Progress®



OpenEdge® Getting Started:
GUI for .NET Primer

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....7
Purpose....7
Audience....8
Organization....8
Using ABL documentation....9
References to ABL compiler and run-time features....9
References to ABL data types....9
Examples of syntax descriptions....10
Long syntax descriptions split across lines....11
Complex syntax descriptions with both required and optional elements....12
OpenEdge messages....12
Obtaining more information about OpenEdge messages....13

Chapter 1: Object-oriented Programming in ABL....15
Classes....15
Data Members....16
Properties....17
Methods....17
Class events....18
Constructor....18
Destructor....18
Inheritance....18
Interfaces....19
Garbage collection....20
Essential ABL elements....20

Chapter 2: GUI for .NET Environment....23
.NET Framework....23
Tools....26

Chapter 3: Basic Types of Forms and Application Design....27
What is a form?....27
Main form....28
MDI and MDI child forms....28
Multiple, non-modal forms....29
Dialog forms....29
Creating and initializing forms....29

Using the WAIT-FOR statement....29
Waiting on a main or MDI form....30
Waiting on multiple, non-modal forms....30
Modal dialogs....30
Form inheritance....31
Form and window coexistence....31
Embedding an ABL window in a .NET form....31

Chapter 4: Controls in Forms....33
Control sources....33
Microsoft .NET Windows Forms UI controls....34
OpenEdge Controls....35
OpenEdge Ultra Controls for .NET....36
Third-party controls....36
ABL user controls....36
ABL-derived controls....36
Working with .NET forms and controls....37

Chapter 5: Events....39
.NET event loop....39
Event handlers....40

Chapter 6: Data Binding....41
Binding to a .NET control....41
Binding to a data source....42
Buffer....43
Query....43
ProDataSet....43
Defining ProBindingSource tables and fields....43

Chapter 7: Service Layers....45
Business Entities....45
Service Adapters and Service Interfaces....46

Preface

For details, see the following topics:

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

Purpose

OpenEdge® supports the ability to access Microsoft .NET objects from ABL much as if they were native ABL classes. ABL access to .NET objects is designed to support the OpenEdge GUI for .NET, which provides an alternative Windows desktop user interface for ABL applications, based on .NET forms and controls. ABL allows your application to use the GUI for .NET together with the OpenEdge GUI, and provides specific support for using .NET forms and ABL windows together in the same ABL session.

For today's modern applications, OpenEdge makes available an in-the-box .NET control set, OpenEdge Ultra Controls for .NET (Ultra Controls), that provide a feature-rich GUI. Ultra Controls includes a number of full-featured .NET controls from Infragistics NetAdvantage. GUI applications can be built using either the Progress Developer Studio for OpenEdge Visual Designer, the OpenEdge ABL text editor, or, at the minimum, any text editor.

ABL supports .NET objects and Ultra Controls entirely within the ABL context by incorporating the .NET class hierarchy as a subset of the ABL class hierarchy. ABL can interact with .NET objects using enhanced ABL features without, itself, becoming a native .NET language. This allows you to continue to use ABL exactly as you have in previous OpenEdge releases that do not provide access to .NET objects.

Much has already been written about working with .NET forms and objects in ABL from a bottom-up, how-to perspective. This manual, however, takes an alternate view, providing a top-down, conceptual look at why one builds a GUI for .NET application using one approach or another, and includes the pros and cons of each approach.

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 provides a guide to choosing among different options for creating a new GUI for .NET application, as well as transforming an existing OpenEdge application to use the GUI for .NET, you can find additional information on this feature in the following manuals:

- *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 Programming* — A guide that describes how to access .NET objects using ABL to implement the OpenEdge GUI for .NET in your OpenEdge applications.
- *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 is written for the ABL developer who is thoroughly familiar with programming in ABL. For more information about ABL, see *OpenEdge Getting Started: ABL Essentials* and *OpenEdge Development: ABL Reference.*

It is also best if the reader is familiar with the Microsoft .NET Framework, object-oriented programming concepts, and has an understanding of another object-oriented programming language, such as C#. For more information on working with .NET in ABL, see *OpenEdge Development: GUI for .NET Programming*. For more information about object-oriented programming in ABL, see *OpenEdge Development: Object-oriented Programming*.

Having a working knowledge of the Progress Developer Studio for OpenEdge Visual Designer tool greatly enhances your productivity in building OpenEdge GUI for .NET applications. For more how-to information about using the Visual Designer, see *OpenEdge Getting Started: Introducing the Progress Developer Studio for OpenEdge Visual Designer*.

Organization

Object-oriented Programming in ABL on page 15

Describes extensions to ABL that provide a cohesive and standard object-oriented programming model, while continuing to support the programming model available in previous releases of OpenEdge.

GUI for .NET Environment on page 23

Describes the ability to access Microsoft .NET objects from ABL much as if they were native ABL objects.

Basic Types of Forms and Application Design on page 27

Describes the types of .NET forms for building different types of ABL applications.

Controls in Forms on page 33

Describes the .NET Windows Forms UI controls and non-UI controls, the built-in ABL controls, and the optional OpenEdge Ultra Controls for .NET that can be added to your application.

Events on page 39

Describes the .NET event loop and the ABL event handlers that are used when your application uses GUI for .NET.

Data Binding on page 41

Describes data binding in the context of the connection between an ABL data source, which can be a ProDataSet, a query on a database table or temp-table, or a buffer, and a .NET UI component.

Service Layers on page 45

Describes the advantages of incorporating Business Entities, Service Layers, and Service Interfaces into your application.

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

Object-oriented Programming in ABL

Before getting into a discussion about OpenEdge GUI for .NET, it helps to understand the object-oriented (OO) foundation that supports it all. This section introduces key terms and concepts of object-oriented programming in ABL.

For details, see the following topics:

- Classes
- Interfaces
- Garbage collection
- Essential ABL elements

Classes

Beginning with OpenEdge® Release 10.1, ABL includes support for *classes*. Classes allow an ABL application to be built from built-in classes, as well as user-defined class-based objects that can be defined and organized for use by an application at compile time. Class-based objects are defined and managed using standard features of object-oriented programming available in ABL, and are similar to other programming languages such as Java. Procedure-based and class-based objects can coexist in a single application.

Object-oriented programming support in ABL extends the language to provide a cohesive and standard object-oriented programming model, while continuing to support the programming model available in previous releases of OpenEdge. That is, ABL provides support for programming with classes in addition to its support for programming with procedures with full interoperability between the two. This object-oriented language support is a natural basis for writing applications that conform to the OpenEdge Reference Architecture (OERA) introduced with OpenEdge Release 10.

You define a class using the CLASS statement and define its members within the class block terminated by the END CLASS statement. A new instance of a class is created using the NEW function. This is comparable to running a persistent procedure using the RUN statement, and returning a handle to the procedure with the SET option.

In OpenEdge releases prior to Release 10.1A, ABL persistent procedures allow you to create and manage objects in which most of the relationships between them are created and managed at run time. ABL classes, on the other hand, contain relationships that you create at compile time. Unlike a persistent procedure, a class defines a well-structured, strong type (data type) that ABL recognizes and verifies at compile time and that can be realized as an object at run time. A persistent procedure is weakly typed and therefore can only be verified at run time. The data type of a class is defined by its members. That is, as long as the interface defined by the members of a class does not change, the data type of the class remains the same no matter how much you change its implementation. This has significance for determining when you have to recompile a given class or procedure when a class upon which it depends changes.

Class members are themselves strongly typed elements that provide data and behavior for the class, similar to the variables, handle-based objects, internal procedures, and user-defined functions of a persistent procedure. Again, as strongly typed elements, the members of a class are verified at compile time, while the similar elements of a procedure can only be verified at run time. In addition, classes are built in strict hierarchies, where any given user-defined class inherits strongly-typed members from another class as well as defining members of its own, all of which can be inherited by yet another user-defined class. This strong typing of classes and their inheritance relationships allows for robust error checking at compile time, when an application is in the development cycle, and long before it is available to end users.

In terms of the static DEFINE versus dynamic CREATE ABL perspective, all OO programming in ABL is dynamic. You instantiate objects explicitly at run time and you set their properties much as you do with the ABL CREATE statement.

ABL and .NET organize their class-based type hierarchies using different mechanisms. ABL organizes user-defined types into packages, which are logical pathnames that correspond to physical directory paths relative to PROPATH. It then stores the types in class definition (.cls) files that reside in the corresponding directories.

On the other hand, .NET organizes its object types into namespaces, which are logical pathnames that are defined along with the object types associated with them. .NET namespaces are completely logically defined and have no correspondence to physical storage of any kind. Instead, .NET stores object type definitions, along with their defined namespaces, in assembly files. An assembly can be a Windows dynamic link library (.dll) or executable (.exe) file that is formatted to store one or more types associated with their individual namespaces. These assemblies are then stored in designated locations, which can include locally configureable directories or a standard location defined by .NET known as the Global Assembly Cache (GAC). OpenEdge supports options for specifying the location of the assemblies you access as part of the GUI for .NET.

Thus, 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 logical construct that is associated with each type that is stored in an assembly.

Both .NET namespaces and ABL packages are hierarchical. The representation of each type of hierarchy in ABL uses the same dot-separated notation, whether it is logical for namespaces or physical directory-based for ABL classes.

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

Data Members

Variables defined within the main block of a class are known as *data members*. Like variables defined in the main block of a procedure, in an internal procedure, or in a user-defined function, data members can be defined as a scalar or an array type, such as CHARACTER, INTEGER, DATE, DECIMAL, LOGICAL, and so on.

In keeping with strong typing, the compiler checks assignments that involve data members at compile time. Assignments involving data members that are considered invalid, for example, performing an integer operation on a character string, are not allowed and raise ERROR. Strong typing allows for catching many potential type errors at compile time. Data members are PRIVATE by default.

Like a variable defined within the main block of a procedure, a data member of a class is scoped to the entire class and can be used freely in methods of the class. Unlike a variable defined within the main block of a procedure, a data member of a class can be defined with an access mode that determines where it can be read or updated. For example:

- **PUBLIC** — A public data member can be read or updated by the defining class, any inheriting class, or any class that instantiates the class.
- **PROTECTED** — A protected data member can be read or updated by the defining class and any inheriting class. An instance can access the protected member of a second instance that is at the same level or in a superclass in the class hierarchy.
- **PRIVATE** — A private data member can be read or updated by the defining class, which is the default access mode. An instance can access the private data member of another instance if both instances are from the same class.

Properties

Similar to ABL variables, *properties* in class files maintain data or state. Like a data member in a class, a property can be defined with an access mode (PUBLIC, PROTECTED, PRIVATE) that determines where it can be read or updated. Properties are PUBLIC by default.

But unlike ABL variables or class data members, properties can be defined with a GET accessor and/or a SET accessor for reading and writing a property's value, respectively. These accessors, which are blocks of ABL statements, can contain code which is executed when the property is read (GET) or assigned (SET). If either the GET accessor or the SET accessor is omitted, the property is defined as write-only or read-only, respectively.

While there is no direct equivalent to properties in traditional ABL, properties can be simulated in persistent procedures, for example in the ADM2, by defining user-defined functions that control the reading and writing of an ABL variable.

Methods

Similar to internal procedures and user-defined functions in procedure files, *methods* provide behavior in class files. Methods can have parameters and can define a return type. A method can also have a return type of VOID, which means that it does not return a value, and is therefore comparable to an internal procedure, as opposed to a function.

Like data members and properties, methods can be defined with an access mode, either PUBLIC, PROTECTED, or PRIVATE, that determines where and how the method is called. Methods are PUBLIC by default.

As with internal procedures and user-defined functions, you can raise ERROR within a method or property GET accessor or SET accessor using RETURN ERROR or UNDO, THROW. This statement allows you to raise (THROW) an error as an object. An error object, part of the structured error handling feature introduced in OpenEdge Release 10.1C, is a class-based object that can be trapped with custom error handlers.

Class events

Similar to named events in procedure files, *class events* provide a way to subscribe a method or procedure that executes in response to application conditions. Unlike a named event, which is defined when you *publish* the event to execute its event handlers, you define class events as members of a class independent of any event handlers that might execute for them. A class event is thus defined with a name, and can have parameters and a return type (similar to a method), which define the signature that must match the signature of any handler for this event. The return type for a class event and its handler is always `VOID`. Finally, unlike named events that can have only internal procedures as handlers, class events can have both internal procedures and methods as handlers, as long as the handler signatures match the signatures of their respective events.

Like data members, properties, and methods, events can be defined with an access mode, either `PUBLIC`, `PROTECTED`, or `PRIVATE`, that determines where and how the event can have a handler *subscribed* for it. Events are `PUBLIC` by default. However, note that events can only be published from within the class where they are defined, regardless of their access mode.

A class event can have multiple handlers subscribed for it in different classes and procedures of an application, and these handlers can also be unsubscribed. `ERROR` can also be raised for a class event on the statement that publishes the event when any one of its subscribed handlers raises `ERROR`.

Constructor

Similar to a Main Block in an ABL persistent procedure, a *constructor* is a block of code that is executed when a class-based object is instantiated. Constructors are special methods that can have parameters, but do not have a return type. In a hierarchy, a constructor in each class of the hierarchy is executed.

Destructor

A *destructor* is a special type of method that is executed when a class-based object is deleted, either as a result of automatic garbage collection, or programmatically using the `DELETE OBJECT` statement. Destructors do not have parameters and do not have a return type. In a hierarchy, all destructors of the classes in the hierarchy are executed.

This is a good place to put cleanup code that you want to ensure is run before the class-based object is deleted. That is, unless the entire session ends abnormally, in which case the destructor might not run.

Inheritance

Inheritance of common data and behavior is at the core of object-oriented programming. A class inherits the non-private data members, properties, methods, and events of its super class (also referred to as the base or parent class). The inheriting class can also extend the interface of its super class by defining its own members, making all of its inherited and defined non-private members available for inheritance by another class (a subclass). In this way, each class in a class hierarchy derives its interface based on the super classes above it.

One of the key benefits of inheritance is code reuse, coupled with defined relationships between classes. The compiler cross-checks the contents of super classes and subclasses in a class hierarchy at compile time, which is not done for procedures. In an inheritance hierarchy of ABL classes, a method in a subclass can also override and redefine a method of the same name and signature in a super class. In this case the most-derived (inherited) version of the overridden method is invoked at run time.

Compared to internal procedures and user-defined functions which have no inherent relationships among them, object-oriented inheritance provides for organizing classes and factoring out common elements of related classes. You can also define a form of inheritance with procedure objects using super procedures. A set of related super procedures can act in much the same way as a class hierarchy, but their relationships are validated at run time.

These are some notable ABL elements for working with inherited classes:

- `INHERITS` option, `CLASS` statement — The `INHERITS` option specifies the super class from which your derived class inherits. If you do not specify a super class, all ABL user-defined classes derive from the top-level super class (*root class*), `Progress.Lang.Object`, by default. Thus, the interface defined by every ABL class includes members inherited from this ABL root class.
- **`METHOD, DEFINE PROPERTY, DEFINE EVENT, DEFINE VARIABLE`**, and various other **`DEFINE`** statements — These statements allow you to extend the inherited interface with additional methods, properties, events, and data members of various types.
- **`OVERRIDE`** option, **`METHOD`** statement — When an ABL class inherits a class, it can modify behavior of the base or parent class by overriding public and protected instance methods in the inherited class hierarchy with its own ABL `OVERRIDE` methods.
- **`SUPER`** statement — The `SUPER` statement invokes a constructor for the immediate super class as the first statement in a constructor of the defining class.
- **`THIS-OBJECT`** statement — The `THIS-OBJECT` statement invokes a constructor defined in the current class definition.
- **`SUPER`** system reference — Using the `SUPER` system reference, a subclass can call the `PUBLIC` and `PROTECTED` instance methods of its super class in the inherited class hierarchy. If the specified method definition is not found in the subclass's immediate super class, the ABL compiler repeatedly looks to the next super class in the inherited class hierarchy until it finds the definition.

Interfaces

Interfaces, which have long existed in OO languages such as C#, are also supported in ABL. An interface is an object type, like a class, that consists of one or more method, property, or event prototypes that a class must implement. That is, the methods, properties, and events defined in an interface are empty and do not contain any executable code. Any class that includes an interface in its definition must implement all the methods, properties, and events of that interface. Note that an ABL interface never defines prototypes for data members that a class must implement. However, it can include temp-table and ProDataSet definitions to specify parameters for method or event prototypes that define temp-tables and ProDataSets as parameters.

You define an ABL interface using the `INTERFACE` statement and define its members within the interface block terminated by the `END INTERFACE` statement. Like a class type, you can define an interface type in a package. Similarly, a .NET interface, like .NET class, can be defined in a namespace.

To implement one or more interfaces in a user-defined class, you specify them with the `IMPLEMENTS` option of the `CLASS` statement. By implementing an interface, you assure any other code (class or procedure) that uses your class that the methods, properties, and events defined by the interface have been implemented. You can use a class object reference defined as an interface to access the interface methods, properties, and events, even for object references of different class types, as long as the given class implements the specified interface.

Defining and implementing interfaces is useful when you want to build classes with a standard set of methods, properties, and events. The implementation of interfaces is validated at compile time.

Garbage collection

ABL developers are familiar with the use of the `DELETE OBJECT` statement for destroying handle-based objects, such as a widget, a procedure, a server, or a socket. Explicitly deleting class-based objects is not necessary because of garbage collection. Class-based objects are deleted during garbage collection when all references to that object have been removed, that is, when its reference count goes to zero.

As long as you need an object, you must ensure that you maintain an object reference to that object. You can always assign the object reference to another variable before it goes out of scope, or you can pass it to another procedure, where you can continue to manage the object until either you delete it or it is deleted by garbage collection.

Garbage collection does not delete a class-based object with a circular reference. For example, Object A has a reference to Object B and Object B has a reference to Object A. To delete an object with a circular reference you must use the `DELETE OBJECT` statement.

Essential ABL elements

The OpenEdge GUI client provides access to .NET forms (analogous to ABL `WINDOW` widgets) and controls using object-oriented extensions to ABL. Using ABL, you create forms, add controls, access data, subscribe to events and write event handlers for .NET controls. Accessing and using .NET objects is very similar to working with ABL objects using classes, however, there are some exceptions, due to the differences between supported constructs in .NET and ABL.

The following are examples of ABL elements that are used when working with .NET forms and controls:

- **`USING`** — The `USING` statement makes code easier to read and less verbose by allowing you to abbreviate the full class or object type name in code. For example:

```
USING Progress.Windows.Form.
USING System.Windows.Forms.*.
USING Infragistics.Win.UltraWinListView.*.
```

- **`CLASS`** — The `CLASS` statement defines a user-defined class, representing a user-defined data type. Its characteristics are defined by a set of data members, including properties, that define class data and methods that define class behavior. For example:

```
CLASS MyUserControl:
  ...
END CLASS.
```

- **`INHERITS`** — When defining a class, the `INHERITS` option specifies the super class whose state and behavior this class inherits. For example:

```
CLASS MyUserControl INHERITS Progress.Windows.UserControl:
  ...
END CLASS.
```

- **IMPLEMENTS** — Classes can implement one or more interfaces, each of which are defined as a separate `.cls` file starting with an `INTERFACE` statement. An interface declares a common public mechanism to access behavior that one or more classes can define, even though these classes do not inherit from another common class. Interfaces allow you to define and manage common behavior that might be implemented differently in different classes and for different purposes. For example:

```
CLASS MyUserControl IMPLEMENTS IUpdatable:
  ...
END CLASS.
```

- **PUBLIC** — The `PUBLIC` access mode specifies that the object, code block, or data member can be accessed by the defining class, any of its inheriting classes, and any class or procedure that instantiates that class (through an object reference), or has access to the object reference. In this example, `VOID` indicates that the method does not return a value:

```
METHOD PUBLIC VOID addRecord( ):
  ...
END METHOD.
```

- **PROTECTED** — The `PROTECTED` access mode specifies that the object, code block, or data member can be accessed by the defining class and any of its inheriting classes. An instance can access a protected member of a second instance that is at the same level or in a super class in the class hierarchy. For example:

```
DEFINE PROTECTED VARIABLE cName AS CHARACTER NO-UNDO.
```

- **PRIVATE** — The `PRIVATE` access mode specifies that the object, code block, or data member can be accessed by the defining class. An instance can access the private member of another instance if both instances are from the same class. For example:

```
DEFINE PRIVATE VARIABLE bindSrc1 AS Progress.Data.BindingSource NO-UNDO.
```

- **CONSTRUCTOR** — Similar to a Main Block in an ABL structured procedure (`.p` or `.w`), a `CONSTRUCTOR` is a block that is executed when a class-based object is instantiated. A constructor of a class is a special method that is called when a class is instantiated, for example, using the `NEW` function. A constructor is named the same as the class. For example:

```
CONSTRUCTOR PUBLIC PurchaseOrderForm( ):
  ...
END CONSTRUCTOR.
```

- **DESTRUCTOR** — The `DESTRUCTOR` is a code block that is executed when a class-based object is deleted, either as a result of garbage collection or by using the `DELETE OBJECT` statement. This block is analogous

to an `ON CLOSE OF THIS-PROCEDURE DO:` block in a persistent procedure. A destructor is named the same as the class. For example:

```
DESTRUCTOR PUBLIC PurchaseOrderForm( ):
  ...
END DESTRUCTOR.
```

- **`METHOD`** — The `METHOD` is a code block that provides class behavior. Methods are similar to internal procedures and user-defined functions, in that they encapsulate object behavior. For example:

```
METHOD PRIVATE VOID InitializeComponent( ):
  ...
END METHOD.
```

- **`NEW`** — The `NEW` function creates an instance of a class and returns an object reference to that instance. The new or derived class is a subclass of its super class parent, inheriting its `PUBLIC` and `PROTECTED` data members, methods, and properties. All classes in ABL (including .NET classes) have the root super class `Progress.Lang.Object`. For example:

```
DEFINE VARIABLE bindSrc1 AS Progress.Data.BindingSource NO-UNDO.
bindSrc1 = NEW Progress.Data.BindingSource( ).
```

- **`THIS-OBJECT`** — An object reference to the class instance currently running. `THIS-OBJECT` is useful as a qualifier when the class member reference name is a reserved ABL keyword. The primary use of `THIS-OBJECT` is to pass a reference of the current object to another routine and to improve readability. For example:

```
THIS-OBJECT:Display( ).
```

- **`VALID-OBJECT`** — The `VALID-OBJECT` function returns `TRUE` if the object reference points to a valid ABL or .NET object instance. For example:

```
VALID-OBJECT(MyObjectRef).
```

- **`CAST`** — The `CAST` function returns a new object reference to the same class instance as an existing object reference, but with a different data type. The underlying object does not change, but the reference to it has changed. For example:

```
CAST(MyObjectRef, MySubClass).
```

2

GUI for .NET Environment

The GUI for .NET environment is comprised of extensions to the OpenEdge GUI client, ABL elements, and tools that support the ability to access Microsoft .NET objects from ABL much as if they were native ABL objects.

For details, see the following topics:

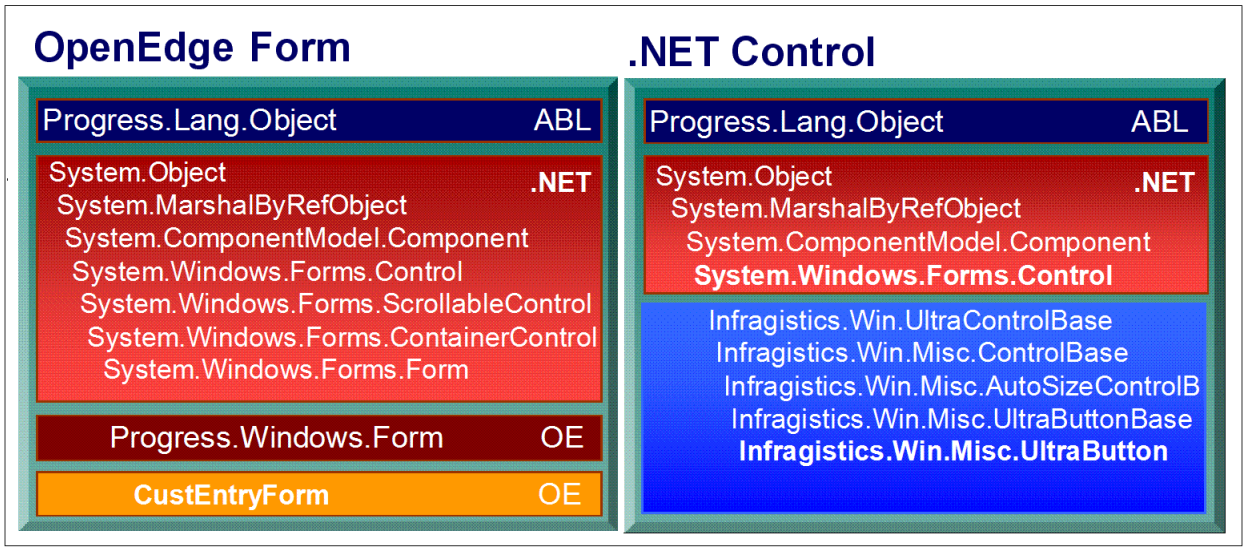
- .NET Framework
- Tools

.NET Framework

The OpenEdge GUI client supports the ability to access Microsoft .NET objects from ABL much as if they were native ABL objects. ABL access to .NET objects is designed to support the OpenEdge GUI for .NET, which provides an alternative Windows desktop user interface for ABL applications based on .NET Windows Forms forms and controls. ABL enables your application to use the GUI for .NET together with the traditional OpenEdge GUI, and provides particular support for using .NET forms and ABL windows together in the same ABL session.

ABL provides access to objects in a .NET class library. The following figure shows that all class-based objects in ABL are derived from the `Progress.Lang.Object` root class.

Figure 1: .NET Class Inheritance Stack



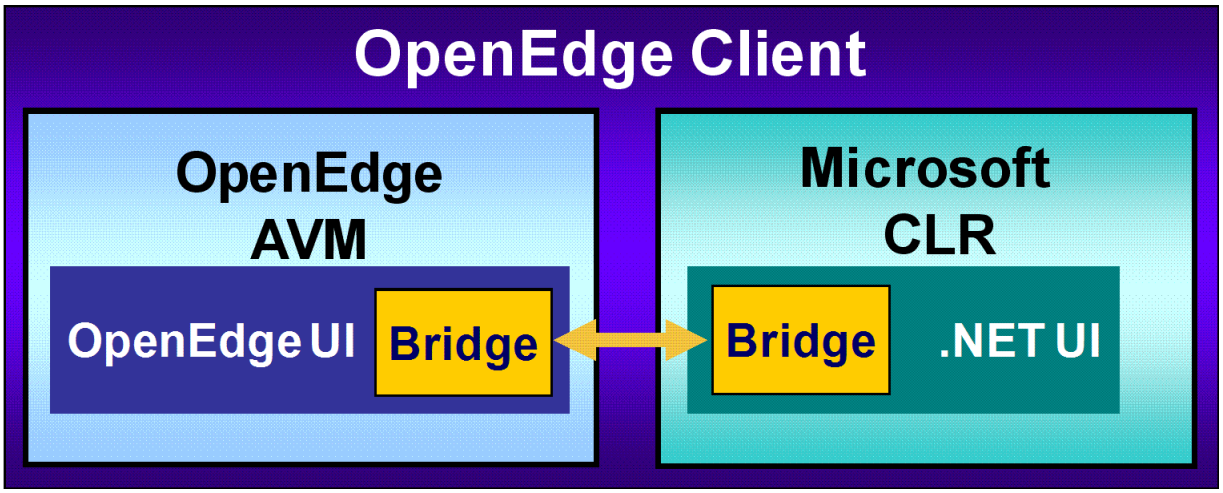
ABL supports .NET controls within ABL. ABL can interact with .NET objects using enhanced ABL features without, itself, becoming a native .NET language. This allows you to continue to use ABL as you have in previous OpenEdge releases. References to .NET controls in ABL are made using object-oriented (OO) syntax. You can make those references from procedural or class-based code or any combination.

The following figure represents the OpenEdge GUI for .NET architecture at a high level. In support of .NET objects and events, ABL incorporates the ABL Virtual Machine (AVM) and the .NET Common Language Runtime (CLR) within a single OpenEdge client process. The CLR is the core run-time engine for executing applications in the Microsoft .NET Framework.

What makes communication between the ABL and .NET possible is the bridge that provides the transformation by allowing information to go back and forth between the AVM and the CLR. When the CLR gets a request, such as to run a .NET method, the bridge converts the data between the CLR and the AVM.

On the AVM side you define your components, do your data access, do I/O blocking (a version of the `WAIT-FOR` statement), and write event handlers in ABL. On the .NET side, the CLR handles the visualization of the .NET UI. The user interacts with the CLR, because that is where the user interface is running.

Figure 2: OpenEdge GUI for .NET Architecture



The figure above depicts a simplified sample flow of control between the AVM and the CLR:

1. In ABL, a new .NET form is created. The form subscribes an ABL event handler method to its FormClosing event, and the form is run.
2. The request is passed over the bridge to the CLR to create the .NET form.
3. The form is displayed using a .NET method specified by the `WAIT-FOR` statement.
4. The user does some work and closes the form.
5. The FormClosing event message is sent to all subscribers from .NET.
6. The bridge sends the message to the AVM and causes the event handler method to run.

There is no one application architecture supported by GUI for .NET. You can either keep your existing architecture or follow a modularized design such as the OERA. There is no requirement to change your application architecture, though some applications might require more modifications than others to provide a GUI for .NET interface that communicates with existing data management and business logic. There might be compelling reasons to update parts of your application architecture to take advantage of advanced features that GUI for .NET provides. Many architectural topologies are supported, including fat client, client/server, and n-tier. GUI for .NET does not require you to change from one topology to another.

Just as there is no one UI technology that is applicable for all of today's applications, an application's primary use often dictates what UI technology works best. To put GUI for .NET development in perspective, it helps to compare this technology with other OpenEdge UI technologies.

The following table lists the various OpenEdge UI technologies, their primary use, and what development tools are best suited for building these types of business applications. These are suggestions; the table is not meant to be all-inclusive and there is overlap between technologies, their primary use, and tools.

Table 1: OpenEdge UI technologies

UI Technology	Primary Use	Tools to Use
GUI for .NET	Build modern, state-of-the-art GUIs for Microsoft Windows in ABL	Progress Developer Studio for OpenEdge Visual Designer
OpenEdge GUI	Build Windows 95 user interface look and feel	OpenEdge StudioProgress Developer Studio for OpenEdge AppBuilder, Editor
Open Client	Build alternate UIs (.NET & Java™) to OpenEdge applications, requires AppServer	Open Client Toolkit
WebSpeed®	Build web-browser based applications with zero client footprint	WebSpeed Workshop
Character	Build traditional green-screen user interfaces	OpenEdge StudioProgress Developer Studio for OpenEdge Editor4GL Development System

WebClient can be used as a deployment platform for OpenEdge GUI, GUI for .NET, and Character UIs.

Tools

The primary tool to use for GUI for .NET application development is Progress Developer Studio for OpenEdge, an Eclipse-based Integrated Development Environment (IDE) that includes a number of plug-ins, for example the OpenEdge Visual Designer, the OpenEdge Editor, and the OpenEdge Class Browser:

- **OpenEdge Visual Designer** — The Visual Designer is a WYSIWYG tool that supports visual building of complex GUIs in ABL based on .NET objects. You create user interfaces with Visual Designer by dragging and dropping .NET controls onto a Design Canvas. The Visual Designer generates ABL code written as classes that encapsulates the creation of the WYSIWYG design code. You can get some hands-on experience working with the Visual Designer in *OpenEdge Getting Started: Introducing the Progress Developer Studio for OpenEdge Visual Designer*.
- **OpenEdge Editor** — The Editor is a full-featured, color-coded editor for working with UI class files. For example, you run the Editor when you want to add methods that implement event handlers. Complementary to the OpenEdge Visual Designer.
- **OpenEdge Class Browser** — The Class Browser is an Eclipse view that lists the members of either an ABL or a .NET class, showing the signature for each member in ABL syntax.

3

Basic Types of Forms and Application Design

ABL supports several types of .NET forms for building several types of applications. A built-in form class can be used to create standard, borderless, and floating windows. You can also use this class to create modal windows such as a dialog. A special form, the Multiple Document Interface (MDI) form, can contain other forms as child forms, including ABL windows that you created in OpenEdge version 11 or earlier versions.

For details, see the following topics:

- What is a form?
- Using the WAIT-FOR statement
- Form inheritance
- Form and window coexistence
- Embedding an ABL window in a .NET form

What is a form?

Your GUI for .NET application will require the use of one or more types of forms. A form is a representation of any window displayed in your application. ABL supports several types of forms for building different types of applications. The OpenEdge .NET `Progress.Windows.Form` class can be used to create standard, borderless, and floating windows. You can also use this class to create modal windows, such as a dialog. A special kind of form, the Multiple Document Interface (MDI) form, can contain other forms as child forms, including ABL windows that you might have created in OpenEdge version 11 or earlier.

The form types you choose to meet your application requirements determine the objects, controls, methods, and event handlers necessary to tie it all together.

In ABL a WINDOW widget is created using the CREATE WINDOW statement. In this example the variable custWin contains a handle to the newly created window:

```
DEFINE VARIABLE custWin AS HANDLE NO-UNDO.
CREATE WINDOW custWin ASSIGN ...
```

A .NET form is similar to an ABL WINDOW widget. In ABL a .NET form is typically an instance of the Progress.Windows.Form class and is created using the NEW function. In this example, FormObjectRef contains an object reference to the new .NET form:

```
DEFINE VARIABLE FormObjectRef AS CLASS Progress.Windows.Form NO-UNDO.
FormObjectRef = NEW Progress.Windows.Form( ).
```

Main form

In this architecture a main, controlling form acts as a container for the objects, controls, event handlers, and other components that make up the application. A main form is analogous to an ABL WINDOW with an associated FRAME widget, which is either defaulted or explicitly defined.

The main form type is useful for dashboard applications. All other application forms are children of the main form. To construct your own application-specific forms use the Visual Designer.

MDI and MDI child forms

A Multiple Document Interface (MDI) form is a window that acts as the background of an application and is the container for MDI child forms. An application can have only one MDI form object but many MDI child forms. You can construct your own application-specific MDI form and child forms using the Visual Designer. If an MDI child form has menus, the child form's menu bar is merged with the MDI form object's menu bar when the MDI child form is active at run time. Your application determines how the menu merge will be performed.

The closest thing to MDI using ABL widgets is an ABL WINDOW containing multiple FRAME widgets. Even though each FRAME is parented to the same WINDOW, the similarity to true MDI ends there. While a WINDOW can contain a MENU-BAR widget, a FRAME cannot, so the idea of merging a child FRAME's menu with its parent WINDOW menu does not apply.

A minimized MDI child form is displayed as an icon within the MDI form. MDI child forms are designed independent of the MDI form, but are always contained within the MDI form at run time.

In ABL, you typically build a .NET MDI using different instances of Progress.Windows.Form. The difference between the MDI main form and the MDI child forms are in the property settings of each form instance.

You can also embed an existing ABL window in a special OpenEdge .NET MDI child form (Progress.Windows.MDIChildForm). When embedded in this way, only the client area of the original ABL window is displayed. The embedded window's title, menu, status area, message area, and border are not displayed. Embedding an ABL window is discussed in more detail in Embedding an ABL window in a .NET form.

Multiple, non-modal forms

An application with multiple, non-modal forms consists of several equal forms which are displayed at the same time or separately. This form type is useful when a set of windows is needed to perform a task and when the forms to display are determined at run time depending on the user and the task. To construct your own application-specific forms use the OpenEdge Visual Designer.

Dialog forms

Dialogs are modal forms that inherit from `Progress.Windows.Form` and that block for user input. Dialog forms are used to interact with the user and retrieve information, in a similar way to an ABL FRAME defined `WITH VIEW-AS DIALOG-BOX`.

You can construct your own application-specific dialog in the OpenEdge Visual Designer and add controls such as `Label`, `Textbox`, and `Button` to customize the dialog form to your specific needs. The .NET Framework also includes predefined dialog forms, such as File Open and message boxes, which you can use programmatically in your own applications.

You can use ABL dialog-boxes and .NET dialog forms in the same application. The ABL `MESSAGE` statement is similar to the `System.Windows.Forms.Application:MessageBox( )` method, both of which display similar types of alert boxes which represent a different type of dialog using a dedicated event loop. The ABL `SYSTEM-DIALOG COLOR` statement is comparable to the .NET `System.Windows.Forms.ColorDialog` or `System.Windows.Forms.FileDialog`.

Creating and initializing forms

The OpenEdge Visual Designer, available as an Progress Developer Studio for OpenEdge plug-in, is the WYSIWYG designer tool for building GUI for .NET applications. While visually designing a form, an ABL class file is generated for the form. The Visual Designer keeps track of all .NET assemblies and resources that you use in a project.

Within a GUI for .NET application, you can plan a gradual migration by converting some of your main windows to .NET forms and leaving the rest of the UI as ABL windows. This is possible because you can display ABL windows alongside .NET forms. OpenEdge allows parenting of forms to windows and windows to forms using the OpenEdge `PARENT` attribute on the window handle.

Using the WAIT-FOR statement

For an application that uses non-modal .NET forms, whether or not it also uses ABL windows, you must use a .NET event loop controlled by a single WAIT-FOR statement.

ABL supports a version of the `WAIT-FOR` statement that instructs the AVM to stop executing the current block and remain in a wait state until a .NET method that it calls returns. The AVM continues to respond to all incoming ABL events as well as .NET events, and it executes any associated triggers, event procedures, or .NET event handlers while in this wait state. To accommodate the different .NET form types, the `WAIT-FOR` statement supports multiple syntax versions. If your application uses non-modal .NET forms, including MDI forms, this .NET-oriented `WAIT-FOR` statement must be the first, non-modal `WAIT-FOR` of any kind in your application.

Waiting on a main or MDI form

Many GUI applications have a single, main form for the application. When the application starts, the form is displayed and when the form is closed, the application terminates. When the application waits on a main or MDI form, the ABL `WAIT-FOR` statement must specify `Application:Run( )` with the main form passed in as a parameter. This starts a .NET message loop which is used to capture events and run ABL event handlers.

This is an example of the `WAIT-FOR` statement for an application that has a single, main form:

```
WAIT-FOR Application:Run( FormObjectRef ).
```

When the main form `FormObjectRef` is specified in the `WAIT-FOR` statement, the form is displayed when the message loop is started.

Closing the main form terminates the `WAIT-FOR`. This continues execution in the current block in your program immediately after the `WAIT-FOR`. The application can close the main form by calling the `Close( )` method on `FormObjectRef`.

Waiting on multiple, non-modal forms

Waiting on a GUI application with multiple, independent, non-modal forms is set up differently than waiting on a main form. In this case, no form is specified on the `WAIT-FOR` statement. None of the forms are automatically displayed, so the `Show( )` method must be called explicitly to display a form. To display forms when the application is started the `Show( )` method is called on each form prior to the `WAIT-FOR` statement. The actual display of any forms is delayed until the `WAIT-FOR` statement executes. For example:

```
Form1ObjectRef:Show( ).
Form2ObjectRef:Show( ).
WAIT-FOR Application:Run( ).
```

Calling the `Close( )` method on all the forms does not end the `WAIT-FOR`. The application must end the `WAIT-FOR` by calling the `Application:Exit( )` method.

Modal dialogs

The `ShowDialog( )` method on the form is used in the `WAIT-FOR` statement to display and wait for a modal .NET dialog box. If you display the modal dialog box within an existing .NET event loop, the nested `WAIT-FOR` must be satisfied before returning to the outer `WAIT-FOR`. The application ends the `WAIT-FOR` by calling the `Close( )` method on the modal form object.

In this example, the `rDateForm` form is created and displayed as a modal dialog using the `ShowDialog( )` method:

```
rDateForm = NEW Progress.Examples.MyDialog.
...
WAIT-FOR rDateForm:ShowDialog( ).
```

This is an example of the WAIT-FOR statement for a modal dialog in which the parent form is specified:

```
WAIT-FOR FormObjectRef:ShowDialog( ParentFormObjectRef ).
```

ParentFormObjectRef is an object reference to the dialog's parent form. This version of the ShowDialog( ) method allows the dialog to display centered over the parent form.

Form inheritance

GUI for .NET supports inheritance from .NET forms. All of your application-specific forms typically inherit from the Progress.Windows.Form class. You might want to create your own ABL form class and then derive forms from that. This technique is a good idea for enforcing a company standard across all forms in your application. One advantage of putting common behavior in a super class is that feature changes or bug fixes to that class are propagated to all derived classes.

However, putting too much functionality in a super form class might result in having to write workaround code (overrides) in derived forms to get the particular instance behavior you want. So, try to put in a super class only functionality that is required by all derived classes.

OpenEdge applications can support both ABL windows and .NET forms. .NET forms can be mixed with ABL windows in the same application. For Windows desktop applications that already use the OpenEdge GUI, you can add new GUI for .NET forms without changing the ABL windows of the existing user interface. The windows and forms can run side-by-side and one can parent the other. Each window and form is an independent entity in the application.

Form and window coexistence

To allow forms and windows to coexist in an application, several ABL enhancements work to tie them together. A FormProxy object is a wrapper that is created for every ABL window. Similar to a form, the FormProxy allows the window to reside on the form chain, which is a list of all running forms.

Except for the default window, a FormProxy is created for windows that are explicitly created in the session, not for windows created internally by the AVM. The AVM maintains form proxies, automatically creating and destroying them. Even though you do not work with FormProxy objects directly, it is helpful to know about them, since they exist on the form chain.

A shadow window widget, which is created for every form, provides a means of parenting a window to a form. The shadow window has no visualization.

Embedding an ABL window in a .NET form

For Windows desktop applications that already use the OpenEdge GUI, you can add new GUI for .NET forms without changing the ABL windows of the existing user interface. Another way to integrate ABL windows with GUI for .NET forms is to embed the client area of ABL windows into a .NET form.

There are several options for embedding OpenEdge windows in a .NET form. Existing ABL windows can be shown embedded in:

- A bounded area in a form, inherited from `Progress.Windows.WindowContainer`. The developer determines the size and location of the container.
- An MDI child form, inherited from `Progress.Windows.MDIChildForm`. The size and location of the embedded window are determined by the size of the MDI child form client area.

The embedded window continues to exist on the ABL session window chain.

4

Controls in Forms

ABL forms are containers for .NET Windows Forms UI controls and non-UI controls, built-in ABL controls, OpenEdge Ultra Controls for .NET, and other third-party controls.

For details, see the following topics:

- Control sources
- Working with .NET forms and controls

Control sources

ABL-derived forms can be containers for .NET Windows Forms UI and non-UI controls. .NET contains a variety of controls that you can add to forms—controls that display text boxes, buttons, drop-down boxes, radio buttons, and even Web pages. From a coding perspective a .NET form is made up of a set of controls.

GUI for .NET controls can also come from several different areas, the difference being in how you obtain the control:

- Microsoft .NET UI Controls. This is the OpenEdge name for the .NET Framework controls that are installed as visual design components.
- OpenEdge Ultra Controls for .NET (Ultra Controls), based on Infragistics NetAdvantage for Windows Forms.
- OpenEdge .NET controls / components:
  - Window Container control (`Progress.Windows.WindowContainer`).
  - User control (`Progress.Windows.UserControl`).

    - ProBindingSource control (`Progress.Data.ProBindingSource`) for binding data to a control.

- Other 3rd party .NET Windows Forms controls.
- ABL-derived .NET custom controls: user controls and inherited controls.

Control vendors normally provide detailed documentation and help on using their controls. While this documentation is often written assuming the use of a .NET language, the description of a control and its properties and events are all relevant for its use in ABL. You can access this documentation in the following ways:

- Visit the vendor's website. For example, to obtain more information on Microsoft .NET UI Controls and all .NET object types available with the .NET Framework version 4.0, see the Class Library reference documentation at the following location: http://msdn.microsoft.com/en-us/library/gg145045.aspx
- Open the control in the OpenEdge Class Browser to see control definitions using a modified ABL syntax.
- Press **F1** after you place the control on the form within Progress Developer Studio for OpenEdge Visual Designer. The online help opens on the respective vendor site.

Microsoft .NET Windows Forms UI controls

Microsoft Windows Forms supports a variety of controls that you can add to forms creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system. Windows Forms controls support a variety of features, from the most basic to features that emulate the functionality in high-end applications like Microsoft Office.

Menus and toolbars

Windows Forms contains a rich set of controls to create custom toolbars and menus with modern appearance and behavior. The `ToolStrip`, `MenuStrip`, `ContextMenuStrip`, and `StatusStrip` controls can be used to create toolbars, menu bars, context menus, and status bars, respectively.

Controls

Some Windows Forms controls are designed for data entry within the application, such as `TextBox` and `ComboBox` controls. The `TextBox` is similar to the ABL `EDITOR` and `FILL-IN` widgets. Other controls display application data, such as `Label` and `ListView`. There are controls, such as `Button`, for invoking commands within the application. The `WebBrowser` control and managed HTML classes, such as `HtmlDocument`, let you display and manipulate HTML pages within your Windows Forms application. The `MaskedTextBox` control is an advanced data entry control that lets you define masks that accept or reject user input.

Layout

Several important controls in Windows Forms define the layout of controls on a display surface, such as a form or user control. `FlowLayoutPanel` lays out all the controls it contains in a serial manner, and `TableLayoutPanel` lets you define cells and rows for laying out controls in a fixed grid. `SplitContainer` divides your display surface into two or more adjustable parts.

Data viewing

Windows Forms defines a rich architecture for binding to data sources such as databases and XML files. The `DataGridView` control provides a customizable table for displaying data, and lets you customize cells, rows, columns, and borders. The `DataGridView` functions like the ABL `BROWSE` widget. Using the OpenEdge ProBindingSource control, you can also use these controls to view ABL data sources.

Components

Besides controls, there are other classes that provide visual or non-visual features to a Windows-based application. These classes are referred to as components. Components, such as `ToolTip` and `ErrorProvider`, enhance control capabilities or provide information to the user. With the `Help` and `HelpProvider` components, you can display Help information to a user of your applications.

Common dialog boxes

Microsoft provides several common dialog boxes that you can use to give your application a consistent user interface when performing tasks such as opening and saving files, manipulating the font or text color, or printing files. The `OpenFileDialog` and `SaveFileDialog` classes provide the functionality to display a dialog box that lets the user locate and enter the name of a file to open or save. The `FontDialog` class displays a dialog box to change elements of the Font used by your application. The `PageSetupDialog`, `PrintPreviewDialog`, and `PrintDialog` classes display dialog boxes that enable the user to control aspects of printing documents.

OpenEdge Controls

OpenEdge provides four notable controls, OpenEdge Window Container (`Progress.Windows.WindowContainer` class), OpenEdge ProBindingSource (`Progress.Data.BindingSource`, ABL User Control (`Progress.Windows.UserControl`), and ABL Inherited Control:

- **OpenEdge Window Container** — The Window Container is an OpenEdge .NET container control used for embedding an ABL window in a .NET form. This control, which is available in the Visual Designer Toolbox, can serve as an aid to migrating your existing GUI to GUI for .NET. This control is placed on a form and is designed to fill the area in which it is placed. The Window Container is similar to the ABL `FRAME`.
- **ABL User Control** — The User Control is the base class for building ABL custom user controls.
- **OpenEdge ProBindingSource** — The ProBindingSource is an OpenEdge .NET, non-visual control used to bind ABL data to one or more .NET UI controls that support data binding.
- **ABL Inherited** — The ABL Inherited control is used to create a customized control from a given class.

Note: If you do not find ABL User Control and ABL Inherited Control on the design canvas, add it from the Toolbox from **File** > **New** options.

OpenEdge Ultra Controls for .NET

The OpenEdge Ultra Controls for .NET (Ultra Controls) include additional controls that can be used for developing OpenEdge GUI for .NET applications. These Ultra Controls are a set of supported Infragistics NetAdvantage Windows Forms controls that provide built-in features different from those provided by the Microsoft controls. These built-in features allow developers to create forms with sophisticated user interfaces primarily by setting a rich set of properties.

Infragistics provides detailed documentation and help on these controls. While this documentation is written assuming use of a .NET language, the description of a control and its properties and events are all relevant for its use in ABL. You can access this documentation in the following ways:

- Visit the vendor's web site. For more information on OpenEdge Ultra Controls, see the API Reference Guide under Windows Forms at the following location: http://help.infragistics.com/NetAdvantage/WinForms
- Open the control in the OpenEdge Class Browser to see control definitions using a modified ABL syntax.
- Press **F1** after you place the control on the form within Progress Developer Studio for OpenEdge Visual Designer. The online help on the respective vendor site opens.

Third-party controls

There are many .NET Windows Forms control vendors providing full control sets or specialized controls, such as graphing and reporting. GUI for .NET supports third-party, Windows Forms controls. Prior to choosing a third-party control for your application, it is recommended that you search the OpenEdge Knowledge Base to see if there are known issues with the control.

ABL user controls

Similar to an ABL `FRAME`, an ABL user control is a container for .NET controls that inherits from `Progress.Windows.UserControl`, which in turn derives from `System.Windows.Forms.UserControl`. You can select and place existing .NET controls onto your user control. User controls can be added to a form just like any other .NET control.

User controls combine a group of controls. Examples of user controls are label and editbox combined, address block, table grid, and standard forms containing individual fields.

User controls are best designed with reuse in mind, to implement a common behavior, or for standard UI look-and-feel.

ABL-derived controls

An ABL-derived control extends an existing .NET control by customization. ABL-derived controls can be created in the Visual Designer and added to the Visual Designer Toolbox through `PROPATH`. ABL-derived controls can be added to a form just like any other .NET control.

An example of an ABL-derived control can be something as simple as an OK button. The OK button can be created as a class that inherits from the `System.Windows.Forms.Button` class. You can then set properties to change the button's shape, color, font, and so on. The button subscribes to the `Click` event which is then assigned to an event handler method. When dropped onto a .NET form, you can set the form's `AcceptButton` property to the OK button object reference.

If you are familiar with the object-oriented extensions to ABL, working with .NET objects looks very familiar since the syntax is the same. All .NET objects are referenced by specifying the namespace followed by the .NET object name.

Working with .NET forms and controls

In the code example that follows, the first line defines an object reference for a form as the `Progress.Windows.Form` class. The `Form` class is contained in the `Progress.Windows` namespace, as shown:

```
DEFINE VARIABLE FormObjectRef AS Progress.Windows.Form       NO-UNDO.
DEFINE VARIABLE helloBtn      AS System.Windows.Forms.Button NO-UNDO.

ASSIGN
  FormObjectRef = NEW Progress.Windows.Form( )
  helloBtn      = NEW System.Windows.Forms.Button( )
  helloBtn:Text = "Hello World".

FormObjectRef:Controls:Add(helloBtn).
MESSAGE helloBtn:Text VIEW-AS ALERT-BOX.
```

In this example two .NET objects are created, one for a form (`FormObjectRef`) and one for a .NET button (`helloBtn`). The `NEW` statement is used to create an instance of the .NET form and button objects and assign them to variables. The "Hello World" string is assigned to the button's Text property. Finally, the `helloBtn` button object is added to the form's control collection. Even though `helloBtn` is a .NET object, it can be used in ABL, for example displaying its property in a `MESSAGE` statement. The fact that `helloBtn` is a .NET object is transparent when working in ABL.

5

Events

A GUI for .NET, event-driven application blocks for input and processes events as they are received. 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.

The main communication between .NET and ABL is the invocation of ABL event handlers. These event handlers are associated with specific .NET events. .NET forms and controls fire events when an action occurs. ABL allows you to subscribe and unsubscribe event handler methods or procedures to .NET control events.

For details, see the following topics:

- .NET event loop
- Event handlers

.NET event loop

An application must use the .NET event loop when there is a chance it might use GUI for .NET. Incorporating the .NET event loop into your ABL application is Step 4 in the following list of steps that are required to create and manage a .NET form:

1. Instantiate the form and its controls using the `NEW` function for classes
2. Initialize the form, together with its controls
3. Subscribe handlers to appropriate .NET events

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

For more information about blocking on .NET forms, see *OpenEdge Development: GUI for .NET Programming*.

Event handlers

.NET supports an event model for classes. Like ABL class-based objects, .NET objects define events as members. An ABL application can respond to any .NET event using an ABL event handler. These event handlers are associated with specific .NET events. .NET forms and controls fire events when an action occurs. As a result all subscribed handlers get called.

ABL allows you to subscribe and unsubscribe event handler methods or internal procedures to .NET events. You can use the `Subscribe( )` method on a .NET event of a particular .NET class instance to subscribe event handlers written in ABL. .NET supports multiple subscribers attached to one event. Normally two parameters are passed to the event handler—the object firing the event and an object that provides event-specific arguments as properties.

In this example, the `FormObjectRef` object subscribes to the owner of the `Load` event, passing it the name of the event handler, the `FormObjectRefLoad( )` method:

```
FormObjectRef:Load:Subscribe(FormObjectRefLoad).

METHOD PRIVATE VOID FormObjectRefLoad
  (sender AS System.Object, e AS System.EventArgs).
  ...
END METHOD.
```

You can use the `Unsubscribe( )` method to detach an event handler from a control.

A .NET event handler can be either a method in a class (`.cls`) or an internal procedure. The main difference between using methods and internal procedures as .NET event handlers is that ABL verifies method event handler signatures at compile time (using strong typing), and verifies internal procedure event handler signatures at run time.

6

Data Binding

From a UI standpoint, data binding is an integration between UI components (controls) and data sources. As such, data binding establishes the connection between the ABL data source, which can be a ProDataSet, a query on a database table or temp-table, or a buffer, and the .NET UI component. We all have worked with data binding in ABL, but may not realize it since in many cases it is implied. Data binding with .NET is a bit more deliberate and more explicit. OpenEdge provides a built-in data binding object, the ProBindingSource, a class used as a conduit between the ABL data source and the .NET controls, and used by the .NET controls to access ABL data.

For details, see the following topics:

- Binding to a .NET control
- Binding to a data source
- Defining ProBindingSource tables and fields

Binding to a .NET control

From a UI standpoint, data binding is an integration between UI components (controls) and data sources. As such, data binding establishes the connection between the ABL data source, which can be a ProDataSet, a query on a database table or temp-table, or a buffer, and the .NET UI component. Examples of tight data binding in ABL include:

- Queries and browses
- Formats applied to UI widgets
- Code to display data

- Code to update data
- Statements that handle data transfer

For bound controls, data is supplied to the control by the binding source. The binding source interface provides methods and properties that a bound control can access and it also provides events to notify a bound control. Think of a binding source as a view to your data.

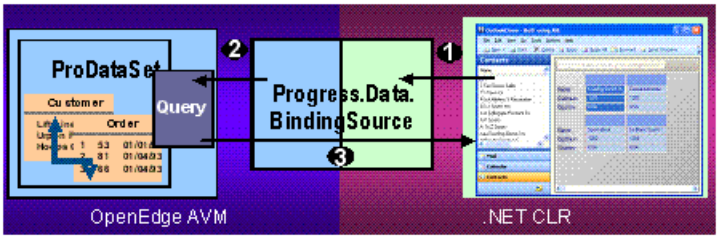
A critical component of a GUI for .NET application is the ability to access and update ABL data from the UI. .NET defines an API that all bound controls use to obtain their data. The OpenEdge data binding `Progress.Data.BindingSource` class (referred to as the ProBindingSource) is an OpenEdge .NET class used by .NET controls to access ABL data. This component, which is available in the Visual Designer Toolbox, is the best mechanism for binding your ABL data to your user interface.

The ProBindingSource is a non-visual component that is used at design time and run time. Once placed on a form, it is made available to all .NET controls as a data source. A binding source can be shared by multiple controls.

The following figure is a graphical representation of where the ProBindingSource fits between the .NET control and the data source, in this case a ProDataSet. UI controls are bound to the ProBindingSource which acts as a data source (#1). The ProBindingSource exposes an API to .NET controls to access the data.

At run time the ProBindingSource is then bound to a buffer, a query, or a ProDataSet (#2). The ProBindingSource spans the bridge (#3) between the OpenEdge AVM and the .NET CLR to pass data to the UI control and back again.

Figure 3: .NET-ABL Data Binding Model



The ProBindingSource provides ABL data-centric behavior to .NET controls, including the common behaviors of Creating, Reading, Updating, and Deleting data records (CRUD). Key benefits of using the ProDataSet as a data source for a ProBindingSource include:

- Automatic data synchronization
- Updating
- Batching
- Currency

A currency manager is built-in, controlling the position of the record cursor. This is in comparison to .NET, in which data source and currency management are handled separately.

Note: Data binding is not limited to visual objects. For example, the Ultra Controls scheduling controls can be data bound using an intermediary control, which then provides data to the visual objects.

Binding to a data source

The ProBindingSource supports three types of ABL data source objects:

- Buffer on page 43
- Query on page 43
- ProDataSet on page 43

Buffer

An ABL buffer acting as a data source supplies one record at a time to its bound, single-value control, such as a text box or a toggle box. The ProBindingSource `Position` property is always 0 when using a buffer in this way.

Query

An ABL query acting as a data source can be either static or dynamic. The query can also represent a join between multiple tables. A query that is part of a ProDataSet is also a valid data source. Key benefits of using an ABL query as a data source are that it provides automatic updating, batching and currency control.

When binding a control to a query, PSC recommends use of the `PRESELECT` option on the query, which is an optimized means of providing the binding source with the query record count. For dynamic queries, the AVM enforces use of the `PRESELECT` option.

PSC also recommends that a query used as a data source be defined using the `SCROLLING` option. This option specifies that you can jump to a location within the list of records that satisfy the query by using the `REPOSITION` statement.

ProDataSet

A ProDataSet, acting as a data source to the ProBindingSource, can provide a hierarchy of tables to the bound .NET control. The ProDataSet's parent buffer controls what tables are available through the ProBindingSource. If the ProDataSet contains more than one top-level table and the parent buffer table is not specified, the first top-level buffer acts as the parent table.

You should only bind to a ProDataSet directly (as opposed to one of its queries), if you want to show more than one level of hierarchy in a single control, such as an UltraGrid or TreeView.

Defining ProBindingSource tables and fields

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, the validation phrase, or the help text for a field. The .NET controls can only access what the binding interfaces in the `Progress.Data.BindingSource` class make available.

The task of binding a data source to the ProBindingSource control includes the identification of data source tables and fields that will be exposed to the bound .NET control. Since the data source is provided at run time, the binding source may provide a design-time schema that matches the schema used at run time. So, how is the schema supplied to the binding source? The ProBindingSource Designer tool, available in the Visual Designer's Design view, is used to define what data source tables and fields are available to be bound to a control. Schema information can either be entered manually or imported from an XML Schema file. Similar to an OpenEdge Data Definition file (`.df`), in this context an XML Schema file defines table name, field name, field data type, field display format, field label, and so on as XML data.

You can create an XML Schema file containing the definition of a ProDataSet, a temp-table, or a temp-table buffer object using the ABL `WRITE-XMLSCHEMA( )` method. This method uses the industry-standard XML Schema Definition (XSD) language. The XML Schema file that is produced may contain more fields than you need. After importing an XML Schema file, you can eliminate unwanted fields in the ProBindingSource control by removing them with the ProBindingSource Designer. Alternately, you can allow the ProBindingSource to contain all the data source fields, so it can be reused and bound to a number of controls.

7

Service Layers

Implementing an OpenEdge GUI for .NET does not mean that you have to rewrite your application code. You should review your application architecture and decide which, if any, components could be modified to take full advantage of the Ultra Controls and the underlying object-oriented extensions to ABL. Materials on Progress Communities and presentations given at Exchange user conferences and other mediums over the past several years have described an approach to building applications called the OpenEdge Reference Architecture (OERA). Most of those existing materials focus on the data management side of an application, describing data management components such as Business Entities that encapsulate access to a particular set of related application data.

For details, see the following topics:

- Business Entities
- Service Adapters and Service Interfaces

Business Entities

A Business Entity is a reusable, modular component that encapsulates all access to a particular set of related application data identified by its use within your application, for example, a Customer or an Order. A Business Entity becomes a type of service, one that can be called from any part of your application needing to retrieve or update data.

One advantage of encapsulating access to your data along with all the business logic that applies to it is that you can substitute a new user interface without having to change your business logic in any way. For this reason, when you consider how to integrate a new generation of user interface based on the GUI for .NET with your existing data and business logic, you should consider how best to keep the data and logic that you already have, while separating that logic from the specifics of the UI. While there is no requirement to re-architect your application to use the GUI for .NET, you should take advantage of the opportunity where you can to make a clean separation of the "client" side of your application (where the user interface is defined), from the "server" side of your application (where the data and business logic are encoded). This is true whether your application is actually divided between OpenEdge client sessions and AppServers, or whether it uses OpenEdge client-server support within a single session.

Many of the coding examples you will be seeing in documentation materials for OpenEdge 10.2 show GUI for .NET forms and their supporting code interacting with Business Entities and other data management components. This is referred to in OERA terminology as the Presentation Layer. For examples of GUI for .NET forms and their supporting code, see *OpenEdge Getting Started: Introducing the Progress Developer Studio for OpenEdge Visual Designer*.

Service Adapters and Service Interfaces

You will also see examples that refer to the concept of a *Service Adapter* and a *Service Interface*. The Service Adapter is simply a layer of indirection between the Presentation on the client side, and Business Entities and other data management components on the server side. A Service Adapter takes responsibility for knowing how and where to make a request for data, so that this connection is not hard-wired into a particular user interface, which is only concerned with the definition of the data as it is displayed and updated, and not where the data comes from.

Likewise, a Service Interface defines an entry point for each accessible operation in each Business Entity or other business component. A Service Adapter on the client side can make a call to a Service Interface on the server side (again, whether the application is physically separated into client and server sessions or not). The Service Interface takes responsibility for identifying and calling the business component, so that any details of how this is done are kept independent of the call itself. These different application layers maximize the reusability, flexibility, and ease of maintenance of all the different parts of your application logic.

The following figure shows where the Service Interface and the Service Adapter fit in this layered architecture.

Figure 4: Service Adapter and Service Interface

