning the .NET class member to it.

You can also instantiate a .NET array object in ABL, but **not** by using the NEW function or statement, as for other .NET objects. Recall that the base .NET class for all array objects is `System.Array`. This class also provides the mechanism you must use to create and manage instances of all .NET array classes. In other words, `System.Array` is both the base class and the factory for creating and managing .NET arrays of all types. When you use `System.Array` to create a .NET array instance, it creates an instance of the specified .NET array type, even if you assign the object reference that you create to a data element of type `System.Array`. You can continue to use the `System.Array` data element to access elements of the created array type. You can also cast the `System.Array` instance as necessary to make it type compatible with a method parameter or property assignment.

Thus, the `System.Array` class provides a variety of public static and instance members that you can use to create and manage .NET array objects. The basic `System.Array` members for working with array objects include the:

- **`CreateInstance( )` static (factory) method** — Creates and returns an instance of `System.Array` for a given element type. This method is overloaded to create arrays from one to any number of dimensions. The first parameter of this method is a `System.Type` that describes the object type of each element in the array contents. You can provide this element type as the value returned by one of the following `GetType( )` methods:
  - OpenEdge static Progress.Util.TypeHelper:GetType( ) method
  - Microsoft .NET static `System.Type:GetType( )` method, as long as the type resides in the .NET core assembly (`mscorlib.dll`)
  - .NET `GetType( )` method on an instance of the class that the array contains

  All three `GetType( )` methods return an appropriate `System.Type` object that corresponds to a given object type name specified as a string:

Syntax

```
System.Array System.Array:CreateInstance
  (class-type-or-instance:GetType( [type-name-string] ), dimensions)
```

The *`class-type-or-instance`* can be one of the following, depending on how you obtain the element type:

- Progress.Util.TypeHelper
- `System.Type`
- An object reference to an instance of the class that can be stored as an element of the array

Note that you specify the type name (*type-name-string*) using a fully qualified type name and only when using the `GetType( )` static method on `Progress.Util.TypeHelper` or `System.Type`. The instance `GetType( )` method takes no parameters. The *dimensions* represents one or more parameters that define the dimensions of the array.

Note: If you call the `GetType( )` method on `Progress.Util.TypeHelper` in order to reference a type that does not reside in the .NET core assembly, you must include the assembly where the type resides in your working assembly references file (`assemblies.xml`). For more information on assembly references files, see Identifying .NET assemblies to ABL on page 35.

When `CreateInstance( )` creates the specified array, it initializes the array elements, depending on the type. For reference types, it initializes each element to a `null` reference. For value types, it initializes each element to its default or 0 value.

- **`SetValue( )` instance method** — Stores an object reference at the specified location in the array, overloaded depending on the number of dimensions:

```
VOID array-object-ref:SetValue( element, index-info )
```

  The *array-object-ref* is a reference to an array object instance. The *element* is an ABL primitive value or a reference to an object you want to store as an element of the array. The *index-info* represents one or more parameters that identify the element location, depending on the array dimensions.

  Note: The *element* parameter is defined as `System.Object`. So, if *element* is an ABL primitive type, and you want to store a .NET mapped data type other than the default match for that ABL data type, you must specify the `AS` option with the `AS` data type that matches the explicit .NET mapped type (see Passing ABL data types to .NET constructor and method parameters on page 218).

- **`GetValue( )` instance method** — Returns an object reference to the element type from the specified location in the array, overloaded depending on the number of dimensions and extents:

```
System.Object array-object-ref:GetValue( index-info )
```

  The *array-object-ref* is a reference to an array object instance. The *index-info* represents one or more parameters that identify the element location, depending on the defined dimensions of the array and their extents. Typically, you want to cast the return value to the type of object that the array stores.

- **`Item` default indexed property** — This property, defined as `System.Object`, provides indexed read and write access to each element of an array object, similar to how you use a subscript to access an ABL array. `Item` is actually an explicit interface member of the `System.Collections.IList` interface, which `System.Array` implements. If you cast an array object to this interface type, you can then read or write all of the array elements using the default property indexer directly on the `IList` object reference, as in the following example:

```
USING Progress.Util.* FROM ASSEMBLY.

DEFINE VARIABLE strArr AS CLASS System.Collections.IList NO-UNDO.

strArr    = CAST(System.Array:CreateInstance
                 (TypeHelper:GetType("System.String"), 2),
```

```
                  System.Collections.IList)

strArr[1] = "Spinach".
MESSAGE strArr[1] VIEW-AS ALERT-BOX.
```

This code fragment defines `strArr` as an object reference to the `IList` interface, and it casts a new string array object reference to this interface before assigning that object reference to `strArr`. It then assigns a string, `"Spinach"`, to the second element of the array (`strArr[1]`, again, zero-based) and reads the same element to display the string in a message. For more information on default indexed properties, see Accessing .NET indexed properties and collections on page 52, and for more information on explicit interface members, see Accessing members of .NET interfaces on page 47.

For more .NET information on these methods and other members of `System.Array`, see the Class Library documentation of the Microsoft .NET Framework SDK. For an on-line reference, see OpenEdge Installed .NET Controls on page 201.

For a working example of accessing a .NET array using `System.Array` mechanisms, see Example: Accessing a .NET array on page 246.

Note: When you return an ABL object reference from a specified .NET array element, the relationship of that object reference to the object stored in the array element depends on whether the element stores a value type object or a reference type object. If it is a value type object, the ABL object reference points to a separate copy of the object stored by the array element. If it is a reference type object, the ABL object reference points to the same copy of the object stored by the array element. Thus, you must manage and update members of the object stored by the array element differently, depending on the object type. For more information, see Support for .NET object types on page 208.

Array mapping: conversion between ABL arrays and .NET arrays

ABL supports a mapping between compatible .NET and ABL arrays when you assign between them, which converts one kind of array to the other. Because ABL arrays are always one-dimensional arrays, array mapping works only with one-dimensional .NET arrays.

You might want to convert a .NET array to an ABL array, for example, if you want all the .NET array data to end up in the ABL context as an ABL array, where you can use native ABL array mechanisms to access it, for example to write a report. On the other hand, you might want to convert an ABL array directly to a .NET array, when you need to initialize the .NET array completely from all the data in the ABL array. You would then assign the ABL array to a .NET array property or pass the ABL array to an array parameter of a .NET method.

In either case, whenever you assign between an ABL array and a .NET array, a *deep copy* of the array elements occurs. When you assign a .NET array to an ABL array, ABL copies all of the .NET array elements into the elements of the specified ABL array variable, property, or routine parameter. When you assign an ABL array to a .NET array object, ABL instantiates a new .NET array object, copies all of the ABL array elements to it, and assigns its object reference to the specified .NET data member, property, or method parameter.

Because assignments between ABL and .NET arrays always copy the entire array, you might want to avoid doing so until you must move the array from one context to the other. Otherwise, where updates occur to only a few elements, you might prefer to apply these updates directly to the context (ABL or .NET) where they apply. Also, note that if you are simply passing a .NET array from one .NET object to another, you do not need to copy it first to an ABL array.

Note that when assigning a .NET array to an ABL array, the ABL array must have exactly the same number of elements as the .NET array or the ABL array must be indeterminate, in which case the .NET assignment fixes the number of elements in the ABL array. However, when assigning an ABL array to a .NET array, the number of elements do not matter because a new .NET is created and the target .NET array reference points to the new array. This is different from assigning to ABL arrays, where each element of the target ABL array is replaced.

When mapping between an ABL array and a .NET array, the data types of the array elements in the two arrays must be compatible. This type compatibility involves the following basic rules:

- In general, a mapped ABL array **cannot** contain either pure ABL object elements (such as `Progress.Lang.Object`) or ABL-extended .NET object elements (see Defining ABL-extended .NET objects on page 70).
- If the .NET array is an array of mapped data types, the ABL array must contain elements of an ABL primitive type that maps (without widening) to the element type of the .NET array. For more information on how ABL primitive types map to .NET data types, see Implicit data type mappings on page 210.
- If the element type of one array is a .NET value type (other than a mapped data type), the element type of the other array (.NET or ABL) must be the same identical .NET value type, for example, `System.Drawing.Size`.
- For arrays containing .NET reference type elements, the two arrays are compatible if the element type of the source array is in the class hierarchy of the element type of the target array, for example a source array element type of `System.Windows.Forms.Button` and a target array element type of `System.Windows.Forms.Control`.
- ABL does **no** automatic unboxing from a `System.Object` scalar to a target ABL array type. ABL does do automatic boxing for direct assignments and parameter passing from an ABL array type to a target `System.Object` scalar, but only for .NET methods and constructors (see .NET boxing support on page 223).
- ABL does not box twice. That is, if ABL boxes or unboxes between a .NET array object and an ABL array, it does not also box or unbox the elements. So for example, it does not convert between a .NET `"System.Object[]"` and an ABL `INTEGER` array.

Any incompatibility between the types of the source and target arrays in an array assignment raises a compile-time error.

For an example of mapping .NET and ABL arrays, see Example: Mapping an ABL array to a .NET array on page 245. The following section describes how array assignments work and some examples of array assignments that are valid and invalid.

Array assignment

An array assignment occurs either when you assign one array to another using an assignment (=) statement or when you pass an array as a routine parameter. In general, the rules for array assignment are different for assignment between ABL arrays compared to assignment between .NET arrays. For more information on the rules for ABL array assignment, see the assignment (=) statement and data types reference entries in OpenEdge Development: ABL Reference, and also see the Web paper, *ABL Data Types in OpenEdge Release 10*. The rules for .NET array assignment follow the general object-oriented rules for assigning object references, where the target type must be the same type as or higher in the same class hierarchy than the source type. The same rules apply to the elements of .NET source and target arrays, with an exception for arrays containing value type elements. For .NET arrays that contain value type elements, the source and target arrays must contain elements of an identical value type, such as `System.Drawing.Size` or `System.Int32`. In all cases, assignments between .NET arrays are object reference assignments (no array elements copied), while assignments between ABL arrays are always deep copies, with all the elements copied from one array to the other.

In a mixed array assignment between ABL and .NET arrays, the target array type (ABL or .NET) dictates the basic assignment rules. However, any assignment that includes an ABL array as either the source or the target causes a deep copy of the source array to the target. In the case of a .NET target array, ABL also creates a new .NET array to hold the elements of the ABL source array. In addition, the behavior of the assignment can be further affected by the element types of the source and target arrays (see Array mapping: conversion between ABL arrays and .NET arrays on page 238).

ABL enforces a special restriction in assignments to a `Progress.Lang.Object`. You cannot assign any ABL array type to a `Progress.Lang.Object`, even if the ABL array contains .NET object elements. In general, because ABL arrays are not objects themselves and ABL supports no automatic boxing to `Progress.Lang.Object`, there is no way to assign an ABL array to this (or any) pure ABL object type.

The following two tables list some common examples of assignments between different source and target array types, where each table focuses on either ABL or .NET target types. The listed .NET type names are shown without quotes and assume the following `USING` statements:

```
USING System.Windows.Forms.*
USING System.Drawing.*
```

Also note that `System.Drawing.Size` and `System.Drawing.Color` are .NET value types.

The following table contains sample array assignments where the target is a given ABL array type and the source is a given .NET or ABL object type. The last column indicates the validity (compatibility) of the assignment (Valid or Invalid) and additional comments. For .NET source array types, assume that the number of elements is identical to the number of elements in the target ABL array type, unless the target ABL array has an indeterminate extent, in which case a compatible source array assignment fixes the extent of the target ABL array.

ABL target type	Source type	Validity — Comments
`System.Object EXTENT`	`Control[]`	**Valid** — Class type elements assigned to super class elements
`System.Object EXTENT`	`System.Boolean[]`	**Invalid** — ABL rules: boxing not supported on array elements during array conversion
`Progress.Lang.Object EXTENT`	`System.Int32[]`	**Invalid** — ABL rules: no boxing supported for `Progress.Lang.Object` elements
`Progress.Lang.Object EXTENT`	`Button[]`	**Valid** — ABL rules: .NET class type elements assigned to ABL root class elements
`Progress.Lang.Object EXTENT`	`Size[]`	**Valid** — ABL rules: .NET value type (`Size`) treated just like any class type assigned to ABL root class elements
`Size EXTENT`	`Size[]`	**Valid** — Elements of same object type assigned to each other

ABL target type	Source type	Validity — Comments
`Control EXTENT`	`Button[]`	**Valid** — Class type elements assigned to super class elements
`Button EXTENT`	`Control[]`	**Invalid** — Attempt to assign class type elements to subclass elements
`INTEGER EXTENT`	`System.Object`	**Invalid** — ABL rules: automatic unboxing and conversion from a `System.Object` to an ABL array not supported, even if the .NET root class object reference represents a compatible .NET array object
`INTEGER EXTENT`	`System.Array`	**Invalid** — ABL rules: automatic unboxing and conversion from a `System.Array` to an ABL array not supported, even if the base class reference for .NET arrays represents a compatible .NET array object
`INTEGER EXTENT`	`Progress.Lang.Object`	**Invalid** — ABL rules: automatic unboxing and conversion from a `Progress.Lang.Object` to an ABL array not supported, even if the ABL root class object reference represents a compatible .NET array object
`INTEGER EXTENT`	`System.Int16[]`	**Valid** — ABL mapping between an ABL primitive array and a compatible .NET array of mapped types
`CHARACTER EXTENT`	`System.Int32[]`	**Valid** — ABL rules: For procedure parameters only, ABL weakly typed mapping supported between source and target ABL array types (converted from .NET source) **Invalid** — Incompatible mapping between ABL primitive and .NET arrays of mapped types for direct assignment or .NET method and ABL user-defined function parameter passing

The following table contains sample array assignments where the target is a given .NET or ABL object type and the source is another ABL or .NET type. The last column indicates the validity (compatibility) of the assignment (Valid or Invalid) and any additional comments. Note that when assigning to a .NET array, the number of elements in source and target arrays can be different, because the target is always assigned an object reference that points to the source array, whatever its extent.

.NET object or ABL root class target type	Source type	Validity — Comments
`System.Object`	`Button EXTENT`	**Valid** — .NET rules: ABL maps the source to a `Button[]` array that .NET assigns to the super class object reference
`System.Object`	`CHARACTER EXTENT`	**Valid** — ABL rules: For direct assignments and passing parameters to .NET methods and constructors, ABL boxes a `System.String[]` in a `System.Object` and assigns it to the target `System.Object` object reference**Invalid** — For passing parameters to ABL routines, boxing to a `System.Object` not supported
`System.Object`	`MyABLClass EXTENT`	**Invalid** — .NET and ABL rules: Cannot map an ABL array of ABL objects to an array object that .NET can assign
`System.Array`	`Button EXTENT`	**Valid** — .NET rules: ABL maps the source to a `Button[]` array that .NET assigns to the super class object reference
`System.Array`	`Color EXTENT`	**Valid** — .NET rules: ABL maps the source to a `Color[]` value type array that .NET assigns to the super class object reference
`System.Array`	`INTEGER EXTENT`	**Valid** — .NET rules: ABL maps the source to a `System.Int32[]` array that .NET assigns to the super class object reference
`System.Array`	`Progress.Lang.Object EXTENT`	**Invalid** — .NET and ABL rules: Cannot map an ABL array of ABL objects to an array object that .NET can assign
`Progress.Lang.Object`	`Control EXTENT`	**Invalid** — ABL rules: Cannot assign an ABL array of any type to a `Progress.Lang.Object` object reference

.NET object or ABL root class target type	Source type	Validity — Comments
`Progress.Lang.Object`	`Button[]`	**Valid** — .NET and ABL rules: Assigns the .NET array object to the super class object reference
`Progress.Lang.Object`	`INTEGER EXTENT`	**Invalid** — ABL rules: Cannot box an ABL primitive array in a `Progress.Lang.Object`
`Progress.Lang.Object`	`Progress.Lang.Object EXTENT`	**Invalid** — ABL rules: Cannot assign an ABL array of any type to a `Progress.Lang.Object` object reference
`System.Object[]`	`Form EXTENT`	**Valid** — .NET rules: ABL maps the source to a `Form[]` array that .NET assigns to the super class object reference of a .NET array of super class reference type elements
`System.Object[]`	`Size EXTENT`	**Invalid** — .NET rules: ABL maps the source to a `Size[]` array object, but .NET can only assign an array of value type elements to an array reference of identical value type elements
`System.Object[]`	`MyABLClass EXTENT`	**Invalid** — .NET and ABL rules: Cannot map an ABL array of ABL objects to an array object that .NET can assign
`System.Object[]`	`DECIMAL EXTENT`	**Invalid** — .NET rules: ABL maps the source to a `System.Decimal[]` array, but .NET can only assign an array of value type elements to an array reference of identical value type elements
`Control[]`	`TextBox EXTENT`	**Valid** — .NET rules: ABL maps the source to a `TextBox[]` array that .NET assigns to the super class object reference of a .NET array of super class reference type elements
`Button[]`	`Button EXTENT`	**Valid** — .NET rules: ABL maps the source to a `Button[]` array that .NET assigns to the object reference of an identical array type

.NET object or ABL root class target type	Source type	Validity — Comments
`Button[]`	`Control EXTENT`	**Invalid** — .NET rules: ABL maps the source to a `Constrol[]` array, but .NET cannot assign an array of object types to an array of subclass object types
`Size[]`	`Size EXTENT`	**Valid** — .NET rules: ABL maps the source to a `Size[]` array that .NET assigns to an array reference of identical value type elements
`System.Int32[]`	`INTEGER EXTENT`	**Valid** — .NET rules: ABL maps the source to an implicitly matching `System.Int32[]` array that .NET assigns to an array reference of identical value type elements
`System.Int16[]`	`INTEGER EXTENT`	**Valid** — .NET rules: ABL maps the source to an implicitly matching `System.Int16[]` array that .NET assigns to an array reference of identical value type elements
`System.Decimal[]`	`INTEGER EXTENT`	**Invalid** — .NET rules: ABL cannot map the source to an implicitly matching .NET array type where the value type (`System.Decimal`) of the source and target array elements match
`System.DateTime[]`	`DATE EXTENT`	**Valid** — .NET rules: ABL maps the source to an implicitly matching `System.DateTime[]` array that .NET assigns to an array reference of identical value type elements
`System.String[]`	`LONGCHAR EXTENT`	**Valid** — .NET rules: ABL maps the source to an implicitly matching `System.String[]` array that .NET assigns to the object reference of an identical array type

Example: Mapping an ABL array to a .NET array

The following example shows how you might access a .NET one-dimensional array using an ABL array:

```
DEFINE VARIABLE rStrFormat AS CLASS System.Drawing.StringFormat NO-UNDO.

/* Define and initialize ABL array to pass to .NET */
DEFINE VARIABLE dTabs AS DECIMAL EXTENT 4
  INITIAL [20.0, 15.0, 10.0, 5.0] NO-UNDO.

rStrFormat = NEW System.Drawing.StringFormat( ).

/* Pass an ABL array to a .NET System.Single array parameter */
rStrFormat:SetTabStops(0.0, dTabs).
```

The .NET `SetTabStops( )` method takes a one-dimensional `System.Single` array as its second `INPUT` parameter. The example passes an ABL four-element array of `DECIMAL` values as required by the implicit mapping to `System.Single`.

The following example expands on the ABL array example in the previous section by invoking the `StringFormat:GetTabStops( )` instance method to return the tab stops that were previously set using the `SetTabStops( )` method. In this case, you do not need to explicitly create the .NET array object, because the `GetTabStops( )` method returns one, as shown:

```
DEFINE VARIABLE rStrFormat AS CLASS System.Drawing.StringFormat NO-UNDO.

/* Define and initialize ABL array to pass to .NET */
DEFINE VARIABLE dTabs AS DECIMAL EXTENT 4
  INITIAL [ 20.0, 15.0, 10.0, 5.0 ] NO-UNDO.

/* Define reference to a .NET System.Single array returned from .NET */
DEFINE VARIABLE rTabsOut AS CLASS "System.Single[]" NO-UNDO.

/* Define ABL array to hold values returned in .NET array */
DEFINE VARIABLE dTabsOut AS DECIMAL EXTENT 4        NO-UNDO.

DEFINE VARIABLE dTabOffset AS DECIMAL NO-UNDO. /* Unused OUTPUT parameter */
DEFINE VARIABLE idx        AS INTEGER NO-UNDO.

rStrFormat = NEW System.Drawing.StringFormat( ).

/* Pass an ABL array to a .NET System.Single array parameter */
rStrFormat:SetTabStops( 0.0, dTabs ).

/* Return a .NET System.Single array
rTabsOut = rStrFormat:GetTabStops( OUTPUT dTabOffset ).

DO idx = 1 TO 4:
  dTabsOut[idx] = UNBOX(rTabsOut:GetValue( idx - 1 )).

  MESSAGE "Tab stop" idx "=" dTabsOut[idx] VIEW-AS ALERT-BOX INFORMATION.
END.
```

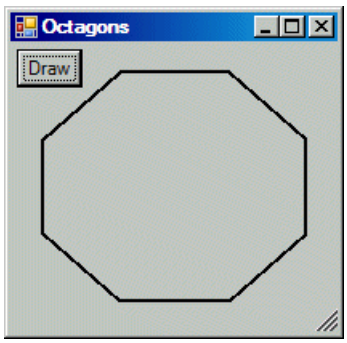
The example defines `rTabsOut` as an array object of type "`System.Single[]`" corresponding to the primitive data type array (C# `float[]`) that the `GetTabStops( )` method is defined to return. Because the array is returned as the method's return type, ABL does not automatically map it to an ABL `DECIMAL` array, but to the specified array object. The code must therefore access the array elements as objects also, and must individually convert them to the ABL `DECIMAL` data type in order to work with the element values. The ABL built-in `UNBOX` function (see .NET boxing support on page 223) allows the example to extract the underlying mapped data type from the .NET `System.Object` returned by the `GetValue( )` method for each array element. Note also the use of 1-based indexing for `dTabsOut` (the ABL array) and 0-based indexing for `rTabsOut` (the .NET array).

Example: Accessing a .NET array

This manual first introduces the use of .NET array objects in ABL with the `EventHandlers.p` example described in Accessing and Managing .NET Classes from ABL on page 29 to demonstrate .NET event handling (see Event handling example on page 62).

The following procedure, `PointArray.p`, is a simpler version of the same application, which handles fewer .NET events, in order to focus on array handling, in this case, using an array of `System.Drawing.Point` objects used to draw a regular octagon in a graphic region, as shown in the following figure.

Figure 19: Octagon displayed by PointArray.p



From a calculation based on the dimensions of the current client area in a .NET form, the `Point` objects of the array specify vertices between the sides of the octagon (determined by an `AdjustOctagon( )` user-defined function). This procedure then calculates and re-displays the octagon from the point array each time you click the **Draw** button. For information on locating and running this sample, see Example procedures.

The procedure begins by defining a `System.Array` object as a point array, a single `Point` object used to set elements of this array, and the basic user interface elements to display the octagon. In particular, the bold code shows the definition and instantiation of the point array object with the eight points required to draw an octagon. This code instantiates a new `Point` object and instantiates the array object by calling `System.Array:CreateInstance( )` with the element type parameter value obtained using the `Point` object instance just created. Note that `System.Drawing.Point` is a value type (structure). So, .NET instantiates the array with default (0-valued) `Point` objects stored in each element.

```
USING System.Windows.Forms.*  FROM ASSEMBLY.
USING Infragistics.Win.Misc.* FROM ASSEMBLY.

DEFINE VARIABLE rPointArray AS CLASS "System.Drawing.Point[]" NO-UNDO.
DEFINE VARIABLE rArray      AS CLASS System.Array             NO-UNDO.
DEFINE VARIABLE rPoint      AS CLASS System.Drawing.Point     NO-UNDO.
DEFINE VARIABLE rDrawBtn    AS CLASS UltraButton              NO-UNDO.
DEFINE VARIABLE rForm       AS CLASS Progress.Windows.Form    NO-UNDO.
DEFINE VARIABLE sqrt2       AS DECIMAL                        NO-UNDO.
DEFINE VARIABLE idx         AS INTEGER                        NO-UNDO.

FUNCTION AdjustOctagon RETURNS "System.Drawing.Point[]" FORWARD.

/* Create .NET Point array and a Point object to set its element values */
ASSIGN
  rPoint      = NEW System.Drawing.Point(0, 0)
  rArray      = System.Array:CreateInstance(rPoint:GetType( ), 8)
  rPointArray = CAST(rArray, "System.Drawing.Point[]").
```

Note: This example casts the `System.Array` reference to a `System.Drawing.Point` array type (`"System.Drawing.Point[]"`), because it later passes the array to a .NET method that requires this particular array type as a parameter. Otherwise, the procedure could continue to use the array directly as a `System.Array` object.

Note: The `Infragistics.Win.Misc.UltraButton` class is one of the OpenEdge Ultra Controls for .NET provided with OpenEdge, and `Progress.Windows.Form` is an OpenEdge extension of `System.Windows.Forms.Form` designed specifically for running in the ABL environment. For more information on these objects, see Creating and Using Forms and Controls on page 89.

Also note that if this procedure did not create an instance of the array element type (`System.Drawing.Point`) to set its array element values, it could obtain the array element type information needed to instantiate the array object using the static GetType( ) method of the Progress.Util.TypeHelper class. The code to support this method could look like the following code in bold:

```
...
DEFINE VARIABLE rType AS CLASS System.Type NO-UNDO.
...

/* Create .NET Point array */
ASSIGN
  rType       = Progress.Util.TypeHelper:GetType("System.Drawing.Point")
  rArray      = System.Array:CreateInstance(INPUT rType, INPUT 8)
  rPointArray = CAST(rArray, "System.Drawing.Point[]").
```

PointArray.p continues with the following code to initialize the user interface elements and graphical constants used to build and display octagons. To create a .NET form, the procedure uses the OpenEdge .NET class, Progress.Windows.Form, which, as previously noted, is derived from the System.Windows.Forms.Form class.

```
/* Prepare UI to draw octagons */
ASSIGN
  rDrawBtn              = NEW UltraButton( )
  rDrawBtn:AutoSize     = TRUE
  rDrawBtn:Text         = "Draw"
  rDrawBtn:DialogResult = DialogResult:None
  rDrawBtn:Top          = 4
  rDrawBtn:Left         = 4.

ASSIGN
  rForm        = NEW Progress.Windows.Form( )
  rForm:Height = 300
  rForm:Width  = 300
  rForm:Text   = "Octagons".

rForm:Controls:Add(rDrawBtn).

sqrt2 = SQRT(2.0).                /* "Constant" needed to draw octagons */
rDrawBtn:Click:Subscribe("Draw"). /* Event handler for drawing */

WAIT-FOR rForm:ShowDialog( ).
rForm:Dispose( ).
```

To actually draw the octagon, the example subscribes an internal procedure, Draw, as a handler for the Click event on the form button. Thus, when you click the button referenced by rDrawBtn, the procedure updates the octagon point coordinates and draws it in the form. For more information on handling .NET events, see Handling .NET events on page 57.

The Draw event handler procedure that follows responds to the button Click event by drawing the octagon from the point array returned by the AdjustOctagon user-defined function. It is the DrawPolygon( ) method that requires that the array object be passed as a "System.Drawing.Point[]" object rather than as the System.Array super class object initially created.

```
PROCEDURE Draw:
  /* Draw octagon from point array */
  DEFINE INPUT PARAMETER sender AS CLASS System.Object    NO-UNDO.
  DEFINE INPUT PARAMETER e      AS CLASS System.EventArgs NO-UNDO.

  DEFINE VARIABLE rGraphics AS CLASS System.Drawing.Graphics NO-UNDO.
  DEFINE VARIABLE rPen      AS CLASS System.Drawing.Pen      NO-UNDO.

  ASSIGN
    rGraphics  = rForm:CreateGraphics( )
    rPen       = NEW System.Drawing.Pen(System.Drawing.Color:Black)
    rPen:Width = 2.

  /* Pass adjusted point array to .NET method to draw octagon */
  rGraphics:DrawPolygon(rPen, AdjustOctagon( )).
END PROCEDURE.
```

Note: The PointArray.p procedure also demonstrates the technique for drawing graphics using the graphics object (rGraphics) that is associated with a .NET form. You can only draw form graphics in a form event handler, because the associated graphics object is available for drawing only during the handling of a form event.

As noted previously, the `AdjustOctagon` user-defined function determines all the points of the array required to draw an octagon within the current client area of the form. This, then, is where ABL uses the single `System.Drawing.Point` object (`rPoint`) to set each element of the array using a 0-based index (`idx`).

```
FUNCTION AdjustOctagon RETURNS "System.Drawing.Point[]":
  /* Adjust and return octagon point array according to form size */
  DEFINE VARIABLE iHorizA    AS INTEGER NO-UNDO.
  DEFINE VARIABLE iHorizSide AS INTEGER NO-UNDO.
  DEFINE VARIABLE iVertA     AS INTEGER NO-UNDO.
  DEFINE VARIABLE iVertSide  AS INTEGER NO-UNDO.
  DEFINE VARIABLE iHoffset   AS INTEGER NO-UNDO.
  DEFINE VARIABLE iVoffset   AS INTEGER NO-UNDO.

  /* Calculate octagon size and position based on form client area */
  RUN CalcOctagonSide(INPUT rForm:ClientSize:Width, OUTPUT iHorizA,
                      OUTPUT iHorizSide, OUTPUT iHoffset).
  RUN CalcOctagonSide(INPUT rForm:ClientSize:Height, OUTPUT iVertA,
                      OUTPUT iVertSide, OUTPUT iVoffset).

  /* Generate octagon points */
  DO idx = 0 TO 7:
    CASE idx:
      WHEN 0 THEN ASSIGN
        rPoint:X = iHoffset + iHorizA
        rPoint:Y = IVoffset.
      WHEN 1 THEN ASSIGN
        rPoint:X = iHoffset + iHorizA + iHorizSide
        rPoint:Y = iVoffset.
      WHEN 2 THEN ASSIGN
        rPoint:X = iHoffset + ( 2 * iHorizA ) + iHorizSide
        rPoint:Y = iVoffset + iVertA.
      WHEN 3 THEN ASSIGN
        rPoint:X = iHoffset + ( 2 * iHorizA ) + iHorizSide
        rPoint:Y = iVoffset + iVertA + iVertSide.
      WHEN 4 THEN ASSIGN
        rPoint:X = iHoffset + iHorizA + iHorizSide
        rPoint:Y = iVoffset + ( 2 * iVertA ) + iVertSide.
      WHEN 5 THEN ASSIGN
        rPoint:X = iHoffset + iHorizA
        rPoint:Y = iVoffset + ( 2 * iVertA ) + iVertSide.
      WHEN 6 THEN ASSIGN
        rPoint:X = iHoffset
        rPoint:Y = iVoffset + iVertA + iVertSide.
      WHEN 7 THEN ASSIGN
        rPoint:X = iHoffset
        rPoint:Y = iVoffset + iVertA.
    END CASE.
    /* Replace point in point array */
    rPointArray:SetValue(rPoint, idx).  END.
  RETURN rPointArray.
END FUNCTION.
```

ABL can use this single `System.Drawing.Point` object to set each array element, because, as a value type, .NET maintains its own copies of all the `Point` objects stored in the array. ABL can then reset the members of all the array element objects from a single modified object that ABL passes to the `SetValue( )` method. After all the `Point` objects have been set, the function returns the updated point array as a value for use in the `Draw` event handler (previously described).

The following `CalcOctagonSide` internal procedure uses a formula based on the square root of 2 to calculate the horizontal or vertical components used to derive coordinates for each point of the octagon.

```
PROCEDURE CalcOctagonSide:
  DEFINE INPUT  PARAMETER pDimension AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER pA         AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER pSide      AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER pOffset    AS INTEGER NO-UNDO.

  /* Calculate a side of a regular octagon with offsets */
  ASSIGN
    pA      = 1 / ((2 + sqrt2) / (pDimension * 0.8))
    pSide   = pA * sqrt2
    pOffset = pDimension * 0.1.
END PROCEDURE.
```

8

Notes on using the ProBindingSource

This appendix complements and expands upon the information in Binding ABL Data to .NET Controls on page 131.

For details, see the following topics:

- Determining what to bind to the ProBindingSource
- Repopulating data in an ABL data source bound to a ProBindingSource
- Using the ProBindingSource AddNew( ) and Remove( ) methods

Determining what to bind to the ProBindingSource

If the control(s) bound to a ProBindingSource will show only data from a single query (single table or single join query), the ProBindingSource should be bound to that query. If the control(s) bound to a ProBindingSource need to show hierarchical data (for example, customers and their orders), the ProBindingSource should be bound to a ProDataSet.

While it is possible to bind to a ProDataSet if you want to see only the data from the top-level table, doing so is inefficient. Internally, objects are created on the .NET side for each level in the ProDataSet's hierarchy. These .NET objects act as the conduit between the control and the underlying ABL data source— the temp-table. When the binding is initiated, each of these .NET objects has to communicate back to the AVM to acquire the schema associated with that level of the hierarchy. Clearly, if you are interested in only one level of data, the extra expenditure of time and resources related to acquiring the schema associated with all levels is unnecessary.

If you already have a ProDataSet, you do not need to create new queries to bind to if you are only interested in one level: You can bind to the queries already associated with the ProDataSet. For example, if you want to see the data from the top-level table, execute the following statement:

```
myBindingSource:Handle = myDataSet:TOP-NAV-QUERY.
```

If you have a control that shows data only from a child table, execute the following statement:

```
myBindingSource:Handle = myDataSet:GET-RELATION(n):QUERY.
```

In this case, *n* represents the level of data in which you are interested.

Repopulating data in an ABL data source bound to a ProBindingSource

If an application needs to repopulate the data in the underlying ABL data source that is bound to a ProBindingSource, you can accomplish this in several different ways. For example:

- You can execute new `QUERY-PREPARE( )` and `QUERY-OPEN( )` methods on the query that the ProBindingSource is bound to.

  Because the ProBindingSource is directly bound to the query that was just reopened, you do not have to do anything further as long as the ProBindingSource's `AutoSync` property is set to TRUE (which is the default). If the property is set to FALSE, simply call `myBindingSource:RefreshAll( )` to notify the control to refresh.

- You can empty and repopulate a temp-table by executing a new query on the database and then creating appropriate temp-table records.

  If you choose this option, you must reopen the query the ProBindingSource is bound to (that is, the query on the temp-table). This is important: Even though the query criteria has not changed (it is still for each temp-table record), the ProBindingSource must be working from the right query result set in order for the control to show the correct records. In particular, having the correct count of records in the result set is critical.

  Reopening the query takes care of this, and you must call `myBindingSource:RefreshAll( )` to notify the control to refresh.

To empty a control bound to a ProBindingSource, the application must modify the query, so that no records satisfy it, and then reopen it. (Neither closing the query nor setting the `Handle` property to the `Unknown value (?)` is sufficient.)

For example, here is a typical query that would empty the result set:

```
FOR EACH Customer WHERE ROWID (Customer) = ?:
```

Here is another way to repopulate the data; however, this approach is highly inefficient and not recommended:

1. Unbind the control from the ProBindingSource. For example:

```
myControl:DataSource = ?
```

2. Disconnect the ProBindingSource from its data source. For example:

```
myBindingSource:Handle = ?
```

3. Update the data in the ABL data source; for example, repopulate the temp-table or reopen a query.
4. Re-establish the bindings.

   While this series of steps does accomplish the task at hand, it is inefficient for the following reasons:

   - When you set `myBindingSource:Handle` to the `Unknown value (?)` as shown in step 1, all of the underlying .NET objects that make the ProBindingSource function are deleted. When the `Handle` property is reset as shown in step 2, these objects are recreated and again need to communicate to the AVM to get the schema of the underlying data source.
   - The .NET control itself is doing work to destroy and re-establish its bindings.

Using the ProBindingSource AddNew( ) and Remove( ) methods

The ProBindingSource inherits the `AddNew( )` method and the `Remove( )` method from its parent `System.Windows.Forms.BindingSource` class. These methods are meant to be called by the .NET control to effect a change in the underlying data source. For example, when a user opens up a new row in a grid, the control calls `AddNew( )`. If a user deletes a row, the control calls `Remove( )`.

There is no need to call these methods from the ABL. The goal is to create a new record in the temp-table. If you called `AddNew( )` from the ABL, you would have to subscribe to the `CreateRow( )` method that this would fire and create the record in the event handler. This approach is rather circuitous. Instead, simply execute the CREATE statement directly (or through a subroutine or method call). There is no benefit to involving the ProBindingSource prior to the creation of the record.

In order to have this record recognized by the control, you can either reopen the query that the ProBindingSource is bound to, or add this record to the result set of the query. You can do the latter by using the `CREATE-RESULT-LIST-ENTRY( )` method of the query.

If the ProBindingSource's `AutoSync` property is set to TRUE (which is the default), you are done. If the property is set to FALSE, simply call `myBindingSource:RefreshAll( )` to notify the control to refresh.