Progress®



Develop Object-oriented ABL Applications

OpenEdge®

Copyright

© 2021 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.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP 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, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, 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.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Index.....267

Preface

Purpose
The purpose of this content is to provide a deeper look at object-oriented ABL topics.

Audience
The audience is those who have completed the basic and advanced ABL developer guided journeys and object-oriented ABL developer guided journey.

Organization
The organization of this content is based on object-orient concepts of classes, inheritance, and interfaces. This content is intended to expand the object-oriented programming skills of ABL developers. Topics include how to design objects, program with class-based objects, comparisons to procedure object, how to deploy classes and an expanded look at overloading methods and constructors .

Documentation Conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content.

Purpose
The purpose of this content is to provide a deeper look at object-oriented ABL topics.

Audience
The audience is those who have completed the basic and advanced ABL developer guided journeys and object-oriented ABL developer guided journey.

Organization
The organization of this content is based on object-orient concepts of classes, inheritance, and interfaces. This content is intended to expand the object-oriented programming skills of ABL developers. Topics include how to design objects, program with class-based objects, comparisons to procedure object, how to deploy classes and an expanded look at overloading methods and constructors .

Documentation Conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content.

1

Object-oriented programming and ABL

Object-oriented programming is a popular programming model. There are standard concepts that all object-oriented programming languages support, such as encapsulation, inheritance, and strong typing. OpenEdge provides language extensions to support these standard object-oriented programming concepts in a way that is commonly available in other object-oriented languages, such as Java.

These object-oriented extensions complement the basic power of ABL, and its procedural programming model, with an alternative programming model that seamlessly coexists with applications written using the procedural programming model. Thus, the object-oriented extensions continue to support the core features of ABL—business orientation, high productivity, and ease of use. These core features represent the true power of ABL that differentiate it from other development languages.

For details, see the following topics:

- Support for classes in ABL
- Programming models in ABL
- Overview of object-oriented programming
- Overview of class-based ABL
- Programming conventions for classes

Support for classes in ABL

The goal of object-oriented programming support in ABL is to extend the language to provide a cohesive and standard object-oriented programming model within ABL while continuing to fully support the programming model available in previous releases of OpenEdge. To this end, ABL provides support for programming with classes in addition to, and together with, its support for programming with procedures. This object-oriented language support is a natural basis for writing applications that conform to modern application frameworks.

A class, like a procedure, contains or references data (*state*) and behavior that operates on that data. Thus, a common element between ABL procedures and classes is the ability to define objects. An *object* is a self-contained unit of code that encapsulates an instance of well-defined state and behavior.

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, which allows the ABL Virtual Machine (AVM) to automate the management of these relationships at run time. Thus, unlike a persistent procedure, a *class* defines a well-structured *type* (or *object type*) that ABL recognizes at compile time and that you, at run time, can realize as an object that behaves according to the definition of that type.

Classes, by definition, support a variety of object-oriented features that help to organize state and behavior in an application. ABL provides limited support for these object-oriented technologies with persistent procedures. The variables, buffers, temp-tables, and so on, defined in the main block of a persistent procedure, represent the object data, and the internal procedures and user-defined functions represent the object behavior. However some of the other standard object-oriented features are not supported using persistent procedures.

The language enhancements for classes provide a complete set of object-oriented constructs that support all the standard behavior for programming with classes expected within the object-oriented development community. Anyone with an object-oriented programming background will feel comfortable programming with classes in ABL. Anyone who wants to learn about object-oriented programming can pick up any book on object-oriented programming and be able to apply the object-oriented concepts in that book using ABL syntax that supports classes.

Advantages of ABL classes

Support for object-oriented programming in ABL provides the following benefits for ABL application development:

- Classes support a powerful programming model by encapsulating related functionality into objects. The benefit of organized code is especially important for maintenance, where changes or enhancements can be limited to the objects that are affected by the change.
- Classes enhance code reuse. Common code can be put into one class and shared among other related classes. The related classes can provide specialized behavior whenever necessary to extend the initial class functionality, as well as the functionality of other related classes.
- Classes provide strong typing that enables ABL to do compile-time checking in order to catch errors before the code is deployed or executed. This compile-time checking verifies all interfaces for correctness, not just the ones that are accessed during run-time testing, as with procedures.
- The object-oriented support using classes maps much more closely to the constructs used in a service-oriented architecture (SOA) and to those used by modeling tools and other design tools that are increasingly used to provide a basis for application design, and even code generation.
- ABL support for classes includes access to .NET objects, which allows you to work with .NET object types on Windows together with ABL class-based object types. Using the OpenEdge GUI for .NET, you can use .NET form and control objects to create a .NET graphical user interface for all Windows-based GUI ABL clients. Using non-GUI .NET objects, you can also access non-GUI .NET features in all ABL sessions that

run on Windows, including non-GUI sessions such as character-mode clients and application server agent sessions. For more information, see *Use GUI for .NET Objects in ABL Applications*.

Foundations of ABL classes

Classes are supported by a number of specific constructs within the language. First and foremost is the CLASS statement, which can be used to define a user-defined type (new object type) in ABL. ABL also provides a set of built-in classes for various features. Classes defined by the CLASS statement contain both state and behavior.

ABL class definitions support inheritance of state and behavior from one class to another. That is, you can define a new class with reference to an existing class so that the state and behavior of the existing class appear to be part of the new class. At the same time, you can define additional state and behavior in the new class that does not exist in the inherited class. Thus, multiple classes can be related to one another in a hierarchy formed by their defined inheritance relationships.

Classes can also implement one or more interfaces, each of which you can define using an INTERFACE statement. This statement also defines a user-defined type in ABL. An *interface* declares a common public mechanism for accessing state and behavior that one or more classes can define and that these classes might not inherit from another common class. Interfaces allow you to more easily define and manage common state and behavior that might be implemented differently in different classes and for different purposes. An interface can also be defined to inherit one or more interfaces. This composes the interface from the definitions of previously defined interfaces, but is not otherwise similar to class inheritance.

Note: In this content, wherever inheritance is described, it refers to class inheritance unless otherwise noted.

In addition to allowing you to define your own interfaces using the INTERFACE statement, ABL provides built-in interfaces to manage common state and behavior for some of its built-in classes. Like a class, an interface also represents an object type, but never contains an implementation of that type. Only a class can implement the type specified by an interface.

Thus, a class type or interface type represents a data type that you can specify in the language anywhere that a built-in data type (such as INTEGER) can be specified. In this way, the support for classes and interfaces in ABL is very similar to classes and interfaces in Java and other object-oriented programming languages.

Any given ABL object built from classes and interfaces can be seen as more than one type, depending on the hierarchy of classes and interfaces used to define the object. In other words, each class or interface type that you use to construct an object provides a different type through which you can access the object. Any consumer of the object only needs to access the object as a supported class or interface type in order to use the functionality supported by that type. Thus, depending on the type you use to access the object, the same object can provide different subsets of its total functionality.

A single CLASS or INTERFACE statement identifies a source code file that contains a class or interface definition and not the definition of an ABL procedure. Within a class or interface definition, there are several language statements that are distinctive to classes or interfaces (respectively) and which can only be used within them. On the other hand, you can use the vast majority of ABL syntax within classes, and for the most part, you can use them in exactly the same way as they are used in procedures.

This means that there is a dichotomy in how you must think about classes and procedures in ABL. On the one hand, there is a clear and absolute distinction between classes and procedures, and the compiler can tell from the presence of a CLASS or INTERFACE statement in a source file which kind of object it is dealing with. On the other hand, the majority of the programming that you do within a class can be much the same as within a procedure. This means that if you are already thoroughly familiar with how to program ABL procedures, programming with classes can quickly become as familiar and natural.

Programming models in ABL

ABL supports two basic kinds of user-defined objects—objects based on procedures and objects based on classes. Although you use much of the same syntax to program each kind of object, each of these object categories supports an entirely different programming model:

- Procedure-based model on page 12
- Class-based model on page 12
- Compare procedure-based with class-based models on page 13

Procedure-based model

Procedure objects support a programming model where you design and instantiate (create) objects based on persistent procedures. These persistent procedures maintain a run-time context that can be accessed by other objects. This run-time context can include variables and various types of handle-based objects that provide the procedure object's data, and internal procedures, user-defined functions, and named events that provide the procedure object's behavior.

With procedure objects, the state and behavior in one object has no well-defined relationship to the state and behavior in another. You establish any such relationships at run time by invoking ABL statements to access the state and behavior in another object. You can set up object hierarchies, but again, these relationships depend entirely on statements at run time to maintain these relationships. As a result, you must organize these procedure objects using interfaces that can, without care, easily become inconsistent, and you must manage the lifetimes of these objects individually. You must account for any relationships that you design between them any time that you invoke code to access state or behavior between them.

Class-based model

Class-based objects support a programming model where you design and instantiate objects based on strongly-typed classes. Like procedure objects, classes maintain a run-time context that can be accessed by other objects. For class-based objects, this context also can include various types of data (data members and properties), but instead of internal procedures, user-defined functions, and named events, a class-based object's behavior is provided by methods and class events. Thus, a class-based *method* is a unit of executable code that has features in common with both internal procedures and user-defined functions, as well as features unique to methods.

A *class event* is a mechanism similar to a named event that allows you to send notification of a particular condition that an application can respond to in different ways. A *data member* is a variable, buffer, temp-table, or similar data element that is defined for a class at the same level as its methods. A class-based *property* is similar to a variable data member, but its access can be further controlled by specifying if it is readable, writable, or both and defining any behavior to be invoked when the property is read or written. Each method, class event, data member, and property defined within a class is a *member* of that class, and the members defined for a class collectively contribute to the type definition of that class.

Note: Class-based methods are analogous to, but entirely different from the built-in methods that ABL provides on handle-based objects, such as procedure, buffer, and query objects (among others). The main difference is that class-based methods can be user defined, and they are strongly typed to the classes in which they are defined.

With class-based objects, the state and behavior in one object has a well-defined (*strongly typed*) relationship to the state and behavior in another class-based object. You establish these relationships at compile time by using syntax designed to associate class-based objects with each other in a well-defined hierarchy that allows objects to share class members that are defined in other objects. As a result, you can organize class-based objects with well defined interfaces before you even compile the objects.

If you change the interface or relationship between the objects, many errors can be caught at compile-time that might not be caught for some time using procedure objects at run time. Because relationships among class-based objects can be defined at compile-time, ABL also supports standard management features for creating and deleting (*destroying*) these objects in a consistent and less error-prone manner. When you access state or behavior in other class-based objects, you can have greater confidence that this access is both permissible and appropriate for your designed task.

Classes can also be abstract. An *abstract class* can have much the same definition as a non-abstract class, except that it can define abstract methods, properties, and events. *Abstract members* of a class are really prototypes for members that an inheriting class must implement. You also cannot create instances of an abstract class, whether or not it defines abstract members for implementation by an inheriting class. Thus, an abstract class allows you to define and manage common state and behavior like any class, but also requires one or more inheriting classes to complete its functionality in a unique way.

Note that an interface type (previously described) can also provide prototypes for methods, properties, and events that a class must implement. However, the difference is that an abstract class provides some implementation that an inheriting class must complete, while an interface provides no implementation that an implementing class must provide. In practice, an interface typically defines members that represent some discrete functionality that is not necessarily related to other features of a class that implements it (for example, properties and methods intended to implement a list structure or collection). An abstract class typically defines abstract members that are related to the overall class functionality that requires some customization by an inheriting class.

The choice of whether to use interface types or abstract class types to define class interfaces is a design decision for any object-oriented application.

Compare procedure-based with class-based models

In general, the run-time nature of procedure-based programming can support a more dynamic coding model than you might write in order to accomplish the same task using class-based programming. Class-based programming, on the other hand, can support a simplified program structure that is much easier to maintain and that helps to identify and more easily reuse code from one object to another. However, because you can almost freely mix class-based and procedure objects in the same application, you can choose the implementation model that best meets your requirements for any given programming task.

Many successful applications have been built, and will continue to be built, with procedures, including the addition of object-based extensions using persistent procedures and the super procedure mechanism to create a run-time inheritance chain. Generally, Progress Software Corporation recommends that you consider using classes for new development. They provide much greater assurance that ABL code that compiles will also execute successfully, whereas many code paths that lead through a set of related procedure instances can be verified only by testing them at run time.

There are certainly circumstances, however, where the use of procedures might well be preferred. Precisely because procedures give you the flexibility to assemble a procedure hierarchy at run time, super procedures allow you to combine procedures in a way that is data-driven or determined by application logic at run time. Likewise, the dynamic `RUN VALUE(`*`procedure-variable`*`)` statement and the `DYNAMIC-FUNCTION` built-in function let your application invoke procedures and functions whose names are determined at run time, which can also support data-driven operations.

Throughout this content, there are frequent comparisons between class-based programming and procedure-based programming. To simplify, the term *class-based* refers to objects and programming that is based on ABL classes, and the term *procedure-based* refers to objects and programming that is based on ABL procedures. There are a number of topics with the common title, "Comparison with procedure-based programming." where the use of a specific class-based programming feature is compared to the use of procedure-based programming that provides the same or similar functionality.

Overview of object-oriented programming

To understand the object-oriented features of ABL and how best to use them, it is helpful to understand the key concepts of object-oriented programming. This section is a brief introduction to object-oriented programming. It also compares these concepts with features of procedural programming used to accomplish the same goals, using ABL.

Object-oriented programming is a programming model organized around objects rather than actions. Conventional procedural programming normally takes input data, processes it, and produces output data. The primary programming challenge is how to write the logic. Object-oriented programming focuses on the objects that you want to manipulate, their relationships, and the logic required to manipulate them.

The fundamental advantage of object-oriented programming is that the data and the operations that manipulate the data are both encapsulated in the object with a well-defined interface. Objects are the building blocks of an object-oriented program. An application that uses object-oriented technology is basically a collection of objects that interact through their interfaces.

The concepts of a *type*, *class*, *interface*, and *object* are closely related but it is important to understand the difference between these four terms. A type is a name that identifies specific members of a class, which can include methods, properties, data members, and events. A class defines the implementation of a type and its class members. An abstract class is essentially a type with an incomplete implementation. An interface (described previously) also defines a type that identifies certain class members (properties, methods, or events) that a class must implement. An object is an instance of a class whose type can be represented as any class or interface that contributes members defined in the object's class hierarchy. Objects have a life cycle in which they can be repeatedly created, used, and destroyed during an ABL session.

Classes in ABL provide support for a basic set of object-oriented concepts. To illustrate these key concepts, this manual includes figures that contain a simple and informal pseudo-code. To simplify this pseudo-code, the data types of data elements and method parameters, other than specific references to classes and interfaces, are implied from context. For example, the name of a data element can serve to denote its type.

The following sections describe these basic object-oriented concepts:

- Encapsulation on page 15
- Inheritance on page 17
- Delegation on page 20
- Polymorphism on page 21
- Method overloading on page 25
- Strong typing on page 26
- Glossary of terms on page 27

Encapsulation

You implement a class by combining both state and behavior together in a well-defined unit, which is the implementation of a specified class type. The data members (variables, buffers, etc.) and properties represent the object state for a given class. The methods and events represent the object behavior for a given class, which define how a class, and code outside the class, can interact with its own data members and properties, and any other data it can access. Objects invoke each others' behavior by sending messages to each other according to the interface defined by each object's type. A *message*, in this context, identifies a specific unit of behavior defined for an object. Thus, you can send a message to an object by calling a method on that object or by publishing a class event that invokes an event handler in that object.

Note: A message is not necessarily a physical data packet, but is a concept that conveys the idea that one object is using some means to request that an action be performed on the part of a second object, which involves a specified subset of the second object's data.

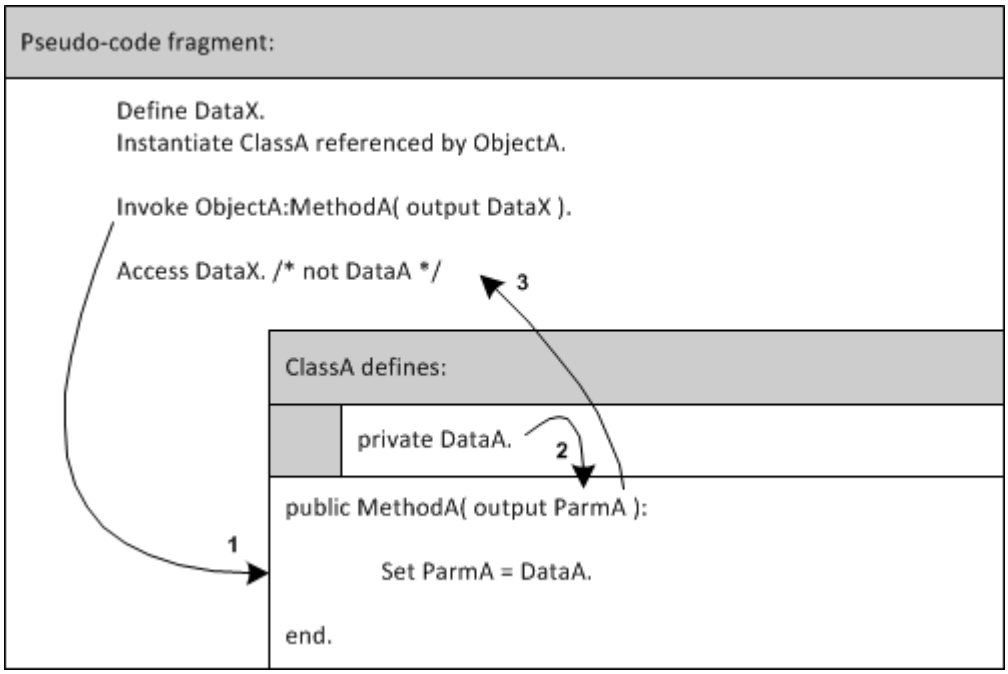
Through the well-defined interface of its class type, an object can satisfy its contract to class consumers while keeping details of its implementation private. The general mechanism that a class uses to provide access to resources according to its contract, while maintaining the privacy of its implementation, is referred to as *encapsulation*. Because of the goal to maintain the privacy of a class's implementation, encapsulation is also known as *information hiding*.

Thus, depending on its interface, a class can allow some of its code and data to be accessible from outside the class hierarchy and some of it to be inaccessible. With complete encapsulation, you hide all data of a class from direct access, and you define a set of properties, methods, and events as the interface to that data. By enforcing access to the data of a class through such a well-defined interface, you can change the implementation of the class at any time without affecting any other code that accesses this interface.

To help define a class's interface, you can define an *access mode* for every class member (`PRIVATE`, `PACKAGE-PRIVATE`, `PROTECTED`, `PACKAGE-PROTECTED` or `PUBLIC`) that controls where and how the member can be accessed. A `PRIVATE` method can be accessed within the class where it is defined. The `PRIVATE` method of an instance can also be accessed by another instance of the same class. A `PROTECTED` method can be accessed within the class where it is defined and within any class that inherits from that class. A `PACKAGE-PRIVATE` method can be called from within the class and any class within its package. A `PACKAGE-PROTECTED` method can be called from within the class, any class within its package, and from within any subclass that inherits the class. A `PUBLIC` method can be accessed within the class where it is defined, from within any class that inherits from that class, and from outside any instance (object) of that class (that is, from within a procedure or a class instance that is outside the class hierarchy where the `PUBLIC` method is defined). You can similarly define access modes for members, properties, and events of a class, and you can further define properties so that they can be read and written according to different access modes.

The following figure shows a simple example of encapsulation.

Figure 1: Encapsulation example



In this figure, ClassA defines DataA as `PRIVATE`. DataA can only be accessed from within the class definition or by an instance of that class. ClassA also defines a `PUBLIC` MethodA that provides indirect access to DataA through the method's parameter, ParmA. Thus, after instantiating the class, the pseudo-code fragment uses ClassA as shown with the numbered arrows:

1. Invokes its MethodA on the ClassA instance referenced by ObjectA.
2. MethodA assigns the value of DataA to ParmA, which the fragment returns as an output parameter.
3. Using DataX, the fragment then accesses the value of DataA without actually accessing DataA, itself.

So, you can define a class interface so that some of its data members can be accessed directly and others accessed only indirectly, through methods of the class. In addition, properties, which allow you to define different access modes for read and write access, can also be defined as read-only (so they cannot be written) and write-only (so they cannot be read). You can define special methods for properties that execute when they are read or written, which effectively hides the implementation of the data they access. Therefore, to define a class with complete encapsulation, you would define all of its data members as `PRIVATE` or `PROTECTED`, and define `PUBLIC` properties, methods, and events to access any of that data from outside the class.

Comparison with procedure-based programming encapsulation

ABL procedures also provide a degree of encapsulation. By default, the variables and other data definitions of a procedure are private and cannot be accessed directly from other procedures. Procedures can use shared variables and other mechanisms to share data between them, which effectively breaks encapsulation. However, to effectively encapsulate its data, a procedure can define public internal procedures and user-defined functions to allow controlled access to this data. Procedures and user-defined functions are public, by default.

To fully encapsulate some of its behavior, a procedure can also define private internal procedures and user-defined functions that cannot be executed from another external procedure.

However the class-based mechanism for defining private, protected, and public methods, data members, and properties allows you to control encapsulation with far more precision and consistency than is possible with procedures.

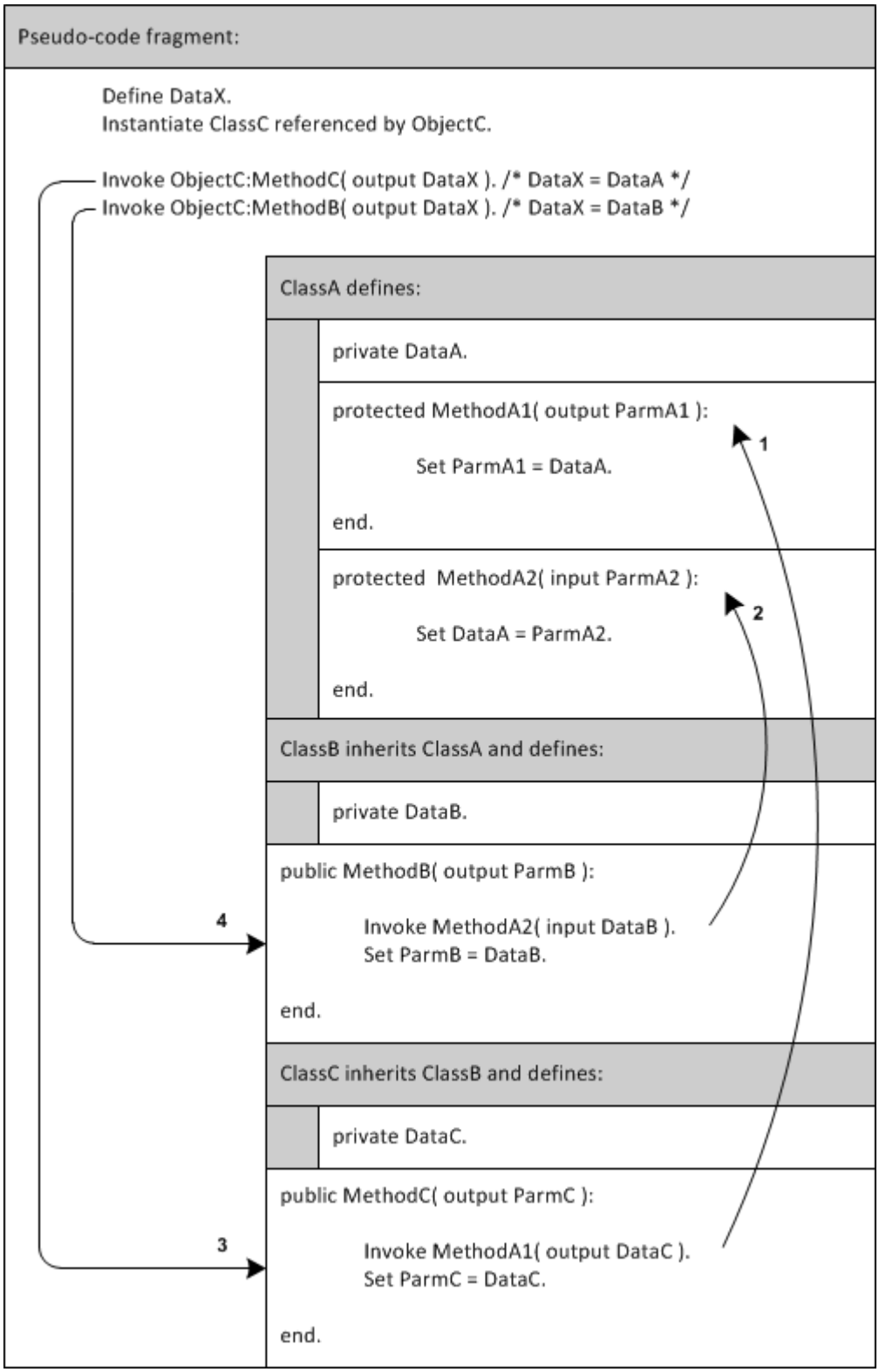
Inheritance

Code reuse is one of the great benefits of object-oriented programming. Procedural programming provides code reuse to a certain degree—you can write a procedure and then use it as many times as you want. However, object-oriented programming goes an important step further by allowing you to define relationships between classes. These relationships facilitate code reuse as well as better overall design, by organizing classes and factoring out common elements of related classes. Inheritance is one means of providing this functionality.

Inheritance allows a given class to inherit data members, properties, methods, and events from another class—that is, it allows a given class to adopt members of another class in such a way that the given class appears to have defined these members within itself. If one class explicitly inherits from another, the inherited class is a *super class* of the class that inherits from it. Conversely, any class that inherits from a super class is a *subclass* (also known as a *derived class*) of the specified super class. The entire series of super classes and the one most derived class that inherits from all of them form a chain of inheritance that represents the *class hierarchy* of that most derived class. Thus, both a class and any of its super classes can provide data members, properties, methods, and events for the definition of that class.

The following figure shows an example of inheritance, where ClassB is the immediate super class of ClassC, and ClassA is the immediate super class of ClassB. The derived class, ClassC, thus inherits class members from both ClassA and ClassB.

Figure 2: Inheritance example



So, when the pseudo-code fragment in the figure instantiates ClassC, the resulting object (ObjectC) contains all three class definitions according to the specified class hierarchy. ClassA defines protected methods that are inherited and accessible only within the class hierarchy itself, ClassB defines a public method inherited by ClassC, and ClassC also defines a public method. Thus, the object defined as an instance of ClassC effectively provides both public methods for access by the pseudo-code fragment. The numbered arrows show the following characteristics that define ClassC and its hierarchy and how they are accessed by the pseudo-code fragment using the ClassC instance:

1. ClassC's public method, MethodC( ) calls ClassA's protected method, MethodA1( ), which sets the private data of MethodC( ), DataC, from the value of DataA. MethodC( ) then passes the new value of DataC as its output parameter.
2. ClassB's public method, MethodB( ), calls ClassA's protected method, MethodA2( ) to set DataA from its own private data, DataB, and passes the value of DataB as its output parameter.
3. The pseudo-code fragment calls MethodC( ) on the ClassC instance, ObjectC, returning the value currently set for DataA.
4. The pseudo-code fragment calls MethodB( ) on the ClassC instance, ObjectC, changing the value of DataA and returning that value, which is currently set for DataB.

Again, these classes also encapsulate all of their data, and the class hierarchy provides well managed access to it. So, ClassC's inheritance as the most derived class allows access to all public and protected methods in the hierarchy, and the protected methods of ClassA allow all of its derived classes to access and modify the value of its private data without providing direct access to the private data itself.

One of the major design goals in object-oriented programming is to factor out common behavior of various classes and separate the common behavior into a super class, which defines behavior for each of its subclasses to implement. This super class contains all the members that are common to the subclasses that inherit from it. This inheritance relationship is defined at compile-time and any modifications to a super class are automatically propagated to each of its subclasses. In other words, inheritance creates a strong, permanent coupling between a super class and its subclasses.

A subclass can also override the behavior of its super class by providing a method with the same signature but different behavior from the super class. The overriding method can access the behavior in the super class method, augmenting the super class behavior with pre- or post-processing. This pre- or post-processing contains the subclass's own unique behavior for the overriding method.

Note: Method overriding is fundamental to implementing polymorphism. For more information, see Polymorphism on page 21.

ABL, like Java and some other object-oriented languages, uses a *single-inheritance model*. This model allows a subclass to explicitly inherit from only a single super class. This single super class can then inherit from a single super class, and so on. Thus, a subclass explicitly inherits from its single, immediate super class and implicitly inherits from any single-inherited super classes above it in the class hierarchy. The very top of the class hierarchy is the *root class*, which is the super class that all subclasses implicitly inherit. ABL provides a built-in root class, `Progess.Lang.Object` (see Use the root class: Progress.Lang.Object on page 91). A subclass and its class hierarchy can also implement one or more interface types that the subclass explicitly specifies, including any interfaces that these interface types inherit. However, interfaces cannot inherit member prototypes from a class, nor can they, themselves, be inherited by a class.

Comparison with procedure-based programming inheritance

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. One internal procedure can invoke another internal procedure of the same name in its super procedure with the RUN SUPER statement, or similarly for user-defined functions using the SUPER built-in function. The key difference between the use of super procedures and a class hierarchy is that the super procedures are related only at run time. ABL has no way to anticipate or validate the interactions between the procedures, and must search the procedure stack to determine if a "super" version of a procedure exists, and where. By contrast, the compiler can validate every reference from one class to another in the same hierarchy, which can then be established reliably at run time.

Delegation

Delegation uses composition to build a class from one or more other classes without relating the classes in a hierarchy. In many cases, one class needs to invoke behavior in another class that is not part of its class hierarchy. A class does this by maintaining a reference to the other class and invoking the other class's public methods or accessing its public data members or properties. When a class references another class this way, it is a *container* class for the other class that it accesses as a *delegate*. Thus, delegation is a relationship between classes where one class forwards a message it cannot otherwise handle to another class (its delegate).

Because the container class delegates behavior to a separate delegate class, the behavior is not automatically accessible from outside the container class. Unlike an inheritance relationship, delegation requires the container class to define a stub for the message in the form of a method to allow other classes to access the behavior of the delegate. Thus, the stub method in the container class is a method that is typically of the same name as the effective method in the delegate. Because a delegation relationship is established entirely by reference to the delegate, delegation offers the flexibility to easily change the referenced delegate at run time without any required change to the container. This there by provides an instance of the container class with a form of dynamic inheritance.

You can create your own container and delegate classes in ABL. You can also use interfaces to help enforce consistency between the method stubs implemented in a given container class and the effective method implementations defined in a corresponding delegate class.

Comparison with procedure-based programming delegation

Much of this description is very similar to how procedures interact. When a procedure named A.p runs an internal procedure in a procedure handle for B.p, it is effectively delegating the behavior of the internal procedure to B.p. In this situation, A.p is similar to the container class in the object-oriented model, and B.p to the delegate. And clearly, a third procedure named C.p cannot invoke the delegate's internal procedure directly through A.p, unless A.p itself has an internal procedure or function definition that runs the effective behavior in the handle to B.p. The key distinction with classes is that the compiler verifies all of the references between classes, whether they are within the same class hierarchy or not. Thus, ABL has a much more detailed and complete definition of everything that is controlled by a container class than it can have for a procedure that runs behavior defined in another procedure.

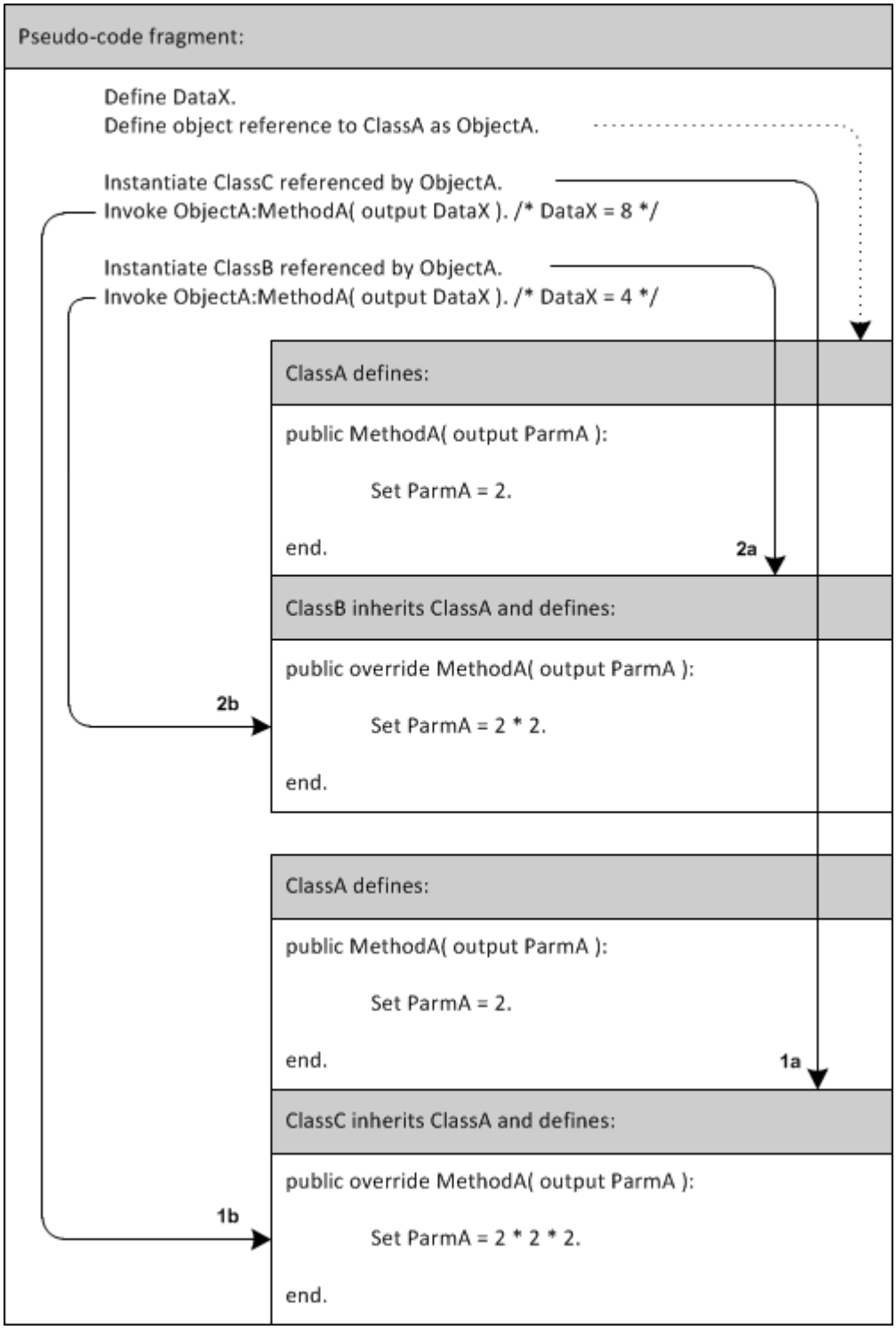
Polymorphism

Polymorphism is one of the most powerful advantages of object-oriented programming, and it relies fundamentally on inheritance and overriding. When a message is sent to an object of a class, the class must have a method defined to respond to that message. In a class hierarchy, all subclasses inherit the same methods from a common super class. The same message sent to each subclass invokes the same inherited method. However, because each subclass is a separate entity, it can implement a different response to the same message by overriding the specified super class method with one that implements the unique behavior of the subclass. In other words, *polymorphism* allows different subclasses based on the same class hierarchy to respond to the same message in different ways.

Thus, polymorphism allows a given message sent to a super class to be dispatched to an overridden method that provides different run-time behavior depending on the particular subclass that overrides the method in the super class. Therefore, polymorphism simplifies programming, where invocation of the same method produces a different result depending on the subclass that implements the method.

The following figure shows an example of polymorphism where ClassA defines a MethodA( ) that is inherited and overridden by ClassB and ClassC in two separate class hierarchies, respectively. The pseudo-code fragment takes advantage of this polymorphism by defining a single object reference (dotted arrow) to ClassA (ObjectA) that can access both overrides of the method.

Figure 3: Example of polymorphism with method overriding



The pseudo-code can reference an instance of both ClassB and ClassC and invoke MethodA( ) as if it were an instance of ClassA. However, the implementation of MethodA( ) that it executes is the one that overrides the ClassA method in the instance actually referenced as ObjectA.

So, as the numbered arrows show:

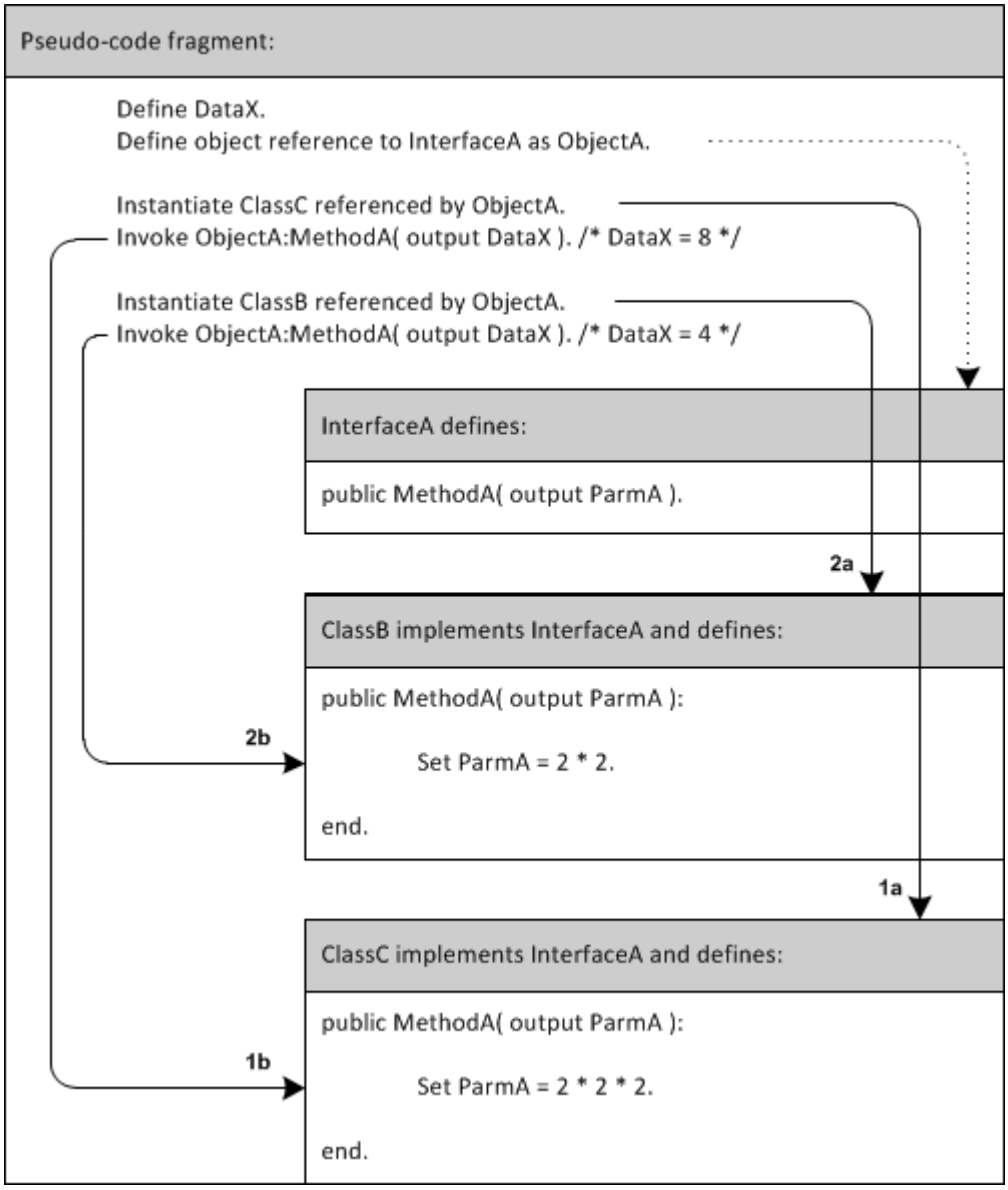
1. The pseudo-code instantiates ClassC and sets ObjectA to reference the instance (**1a**). So, when the code invokes MethodA( ) on ObjectA (**1b**), it is ClassC's override that executes.
2. The pseudo-code instantiates ClassB and sets ObjectA to reference the instance (**2a**). So, when the code invokes MethodA( ) on ObjectA (**2b**), it is ClassB's override that executes.

As a practical example, you might have a system with many different shapes, where `Shape` is a super class. However, a circle, a square, and a star are each drawn differently. By using polymorphism, you can define a `Draw( )` method in the `Shape` super class, then send the super class a message to invoke this `Draw( )` method, and each subclass of `Shape` (`Circle`, `Square`, or `Star`) is responsible for drawing itself using its own implementation of the `Draw( )` method. There is no need, when invoking the `Draw( )` method, to know what subclass of `Shape` is responding to the message.

Another kind of polymorphism that provides features similar to method overriding relies on interfaces alone (without super classes) to specify common methods that are implemented differently by different classes. This allows a given message sent to an object referenced as an interface type to be dispatched to an implemented method that provides different run-time behavior depending on the particular class that implements the interface type.

The following figure shows an example of polymorphism where ClassB and ClassC each provide different versions of the MethodA( ) declared by an interface, InterfaceA, that they both implement. The pseudo-code fragment takes advantage of this polymorphism by defining a single object reference (dotted arrow) to InterfaceA (ObjectA) that can access both implementations of the method. Thus, this example provides exactly the same functionality using interfaces as does the overriding example.

Figure 4: Example of polymorphism with interfaces



Because both classes implement InterfaceA, when the pseudo-code references each instance of ClassB and ClassC using the ObjectA reference, it can invoke MethodA( ) in exactly the same way for the two different class instances. However, the version of MethodA( ) that executes is the one that is implemented by the class instance actually referenced as ObjectA.

So, as the numbered arrows show:

1. The pseudo-code instantiates ClassC and sets ObjectA to reference the instance (**1a**). So, when the code invokes MethodA( ) on ObjectA (**1b**), it is ClassC's implementation that executes.
2. The pseudo-code instantiates ClassB and sets ObjectA to reference the instance (**2a**). So, when the code invokes MethodA( ) on ObjectA (**2b**), it is ClassB's implementation that executes.

Again, as a practical example, the `Draw( )` method defined by different subclasses of a `Shape` super class can just as well be a `Draw( )` method defined by different classes that implement a `Shape` interface. Like method overriding, the same message invokes different behavior, but the class that implements an interface is entirely responsible for implementing that behavior, because it does not inherit any of its implementation from a super class. This kind of polymorphism, using interfaces, can also be used to implement delegation. For more information on delegation, see Delegation on page 20.

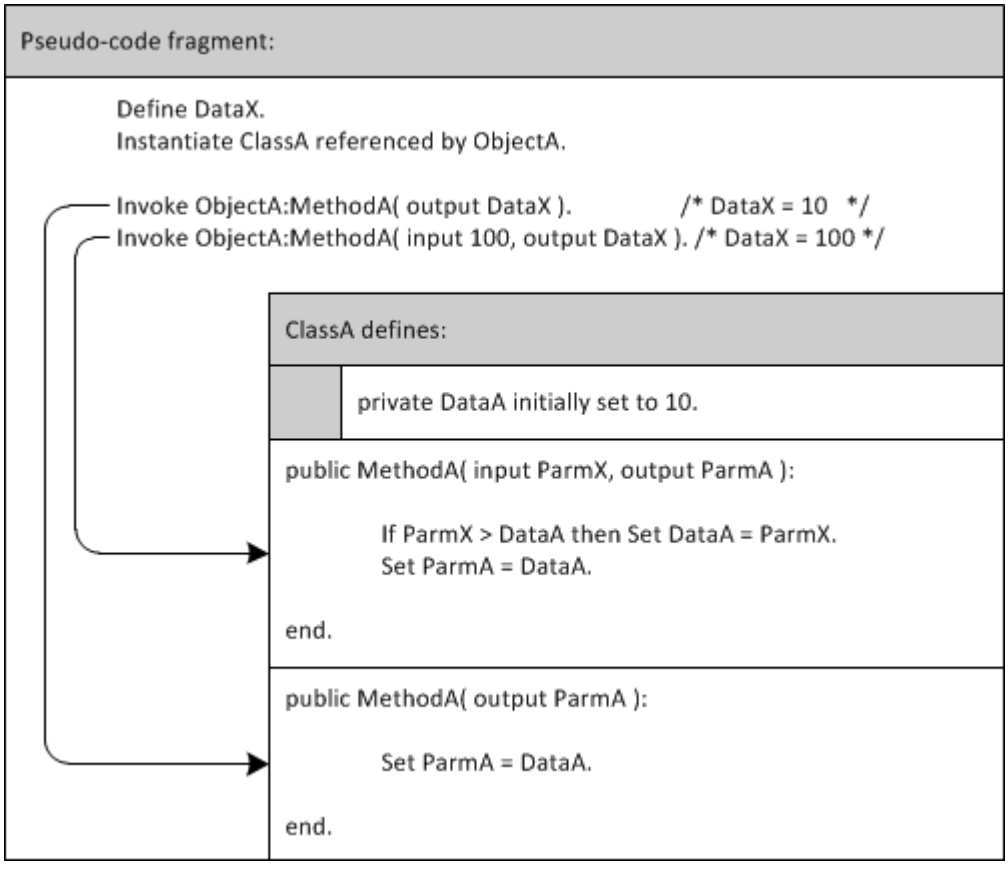
Note that method overloading also provides a less powerful mechanism, similar to (and sometimes referred to as a kind of) polymorphism. With method overloading, an object responds to different messages, each of which invokes a method of the same name, but a different signature, depending on the message. With method overloading, although you are invoking a method of the same name, the behavior depends on a different message for a given object, which you must know at compile time, to invoke the appropriate method. This contrasts with method overriding, where the behavior depends on different objects at run time that all respond to the same message, which you know at compile time, to invoke the appropriate method. For more information on method overloading, see Method overloading on page 25.

Method overloading

Method overloading allows a class to define multiple methods with the same name, but different signatures. That is, it allows you to define different methods that have the same name, but that respond to correspondingly different messages sent to an instance of the class.

The following figure shows an example of overloading where a ClassA defines two overloads of a MethodA( ).

Figure 5: Method overloading example



In this example, the two methods return similar data as an output parameter, but one of them takes an input parameter that allows the method to conditionally change the data before returning it. The only requirement for the overloading to work, different method signatures, is satisfied by the additional input parameter in one of methods.

There is no requirement that overloaded methods have any functional relationship to each other, and they can, in fact, each implement behavior that is totally unrelated to the others. However, method overloading provides a notational convenience that allows you to define similar (but different) methods that provide a related set of behaviors (indicated by the same method name), but where access to a given implementation of the behavior requires a different set of parameters.

For a practical example using a `Shape` super class, a method to calculate the area of a shape really requires a different signature, depending on the shape. So, for example, to calculate the area of a rectangle, you need two values, its length and width, but to calculate the area of a circle, you need only one value, its radius. Therefore, for `Shape`, you might define two overloads of an `Area( )` method, one that takes two numeric parameters (*`parm1`* and *`parm2`*) and one that takes a single numeric parameter (*`parm`*). Then, when calculating the area of a rectangle, you call the `Area(`*`parm1,parm2`*`)` overload of the method with a definition that calculates the area from a length and width. When calculating the area of a circle, you call the `Area(`*`parm`*`)` overload of the method with a definition that calculates the area from a radius.

Note that this example can also rely on the polymorphic overriding of every `Area( )` method overload in each subclass. For example, in order for each subclass to respond when a particular `Area( )` overload does not apply, the super class implementation of these methods can be defined to raise an error message indicating that the method does not apply to the given subclass. Invoking code can simply call each method overload, and respond accordingly.

For more information on polymorphism, see 'Polymorphism' in *Develop Object-oriented ABL Applications*.

Strong typing

Strong typing is the enforcement of type conformance. The object types defined by classes and interfaces are analogous to the built-in primitive types of the language. Object types are identified using an object type name that includes the name of the class or interface. For a user-defined class or interface, this name matches the filename of the file that stores the definition of the class or interface type. The object type name that includes this class or interface name allows ABL to locate the specified class or interface definition at both compile time and run time. Thus, the class file also defines the class or interface as a user-defined object type, which the compiler uses to validate references to the class or interface and its members. For built-in classes and interfaces, ABL validates references against internal class and interface definitions.

Object type validation is similar to data type checking for variables. ABL uses the entire class definition, including all of its data members, properties, methods, and events, as a distinct type which must be perfectly matched by all references to the class. Most object type usage is validated at compile time ensuring that a member cannot be accessed on a class unless a member of the same name with the exact data type or signature is defined by the class or its hierarchy. Thus, the compiler ensures that the program will execute without type errors. Again, this is similar to ABL ensuring that you do not attempt to perform a multiply operation on two character strings, because only numeric data types can be operated on using multiplication.

Comparison with procedure-based programming complie time validation

Strong typing is one of the key differences between procedure-based and class-based programming. When one ABL procedure runs another, ABL is generally unable to tell at compile time whether the `RUN` statement is valid or not. Even when you run an internal procedure in the same procedure, ABL does not verify that the internal procedure exists or that its signature matches. If you run another external procedure, or an internal procedure that is contained in another external procedure, the compiler has no way of checking whether the `RUN` statement is valid because the compiler never examines one procedure while it is compiling another.

When you use classes, because of strong typing, the compiler always verifies the correctness of every statement that references another class, and might even compile the other class to do this, depending on the relationship between the classes. This compile-time validation across classes is one of the most fundamental benefits of using classes. Procedure-based programming gives you greater flexibility in how the pieces of your application fit together, even allowing you to specify the name of a procedure as a variable or expression. But this flexibility comes at the cost of much less confidence at compile time that all the elements of your application will operate and interact correctly when you run it. Thus, class-based programming provides the opportunity to move much more of the verification of an application's correctness from the burden of run-time testing into the compiler.

Glossary of terms

A glossary of terms introduces the object-oriented concepts.

Table 1: Glossary of terms

Term	Meaning
Access mode	Controls where and how a class member can be accessed.
Class	Provides the definition of a type, which represents the state and behavior for instances of the class (objects). A class can define members, including data members and properties (representing the state), and methods and events (representing the behavior) for any object belonging to the class. A class can be abstract, in which case it can define abstract members, including properties, methods, or events, that must be implemented by a derived class. An abstract class member is defined only by a name, data type, and signature as appropriate for each abstract method, property, or event, but no implementation.
Class hierarchy	The set of classes that make up the class definition. The hierarchy includes all classes in the inheritance chain as well as any implemented interfaces.
Delegation	A design technique where one class (referred to as the container) accesses the `PUBLIC` interface of another class (referred to as the delegate). Normally, the container class uses the capabilities of the delegate class by delegating work to it. The container class then defines its own `PUBLIC` interface to allow other classes outside the container class to access the delegate class's capabilities.
Derived class	See Subclass.
Encapsulation	Refers to the design and creation of self-contained and purposed components that are implemented as classes. Because a well-designed class strictly controls access to its state and behavior without revealing its implementation, it supports information hiding (encapsulation) to protect that state and behavior. Thus, encapsulation allows the implementation of an object’s behavior to change without affecting any caller that invokes that behavior in the object.

Term	Meaning
Inheritance	The mechanism through which new classes (or types) can be derived from existing classes or types. The relationship of subclass and super class is established by inheritance. The subclass inherits the `PUBLIC`, `PACKAGE-PROTECTED`, and `PROTECTED` data members, properties, methods, and events from the super class. `PACKAGE-PRIVATE` members are inherited if the super class and subclass are in the same package. The subclass can then provide its own implementation of inherited methods, and it can also add additional data members, properties, methods, and events of its own. Inheritance can thus be viewed as a specialization mechanism. If the super class is abstract, any abstract properties, methods, or events that it defines must also be implemented by a subclass. The hierarchy that is established when one class inherits from another treats the classes in the hierarchy as a single unit. From outside the class, you cannot tell whether the class's state and behavior are defined directly in the class or inherited from a super class. The class that you actually instantiate becomes the bottom (most derived subclass) of its class hierarchy in the running class instance. An interface can inherit member prototypes from one or more other interfaces. The interface that inherits member prototypes is the subinterface and the interface from which the member prototypes are inherited is the super interface. The inheriting interface effectively combines all inherited super interface member prototypes together with its own member prototypes as members of a single interface. Interface inheritance has no effect on class inheritance, but it requires that any class that implements an interface to implement all interfaces that the interface inherits. The member prototypes defined or inherited by an interface that a given class implements can be implemented by any class or classes in the given class inheritance hierarchy.
Interface	Provides the definition of a type that specifies a contract consisting of one or more properties, methods, and events that must be defined by any class that implements the interface. An interface assures a common way for accessing functionality that might differ from one class to the next depending on its implementation. All properties, methods, and events in an interface are abstract, meaning that the interface provides only a name and signature definition (prototype) for each property, method, and event, but no implementation.
Member	Class definition:An element of a class definition that defines its state or behavior. Depending on its definition, a member of a class can be inherited by other classes (subclasses) or accessed from contexts outside the defining class hierarchy. Members of a class can include data members, properties, methods, and events. All other components of a class definition support the definition or function of these class members. Interface definition:An element of an interface definition that defines a prototype for state or behavior that a class must implement. Members of an interface can include prototypes for properties, methods, and events, which the interface can define or inherit from other interfaces (super interfaces). All other components of an interface definition support the member prototypes that it defines.
Message	A communication with an instance of a class (object) that identifies specific behavior to invoke in that class. One way to communicate with (send a message to) an object is to invoke a method on the object that is defined by the object's type.

Term	Meaning
Method overloading	Defining a method in a class that has the same name as, but a different signature than, another method defined in or inherited by the class. It provides a means to conserve method names, where you might specify related but different behavior for each method defined with the same name.
Method overriding	Defining a method in a class that has the same name, signature, and return type of a non-private method defined in its class hierarchy. An overriding method can access the overridden method in a super class in order to extend that super class's behavior. Method overriding is used to implement a powerful type of polymorphism.
Object	For classes, an instance of a class with state (represented by its data members and properties) and behavior (implemented by its methods). For procedures, an instance of a persistent procedure (procedure object). Also, an instance of an ABL data, visual, or other element (such as a procedure or socket object) that can be referenced by a handle (handle-based object), and an instance of an ActiveX control (COM object) as supported in ABL.
Object reference	A value that provides access to an instance of a class. An object reference is strongly typed and can be stored in an ABL data element, such as a data member or property of a class or a variable in a method, procedure, or user-defined function.
Polymorphism	A mechanism where by a method can be implemented or overridden in different ways by different subclasses of a super class. Any reference to the method on a super class object reference, or within the class hierarchy of the object, invokes the most derived version of the method, that is, the version of the method defined in the most derived subclass in the class hierarchy of the object. This ability to provide multiple behaviors for a method by allowing different implementations in different subclasses and, accessing the different subclass implementations through the super class, is called polymorphism—hence *poly-* (many) *morphism* (forms).
Root class	The super class for all classes that do not explicitly inherit from some other class. In ABL, the root class is the built-in class, `Progress.Lang.Object`. For more information, see 'Use the root class Progress.Lang.Object' in *Develop Object-oriented ABL Applications*
Super class	A class that is inherited by another class. A super class is also called a base class. The top-most super class in a given class hierarchy is the *root class* (`Progress.Lang.Object` in ABL).
Subclass	A class that inherits behavior from another class (a super class) in its class hierarchy. A subclass has access to all of the non-private state and behavior in its super class hierarchy. A subclass is also called a *derived class*, and the bottom-most subclass in a given class hierarchy is called the *most derived class* (*or subclass*).
Subinterface	An interface that inherits member prototypes from one or more other super interfaces. Inherited members must be implemented by a class as if they were defined directly by the subinterface that the class implements.

Term	Meaning
Super interface	An interface whose member prototypes are inherited by another interface.
Type	Specifies the structure and semantics of an object, but not its implementation. Types are used to enforce strong typing, and an object type is identified by a name and the interface to its members. The implementation of an object, however, is defined by the behavior of its class members.

Overview of class-based ABL

The following topics introduce ABL support for programming with classes. The syntax and behavior of each ABL element described in these topics is described more completely in other topics, as noted.

Define classes

ABL allows you to define a class as a named block that always begins with the `CLASS` statement and always ends with the `END CLASS` statement. The source code for a `CLASS` statement can appear only in a class definition file (`.cls` file type), and a class definition file can contain only one such `CLASS` statement. This statement defines the class type name that any other class or procedure can use to reference the defined class and that any subclass can use to inherit from this class. (For more information on class definition files, see Class definition files and object type names on page 44.) This statement can also optionally identify the type name of a super class that the defined class inherits from, as well as one or more interfaces specifying member prototypes (properties, methods, or events) that the class must implement (see the). The class can define itself as `FINAL`, which prevents it from being inherited by a subclass. The class can alternatively define itself as `ABSTRACT`, which requires that it be inherited by a subclass. Classes are always public, which means they are always accessible to other classes and procedures.

A class can contain similar kinds of definitions for data elements and executable code as in procedures. However, the data elements and executable code of a class is organized into strongly-typed class members, including data members, properties, methods, and events, that can be inherited from one class by another. If the class is abstract, it can also define any properties, methods, and events as abstract. Abstract class members must be implemented by a subclass that inherits them, similar to the members of an interface that a class implements.

The main block of a class can contain non-executable statements that define:

- Any number of data members, including ProDataSets, data-sources, temp-tables, buffers, queries, and simple variables. (See Define data members on page 32.)
- Any number of properties, which are class members similar to simple variables, but with associated behavior. (See Define properties on page 33.)
- Any number of named methods, which are members that define class behavior. (See Define methods on page 31.)
- Any number of class events, which are members that support behavior based on named methods or internal procedures that can be dynamically configured to execute in response to run-time conditions. (See Define class events on page 33.)
- Any number of optional constructors, which are special methods that define initial behavior for a class. (See Define constructors on page 34.)

- An optional destructor, a special method that defines final behavior for a class. (See Define the destructor on page 35.)
- Any number of static definitions for class-scoped handle-based objects, including widgets, streams, and work-tables that provide associated ABL resources to the class, but are not themselves class members. (See Define class-scoped handle-based objects on page 91.)
- Any number of `ON` statements, which specify ABL triggers for widget and other low-level handle-based object events managed as part of class behavior. (See the `ON` statement reference entry in *ABL Reference*.)
- Any number of user-defined function prototypes (not the function definitions themselves) for user-defined functions called in the class. (See the `FUNCTION` statement reference entry in *ABL Reference*.)

The main block of a class cannot contain any executable statements that are outside of a method, constructor, destructor, or ABL trigger definition.

For information on the ABL to use a class type, see Use object types on page 36. For more information on defining classes, see Define classes on page 49.

Comparison with procedure-based programming file structure

A persistent procedure can contain executable statements in its main block and can define parameters. Classes support constructors to provide the same functionality. The equivalent of data members for persistent procedures are variables and other data elements defined in the main block. The equivalent of methods for persistent procedures are internal procedures and user-defined functions. The closest equivalent of properties for persistent procedures are user-defined functions and internal procedures that encapsulate access to data elements defined in the main body of the persistent procedure. There is no equivalent in a persistent procedure for an abstract member of a class.

Define methods

Within the main block of a class, you can specify executable behavior by defining methods as class members. ABL allows you to define a named method (here on referred to simply as a *method*) as a block that always begins with the `METHOD` statement and always ends with the `END METHOD` statement. A method can be defined with an access mode where `PUBLIC` is the default. Where and how a method can be invoked depends on its access mode:

- **`PRIVATE`** — A private method can be invoked only from within the defining class itself, although an instance can access a private method of another instance if both instances are from the same class.
- **`PACKAGE-PRIVATE`** — A package-private method can be called from within the class and any class within its package.
- **`PROTECTED`** — A protected method can only be invoked from within the class hierarchy.
- **`PACKAGE-PROTECTED`** — A package-protected method can be called from within the class, any class within its package, and from within any subclass that inherits the class.
- **`PUBLIC`** — A public method can be invoked from inside or outside the class hierarchy.

A method can have parameters and a defined return type (similar to a user-defined function), which together represent its signature; its return type can also be `VOID`, which means that it does not return a value. You can also define a method as `FINAL`, which means that it cannot be overridden in a subclass. The statements that you include inside a method block can include most of the statements that you include within the block of an internal procedure or user-defined function. You can override any inherited method not defined as `FINAL` by implementing it with an identical signature and a different method block.

A method override can also specify a less restrictive access mode. It is the method override in the most derived class that provides the method behavior to the entire class hierarchy and its consumers.

Each non-overridden method defined within a class must have a unique identity, but multiple methods in the class can have the same name as long as they are overloaded with unique calling signatures.

Within an abstract class, you can also define a method as ABSTRACT (and not FINAL), which means that it must be overridden and implemented in a subclass. An abstract method is defined only by its signature, as a prototype, without any code block or associated END METHOD statement. The first non-abstract derived class in the subclass hierarchy must override and implement the method as non-abstract (with a code block and terminating END METHOD statement), unless an abstract class has already done so. As with any method override, the non-abstract override of the method in the most derived class provides the method implementation used by the class hierarchy and its consumers, and any method override (abstract or non-abstract) can specify a less restrictive access mode.

ABL also supports static methods that are associated with a class type rather than a class instance as for instance methods. Unless otherwise specified, any reference to a method in this manual refers to an instance method. For more information on static class members, see Use static members of a class on page 186.

For information on the ABL to call methods, see Call class-based methods on page 140. For more information on defining methods, see Define methods within a class on page 71.

Comparison with procedure-based programming internal procedures and user-defined functions

Methods generally combine the characteristics of internal procedures and user-defined functions, and add features unique to classes.

Define data members

Variables and many other data elements that you can define within the main block of a class are known as its data members. The DEFINE statements for data members include support for an access mode. The available access modes differ depending on the type of data member. By default, all data members are PRIVATE.

Note: Data members cannot be abstract.

ABL also supports static data members that are associated with a class type rather than a class instance. Unless otherwise specified, any reference to a data member in this manual refers to an instance data member. For more information on static class members, see Use static members of a class on page 186.

For information on the ABL to access data members, see Access data members and properties on page 153. For more information on defining data members, see Define data members within a class on page 58. You can also define class members known as properties, which combine features of data members and methods in a single class member. For more information, see Define properties on page 33.

Comparison with procedure-based programming data elements

ABL supports data members in classes in a similar way to the data elements that you define in the main block of a persistent procedure. However, data elements in a procedure are always private, unless they are specified as SHARED, in which case they can be shared among different external procedures (persistent or non-persistent). ABL does not allow shared variables to be directly accessed by classes.

Define properties

Properties are class members similar to variable data members, but they can have behavior defined and associated with them. This behavior specifies if a property can be read or written, and includes any statements to be executed when the property is read or written. ABL allows you to define a property using a `DEFINE PROPERTY` statement, which includes the definition of any one or two special methods, each of which is referred to as an *accessor*. A `GET` accessor indicates that the property is readable and includes optional statements to be executed when the property is read. A `SET` accessor indicates that the property is writable and includes optional statements to be executed when the property is written. The data type of a property can be any data type allowed for the return type of a method. Each property has an access mode which can be separately defined for one of the two specified accessors. By default, all properties are `PUBLIC` (unlike data members).

Within an abstract class, you can also define a property as `ABSTRACT`, which means that it must be overridden and implemented in a subclass. An abstract property is defined by its data type and accessor declarations, declared as a prototype without an initial value or any accessor code blocks. The first non-abstract derived class in the subclass hierarchy must override and implement the property as non-abstract (by defining any required initial value and accessor code blocks), unless an abstract class has already done so. An abstract property override can also specify a less restrictive access mode.

Note: Unlike methods, you can only override an inherited property that is defined as abstract.

ABL also supports static properties that are associated with a class type rather than a class instance. Unless otherwise specified, any reference to a property in this manual refers to an instance property. For more information on static class members, see Use static members of a class on page 186.

For information on the ABL to access properties, see Access data members and properties on page 153. For more information on defining properties, see Define properties within a class on page 62. You can also define data elements known as a data members. For more information, see Define data members on page 32.

Comparison with procedure-based programming user-defined functions

Persistent procedures have no direct equivalent to properties of a class. You can simulate the function of properties in a persistent procedure by defining user-defined functions, each of which encapsulates access to a corresponding variable defined in the main block.

Define class events

You can define events as class members that allow you to send (*publish*) notification of any run-time conditions that you detect. Your application can then respond to this notification by executing one or more methods and internal procedures (*event handlers*) that you specify (*subscribe*) for the event. ABL allows you to define a class event with a name and a signature using the `DEFINE EVENT` statement. The signature defines the parameters (if any) that any subscribed event handler must have. An event can be defined with an access mode, where `PUBLIC` is the default. The event access mode determines where in an application you can subscribe event handlers to the event. You define no executable code in an event definition. Event handlers provide all the code that executes for an event.

Within an abstract class, you can also define an event as ABSTRACT, which means that it must be overridden and implemented in a subclass. An abstract event is defined by its signature, declared as a prototype. The first non-abstract derived class in the subclass hierarchy must override and implement the event as non-abstract, unless an abstract class has already done so. However, unlike abstract properties and methods, there is no executable code to implement an abstract class event. So, the overriding event definition used to implement an abstract event is very similar to the abstract prototype originally used to define the abstract event. The result of implementing an abstract event is that the subclass which implements the event is the only class where you can directly publish the event. Any override of an abstract class event can also specify a less restrictive access mode.

Note: Unlike methods, you can only override an inherited event that is defined as abstract.

ABL also supports static events that are associated with a class type rather than a class instance as for instance events. Unless otherwise specified, any reference to a class event in this manual refers to an instance event. For more information on static class members, see Use static members of a class on page 186.

You can publish and subscribe event handlers to class events by calling built-in methods (*event methods*) that ABL supports only on class events. For more information on the event methods for publishing and responding to class events, see Publish and respond to class events on page 39. For more information on defining class events, see Define events within a class on page 77.

Comparison with procedure-based programming named events

Class events provide a similar notification feature for classes that named events provide for procedures. One difference is that you must define class events as strongly-typed members of a class before you can publish them. However, you define named events at run time when you publish them.

Define constructors

A class can define a special method called a constructor, which allows you to execute behavior and initialize data members and properties during the instantiation of a class. ABL allows you to define a constructor as a named block that always begins with the CONSTRUCTOR statement and always ends with the END CONSTRUCTOR statement. A constructor must have the same name as the name of the class in which it is defined and it can only execute when the class is instantiated. A constructor can be defined with any access mode. PUBLIC is the default.

Where and how a constructor can be invoked depends on its access mode:

- **PRIVATE** — Can be invoked from another constructor of the class in which it is defined, and an instance can directly invoke a private constructor of its own class.
- **PACKAGE-PRIVATE** — Can be called from the defining class or any class within its package. This is done via the SUPER statement within another constructor or using the NEW statement or function elsewhere.
- **PROTECTED** — Can be invoked by another constructor of the class or from a constructor of a subclass. An instance can invoke a protected constructor as long as the constructor is defined in the same class as the instance or a super class of the instance. If all constructors of the class are defined as PROTECTED, this class can only be instantiated indirectly when you directly instantiate a subclass of this class.
- **PACKAGE-PROTECTED** —Can be called from the defining class, from a subclass, or from any class within its package. This is done via the SUPER statement within another constructor or using the NEW statement or function elsewhere.
- **PUBLIC** — Can be invoked from inside or outside the class hierarchy.

A constructor can also have parameters, but no defined return type. You can define more than one constructor (overload constructors) in a class as long as the calling signature of each constructor is unique.

ABL also supports a static constructor that initializes static data for a class type instead of instance data (for a class instance) as with instance constructors. Unless otherwise specified, any reference to a constructor in this manual refers to an instance constructor. For more information on static constructors, see Use static members of a class on page 186.

For information on the ABL to instantiate a class, see Create and destroy a class instance on page 37. For more information on defining constructors, see Define class constructors on page 84.

Comparison with procedure-based programming main block

For persistent procedures, the equivalent behavior executes directly in the main block, and there are no restrictions on where and when a particular persistent procedure can be instantiated.

Define the destructor

A class can define a special method called its destructor, which always executes when an instance of the class is deleted (destroyed). ABL allows you to define one destructor as a named block that always begins with the `DESTRUCTOR` statement and always ends with the `END DESTRUCTOR` statement. This destructor must have the same name as the name of the class in which it is defined. The destructor is always `PUBLIC`, takes no parameters, and has no return type.

For information on the ABL to destroy a class, see Create and destroy a class instance on page 37. For more information on defining destructors, see Define the class destructor on page 88.

Comparison with procedure-based persistent procedures

Persistent procedures have no direct equivalent for a destructor. However, you can create a similar mechanism using triggers that respond to the equivalent of clean-up events.

Define interfaces

An interface declares property, method, and event prototypes and certain related data definitions that must be implemented by a class that specifies the interface as part of its own class definition. ABL allows you to define an interface as a named block that always begins with the `INTERFACE` statement and always ends with the `END INTERFACE` statement. The source code for an `INTERFACE` statement appears only in a class definition file (`.cls` file type), and a class definition file can contain only one such `INTERFACE` statement. This statement defines the interface type name that other classes can use to identify this interface as one they intend to implement. (For more information on class definition files, see Class definition files and object type names on page 44.)

Note: A class definition file can define a single class or a single interface, but not both, in the same file.

The INTERFACE statement differs from the CLASS statement in that it does not support the inheritance of classes or the implementation of other interfaces, and by definition, it cannot be specified as FINAL. However, it does allow you to specify one or more other interface types (super interfaces) from which this interface inherits member prototypes. In this way, you can define an interface from a combination of members that it inherits and members that it defines. Properties, methods, and events are the only members you can define for an interface. These property, method, and event definitions are limited to defining prototypes that contain no executable code. Members of an interface are always PUBLIC. The only data that an interface can define are temp-tables and ProDataSets used to define parameters for the methods that it defines.

For information on using an interface type in ABL, see Use object types on page 36. For more information on defining interfaces, see Define interfaces on page 93.

Comparison with procedure-based programming persistent procedures

Persistent procedures have no equivalent for an interface.

Use object types

In order to support strong typing of references to objects, a class or interface type can be used to specify a data type in a variable definition. The variable can then be used to hold an object reference, which is a value that references an instance of a class and its members. The object type name that defines the object reference can identify a class or an interface. If it identifies an interface, the object reference is used to point to an instance of a class that implements the specified interface. This interface can be an interface type that the class instance explicitly implements or a super interface inherited by the explicitly implemented interface.

Note: An object reference never points to an instance of an interface, because an interface is not an object itself, but is only used for defining a class and for referencing an instance of a class that implements the interface definition.

The object type name for a class or interface consists of two parts—a class or interface name preceded by a qualifying package name, for example, Classes.Inventory.MyClass. The package name corresponds to a directory path (related to PROPATH) where the class file that defines the class or interface type is stored, and the class or interface name is identical to the class filename. In a supported context, you can reference an object type without its qualifying package name. For more information, see Specify unqualified class or interface type names on page 37. For more information on class or interface type names, see Define and reference object type names on page 45.

An object reference is comparable to the HANDLE data type, with the important distinction that an object reference identifies a specific object type that a variable or field defined for that object reference can hold. This allows the compiler to verify that all uses of the object reference are correct, which is not possible with a weakly typed handle.

In addition to a variable, you can define the following ABL elements to hold an object reference to a class or interface data type:

- A property or the return type of a method or user-defined function
- A parameter to a method, local procedure, or user-defined function
- A field in a temp-table, which must always be defined as the root class type, Progress.Lang.Object

Note: You cannot define a database field as an object reference.

You can pass an object reference as a parameter between an application server and an ABL client, as long as the object's class satisfies certain restrictions and the class is defined on both the client and server sides. See Pass object reference parameters on page 159 for more information.

You never use an object reference to access static members of a class. Instead, you can use a class type name to access the static members of a class. For more information, see Access static members on page 186.

For more information on defining and using object references to class and interface types, see Instantiate and manage class-based objects on page 134.

Specify unqualified class or interface type names

Any procedure or class source file can begin with one or more `USING` statements, each of which identifies a fully qualified class or interface type name or the package name for one or more classes or interfaces that the file references. Each `USING` statement specifies one or more classes or interfaces defined in a given package whose object type names you can reference using their unqualified class or interface names (without specifying the package). For more information, see Reference an object type name without its package on page 48.

Create and destroy a class instance

The first requirement to program with classes is to instantiate a class (create a class-based object). In ABL, you must use the `NEW` function (classes) to create an instance of a class. You can also assign the object reference for that instance to an appropriate object reference data element using the `NEW` statement, which calls the `NEW` function. This function invokes the specified class constructor to complete class instantiation. To help support OERA-compliant applications, you can also use the `DYNAMIC-NEW` statement, which allows you to instantiate a class from an object type determined at run time. You can also instantiate a class from criteria identified at run time using the `New( )` method of the `Progress.Lang.Class` class.

The first class in an ABL application must be instantiated in a procedure file (not a class definition file). Thus, you must use an initial procedure file to start up a class-based application.

Objects (class instances) are automatically deleted (garbage collected) by the AVM some time after no reference to the object exists in the ABL session. However, you can force any class instance to be deleted immediately using the `DELETE OBJECT` statement. As with garbage collection, this statement invokes any destructor specified for the class instance. If you or the AVM create a class instance that is never assigned to an ABL data element (for example, when `NEW` appears in an expression without assignment), you do not need to delete the object; the AVM cleans it up automatically. Note that, once initialized, static members of a class cannot be re-created or destroyed.

For more information on managing the creation and destruction of class instances, see Garbage collection on page 98. For more information on how the class hierarchy of an object is created and destroyed, see Design objects: inheritance, polymorphism, and delegation on page 101. For more information on using the `NEW` function and `DYNAMIC-NEW` statement to instantiate a class, see Create a class instance on page 136. For more information on initializing static class members, see Initialize and delete static members on page 192.

Comparison with procedure-based programming persistent procedures

Using the VAR or NEW statement to instantiate a class and assign its object reference is roughly equivalent to running a persistent procedure and setting its procedure object handle using the RUN statement. Just as a procedure-based ABL application that instantiates persistent procedures must begin with a startup procedure file that directly or indirectly creates procedure objects through other procedure files, a class-based ABL application must also begin with a startup procedure file that directly or indirectly creates class-based objects. (You cannot directly start up an application with ABL using a class file as you can, for example, with Java.) When it is no longer needed, you can use the DELETE OBJECT statement to destroy a persistent procedure. Class instances are automatically deleted (garbage collected) when a reference to the object no longer exists in the ABL session.

Invoke methods

You can invoke a method within a class that defines or inherits it simply by naming it and, if it returns as value, by including it in an expression, much as with a user-defined function, for example, `GetItem( )`. You can invoke a PUBLIC method defined in another object (outside the running class hierarchy) in a similar fashion, except that you must prefix the method name with a reference to the other object separated by a colon (`:`), for example, `rInventory:GetItem( )`. Invoking a method *on* (defined in) another object is analogous to invoking a built-in method on an ABL handle-based object, for example, `ERROR-STATUS:GET-MESSAGE( )`. To help support OERA-compliant applications, you can also use the DYNAMIC-INVOKE function, which allows you to invoke a method whose name is determined at run time. You can also invoke a method from criteria identified at run time using the `Invoke( )` method of the `Progress.Lang.Class` class.

For PACKAGE-PROTECTED or PACKAGE-PRIVATE methods outside of the hierarchy but within the same package, you must use the *`object:method()`* syntax.

Outside of the class hierarchy where it is defined, you must invoke a PUBLIC *static* method by prefixing the method name with the class type rather than an object reference. This is also the case for PACKAGE-PROTECTED or PACKAGE-PRIVATE *static* method outside of the hierarchy but within the same package. For more information on static methods, see Use static members of a class on page 186.

For more information on invoking methods, see Call class-based methods on page 140.

Comparison with procedure-based programming invoking internal procedures

When you invoke an internal procedure from within an external procedure where it is defined, you use a RUN statement that simply names the procedure. When you invoke a user-defined function within an external procedure where it is defined, you name the function in an expression, similar to invoking a method within a class. However, you have to forward reference the definition for the user-defined function if it occurs after the point of invocation.

When you invoke an internal procedure defined in another external procedure, you use the RUN statement with an IN option to specify the location of the internal procedure definition. When you invoke a user-defined function defined in another external procedure, you must specify the prototype and reference the location of the function definition, then invoke the function by naming it in an expression. Method invocation within its defining class or on an another object instance is far more consistent than for internal procedures and user-defined functions.

Access data members and properties

You can access an accessible data member or property from within a class that defines or inherits it simply by naming it, for example, `ItemCount`. A data member can appear wherever its data type is allowed. A property can appear wherever its data type and accessor definitions allow. If the property is readable, it can appear wherever an expression of the specified data type is allowed. If the property is writable, it can appear wherever the specified data type can be written.

You can access a `PUBLIC` data member or property that is defined in another class instance (outside the running class hierarchy) in a similar fashion, except that you must prefix the data member or property name with an object reference to the other class instance separated by a colon (`:`), for example, `rInventory:ItemCount`. Accessing a data member or property *on* (defined in) another object is analogous to accessing a built-in attribute on an ABL handle-based object, for example, `ERROR-STATUS:NUM-MESSAGES`.

You can also access a class property dynamically using the DYNAMIC-PROPERTY function. This is initiated with the help of GetPropertyValue( ) and SetPropertyValue( ) methods that make it possible to get or set a property's value without knowing the type of the property's name at compile time,

For more information on accessing a class property dynamically, see Dynamically access a property at runtime on page 156.

Outside of the class hierarchy where it is defined, you must access a `PUBLIC` static data member or property by prefixing the member name with the class type rather than an object reference. For more information on static data members and properties, see Use static members of a class on page 186.

For more information on accessing data members and properties, see Access data members and properties on page 153.

Comparison with procedure-based programming shared variables

You access variables and other ABL data elements defined in procedures the same way no matter where and how they are defined—simply by referencing their names or handles (as appropriate). So, for example, a variable defined as `NEW GLOBAL SHARED` in one procedure, once defined as `SHARED` in a second procedure, can be accessed by name as if it were newly defined in the second procedure. No such mechanism exists or is required for data members or properties.

Also, where you define variables in a procedure as `SHARED` in order to access them from outside the procedure, you define data members or properties of a class as `PUBLIC` in order to access them from outside the class.

Publish and respond to class events

You can publish a class event using the built-in `Publish( )` event method from within the class that defines the event (see the). The event definition cannot be abstract. When you invoke this method for a given class event, it executes any and all event handlers that you previously subscribed to the event. You also pass any parameters to this method that you defined for the event, and they get passed to each event handler that executes. The method returns any results to the caller.

Caution: Parameter passing for multiple event handlers requires careful design, as results are returned only from the last event handler to execute, and some parameter results can be passed from one event handler to the next. For more information, see Publish class events on page 170.

You can subscribe an event handler to an event using the built-in `Subscribe( )` event method. An event to which you subscribe an event handler can be defined as abstract. The subscription then applies to the class in the hierarchy where the event is actually implemented. The access mode of an event determines where you can execute the `Subscribe( )` method on the event, whether inside or outside the class or class hierarchy where the event is defined. In any case, you can subscribe any accessible method or internal procedure with a compatible signature as a handler for the event, even if the method or internal procedure is not accessible to the publishing class. Using the built-in `Unsubscribe( )` event method, you can also remove a method or internal procedure as an event handler for the event, again, depending on the access mode of the event.

For more information on publishing and responding to class events, see Publish and subscribe to class events on page 167.

Comparison with procedure-based programming PUBLISH and SUBSCRIBE

You publish named events for procedures using the `PUBLISH` statement; you subscribe internal procedures as event handlers for named events using the `SUBSCRIBE` statement; and you remove internal procedures as event handlers for named events using the `UNSUBSCRIBE` statement.

Note: Unlike with class events, operations on named events can only be verified at run time, and named events can only be defined at run time when they are published.

Supporting ABL

In addition to ABL for defining and using class-based objects, the following ABL elements support programming with class-based objects:

- `SUPER` **statement** — Within a class constructor, this statement invokes the constructor with the matching signature in the immediate super class. For more information, see see Construct an object on page 116.
- `SUPER` **system reference** — Invokes the specified method as implemented or inherited by the immediate super class. Analogous to the `RUN SUPER` statement or `SUPER` function in procedure-based programming. For more information, see see Construct an object on page 116.
- `CAST` **or** `DYNAMIC-CAST` **function** — Allows a class object reference to be assigned to a data element defined as a subclass or interface object reference. For more information, see Object reference assignment and casting on page 179.
- `TYPE-OF` **function** — Verifies the type of an object. For more information, see TYPE-OF function on page 174.
- `VALID-OBJECT` **function** — Validates that an object reference points to a real object. Analogous to the `VALID-HANDLE` function for internal object handles. For more information, see VALID-OBJECT function on page 172.
- `THIS-OBJECT` **statement** — Invokes a constructor of the defining class (specified by its signature) from another constructor of the same class. For more information, see see Construct an object on page 116.
- `THIS-OBJECT` **system reference** — Returns an object reference to the current object from a method within the class hierarchy of the object. Also used to call a method from within the class hierarchy where it is defined when the method name is identical to an ABL keyword. Analogous to the `THIS-PROCEDURE` system handle for procedures. For more information, see THIS-OBJECT system reference on page 120.
- `FIRST-OBJECT` **attribute** — A `SESSION` system handle attribute that returns an object reference to the first class-based object instantiated in an ABL session. Analogous to the `FIRST-PROCEDURE` attribute for persistent procedures. You can use the `NEXT-OBJECT` property of any class to walk the list of instantiated

class-based objects forward from the first to the most recent object currently instantiated in the session. All ABL classes inherit NEXT-OBJECT from the root class, Progress.Lang.Object. For more information on the FIRST-OBJECT attribute and the NEXT-OBJECT property, see ABL session object reference attributes on page 177.

- LAST-OBJECT **attribute** — A SESSION system handle attribute that returns an object reference to the most recent class-based object instantiated in an ABL session. Analogous to the LAST-PROCEDURE attribute for persistent procedures. You can use the PREV-OBJECT property of any class to walk the list of instantiated class-based objects backward from the most recent to the first object currently instantiated in the session. All ABL classes inherit PREV-OBJECT from the root class, Progress.Lang.Object. For more information on the LAST-OBJECT attribute and the PREV-OBJECT property, see ABL session object reference attributes on page 177.

- UNDO, THROW **option** — Available on several ABL constructs, this option provides a means to raise ERROR by throwing an error object that can be handled in a variety of ways, depending on the context. An *error object* can be an instance of several built-in ABL classes that implement the Progress.Lang.Object interface, including an application error class that you can extend according to your application requirements. For more information on error handling with classes, see Raise and handle error conditions on page 197.

- RETURN **statement and options** — From a method, returns a value or returns ERROR, depending on the method definition and RETURN options:

  - Without the ERROR option, sets the return value of a non-VOID method according to its defined data type

  - With the ERROR option, returns ERROR and optionally returns an error string (read by the RETURN-VALUE function in the caller) or returns a specified error object. If the method is a constructor, this also cancels and reverses any class instantiation. For more information, see Define data members within a class on page 58 or Define class constructors on page 84.

  The RETURN also works the same way as an option of various ON phrases (such as the ON ERROR phrase) and on the UNDO statement. For more information on error handling with classes, see Raise and handle error conditions on page 197.

- ROUTINE-LEVEL ON ERROR UNDO, THROW **statement** — Changes the default error handling for procedures, user-defined functions, methods, and constructors so that any error object that has been thrown to the routine block is passed up to the caller of the routine. For more information on error handling with classes, see Raise and handle error conditions on page 197.

- CATCH **statement** — Catches and allows access to a specified type of error object that has been raised within the same ABL block. For more information on error handling with classes, see Raise and handle error conditions on page 197.

General comparison with procedure-based programming

There is no expectation that you will convert existing procedural ABL applications to use classes unless you have a reason to do so. The language statements supporting classes provide options for object-oriented programming, and you can choose to take advantage of these options when they are beneficial to you. This section summarizes and compares a few key differences between using procedures and using classes in your development.

When multiple procedures need to access the data in a temp-table or a ProDataSet, the temp-table or the ProDataSet and its temp-tables are routinely defined in an include file and included in all procedures that need access to the temp-table or ProDataSet. In this way the definitions only need to be written once and you have assurance that the definitions are consistent between all the procedures that use them, and that a change to the definitions is propagated to all procedures that use them simply by recompiling them. The temp-table or ProDataSet can then be passed between the procedures as a parameter. Other documentation describes how to pass both ProDataSets and temp-tables by reference, so that they are not copied from one procedure to another (see *Develop ABL Applications* and *Use ProDataSets*).

Within a class hierarchy a super class can define a `PROTECTED` temp-table or ProDataSet whose definition is implicitly shared among all the subclasses of that super class. There is no need to repeat the definition in each subclass. (Indeed, it would be a compiler error to repeat the definition). If the methods that access the common data held in a temp-table or ProDataSet are all in the same class hierarchy, then they can all access the data through its one `PROTECTED` definition, without the need to pass the temp-table or ProDataSet as a parameter.

Similarly, variables and other data definitions that might be defined as `SHARED` in procedure-based ABL applications can be defined as `PROTECTED` within a class hierarchy. However, note that classes cannot define or access the `SHARED` variables used by procedures.

Also, you cannot specify a class file (either source code or r-code) as a startup routine using the Startup Procedure (`-p`) startup parameter. You can only start up an application (whether procedure-base or class-based) using a procedure file.

Programming conventions for classes

Progress recommends the following programming conventions for classes, which are demonstrated in this manual:

- Class names and interface names should begin with an upper case letter and each separate word in the name should start with an upper case letter. This is known as camel case, for example, `MyCustomerClass`.
- Classes and interfaces should be put into packages to uniquely identify their type. Using your company name as part of the package name might help avoid conflicts with similarly named types from other companies that might be on your `PROPATH`. For example built-in classes provided by ABL are contained in the package, `Progress.Lang`.
- References to static class members should be qualified with the class type name.
- You can use ABL reserved keywords to define class, interface, method, property, data member, or class event names. However, Progress Software Corporation recommends that you do not use keywords to define these ABL elements. If you do use keywords to define method, variable data member, property or class event names, you must use additional syntax to access such methods, variable data members, properties, and events within the class hierarchy where they are defined. For more information, see Use the CLASS construct on page 54 and the sections on accessing class members in Program with class-based objects on page 133.
- When defining a class member, its access mode, although optional, should be specified.
- Calls to a method should include the parameter mode, e.g. (`INPUT` cName).

2

Get started with classes, interfaces, and objects

The basic unit of executable code for object-oriented programming is the object, which encapsulates a specific set of state and behavior according to its type. The primary mechanism for defining the type of an object is the class, which specifies the class members for an object and its relationship to other classes. Typically a single object is defined by a hierarchy of classes. Thus, referencing an object by a given class type in its class hierarchy allows you to access all of the public class members defined by that class type and the super classes above it in the class hierarchy.

You can also define interfaces for objects. An interface represents a type that declares method, property, and event prototypes that any class implementing the interface must define. Thus, an interface type allows multiple classes to implement behavior according to the shared contract specified by the interface. You can then reference any instance of these classes as an interface type to access the class-specific implementation of a given interface-declared property, method, or event.

You can manage the implementation of an application using the class hierarchies and interfaces that define your objects.

ABL also supports enumerations, or enums. An enum is made up of a list of strongly typed, named constants called members. The value of a variable defined as an enum is restricted to the list of members defined for that enum. Enums are implicitly class based, so you can use them in many of the same ways you use a class type or instance. For more information, see "ENUM statement" in *ABL Reference*.

For details, see the following topics:

- Class definition files and object type names
- Define classes
- Use the CLASS construct

- Define interfaces
- Use the INTERFACE construct
- Garbage collection

Class definition files and object type names

All class and interface definitions reside in a class definition file, and each class definition file can contain only one class or interface definition. The source (definition) file for a class (distinct from an external procedure) has the `.cls` filename extension. However, a compiled class file contains r-code, and, like a compiled procedure file, it has the `.r` filename extension. So as with `.w` (AppBuilder-coded) and `.p` (hand-coded) procedure files, you cannot have class and procedure files with the same name whose r-code resides in the same location.

As with procedures, you can use include files to extend the definition of a class beyond a single source code file. Classes are always publicly scoped and therefore are available to all other classes and procedures within an application. Classes are not accessible across an application server boundary. For more information on a comparison of class and procedure files, see Compare class definition files and procedure source files on page 48.

Each class or interface definition represents a user-defined object type that you identify using a class or interface type name. This object type name derives from both the class or interface definition and the location of the class file where it is defined. For more information on class and interface type names, see Define and reference object type names on page 45.

Class definition file structure

The structure for coding the source file for a class or interface definition has the following general syntax.

Syntax

```
[ ROUTINE-LEVEL-statement ] [ [ USING-statement ] . . . ]
{ CLASS-statement | INTERFACE-statement }
    class-or-interface-definition
END-statement
```

Element descriptions for this syntax diagram follow:

ROUTINE-LEVEL-statement

A `ROUTINE-LEVEL ON ERROR UNDO, THROW` statement. This statement changes the default error handling for procedures, user-defined functions, methods, and constructors so that any error object that has been thrown to the routine block is passed up to the caller of the routine. This statement can appear before or after any *USING-statement*. For more information on this statement, see Raise and handle error conditions on page 197.

USING-statement

A statement that allows abbreviated references to other class or interface types within this class or interface definition, without having to specify the package in the class or interface type reference. You can specify multiple `USING` statements to make abbreviated references to class or interface types defined in different packages. This statement can appear before or after any

ROUTINE-LEVEL-statement. For more information on this statement, see Reference an object type name without its package on page 48.

CLASS-statement

A statement that begins a class definition by specifying its class type name and its relationship to implemented interfaces and other classes in its class hierarchy. For more information, see Define classes on page 30.

INTERFACE-statement

A statement that begins an interface definition by specifying its interface type name. An interface is defined independently of and unrelated to any class, but it can inherit members from one or more other interfaces. For more information, see Define interfaces on page 35.

class-or-interface-definition

Statements that are part of the class or interface definition. For user-defined classes, these statements define class members and other elements of the class definition. For user-defined interfaces, these statements define method, property, and event prototypes (and related data definitions) for methods, properties, and events that can later be implemented by user-defined classes. For classes, see Define classes on page 30. For interfaces, see Define interfaces on page 35.

END-statement

The statement that terminates a class or interface definition. For classes, see Define classes on page 30. For interfaces, see Define interfaces on page 35.

Define and reference object type names

A class file, together with its filename and location, defines the type of a given user-defined class or interface. You can refer to this class or interface type using an object type name that you define using either a CLASS statement or an INTERFACE statement, depending on the object type. You specify an object type name for both class and interface types using this syntax.

Syntax

```
[ " ][package.]class-or-interface-name [ " ]
```

Element descriptions for this syntax follow:

package

A period-separated list of names that, along with *class-or-interface-name*, uniquely identifies the class or interface among all accessible classes and interfaces in your application environment. These names correspond to the directory names in a valid path name that is relative to PROPATH, where each name is identical to a directory name in the path, and each forward (/) or backward (\) slash separator in the path is replaced with a period (.). The relative path name represented by *package* specifies the location of the class file that contains the class or interface definition. Any *package* specification must remain constant between compile time and run time, and must be complete (cannot correspond to a partial path name above the directory containing the class file). An object type name without a *package* refers to a class or interface whose definition can be found

directly on PROPATH. If *package* contains embedded spaces, you must enclose the entire object type name specification, including *class-or-interface-name*, in quotes.

Note: Do not place a class file in a directory whose name contains a period (.) character. ABL interprets the component after the period as another directory level and therefore will not find the referenced class file.

---

class-or-interface-name

The name of the class or interface. This name must exactly match the filename of the class file (excluding its .cls or .r extension) that contains the definition for the specified class or interface. This class file must be located in the relative path represented by *package* (if specified). The *class-or-interface-name* must begin with an alphabetic character and it cannot contain any periods or spaces. This name thus represents the unqualified name of a defined class or interface.

Once you define a class or interface type, you can reference it by using its fully qualified object type name (including its *package*) or, when an appropriate USING statement is specified, by using its unqualified class or interface name (without its *package*). For more information on the USING statement, see Reference an object type name without its package on page 48.

Note that both elements of an object type name (*package* and *class-or-interface-name*) must conform to the case sensitivity requirements of the operating system (e.g., UNIX or Windows). On a case-sensitive OS, only the first reference to the object type name must be case correct. ABL follows this initial letter case for all subsequent references to the type.

For example, the first reference on UNIX to a class with the type name, topdir.subdir.SomeClass, must be found at compile time in the source file, SomeClass.cls, in the directory, topdir/subdir, that is relative to PROPATH. If the class is compiled and saved, its r-code must also be found at run time in the file, SomeClass.r in the same directory path, topdir/subdir relative to PROPATH. If you later reference an object type name with the letter case specified by Topdir.Subdir.SomeClass, ABL considers this type name to reference the same object type that you initially referenced as topdir.subdir.SomeClass and matches it accordingly.

Note: The requirement to maintain a constant relative path between compile time and run time applies only to class files (**not** to procedure files).

For more information on object type names, see the Type-name syntax reference entry in *ABL Reference*.

Valid and invalid object type name references

Any object type name that you define or reference must evaluate at compile time to the name and location of a class file. ABL syntax includes several elements where object type names are specified, most of which are described in this book. Although you can specify an object type name with or without surrounding quotes, you cannot use a character variable or character expression to specify the object type, except in specific ABL elements that are verified only at run time.

Assuming appropriate package configurations on PROPATH, these are examples of valid object type names that you can specify anywhere an object type name can appear:

```
acme.myObjs.CustObj
"Program Files.myObjs.CustObj"
{&myClassName}
```

In this previous example, the reference to {&myClassName} is a preprocessor directive that is resolved prior to compilation.

These are examples of invalid object type names, where myLocalVariable is a character variable:

```
myLocalVariable
"Program Files" + "." + "myObjs.CustObj"
```

In this previous example, the quotes identify elements of a character expression, not an object type name.

These examples of invalid type names illustrate one of the key aspects of using classes, namely strong typing. In the first example of an invalid object type name, if a character variable, like myLocalVariable, is allowed to specify an object type name, the compiler cannot know the run-time value of myLocalVariable, and thus cannot verify the physical location of the class file or any references to the specified class or interface. Similarly, as shown in the second example of an invalid object type name, because character literals that are concatenated using the "+" operator are only evaluated at run time, the compiler cannot assemble an object type name from a character expression.

Note: Very few ABL elements, such as the DYNAMIC-NEW statement, take a character expression to specify an object type name. In these few cases, such object type names are verified only at run time.

However, you can combine a preprocessor name with other parts of an object type name, because this combination is evaluated and combined into a single name element at compile time. For example:

```
{&AccountingPackage}myClassName
```

Use object type names to locate type definitions in PROPATH

The examples in the following table show several object type names and where the corresponding class files must be located for the specified type names to be valid.

Table 2: Object type names and PROPATH

The file that defines the following object type name . . .	Must be found in . . .
Customer	A directory in PROPATH
accounting.Customer	The accounting subdirectory relative to PROPATH
"Program Files.MyApp.Objects.Customer"	The "Program Files\MyApp\Objects" subdirectory relative to PROPATH

These files must be able to be located on PROPATH at both compile time and run time. However, the value of PROPATH can change between compile time and run time.

Reference an object type name without its package

In both procedure and class definition files where you refer to class or interface types, you can specify one or more USING statements that allow you to reference object types defined in packages using their unqualified class or interface names. Any and all USING statements must appear at the beginning of the source file before all other compilable statements, and you can specify each statement using this syntax:

```
USING { object-type-name | package.* } .
```

You can specify *object-type-name* as the fully qualified object type name for a single class or interface that you want to reference, or you can specify *package* as the package defined for multiple classes and interfaces that you want to reference. Note that the asterisk (*) wildcard cannot match a partial *package*, but only the names of class or interface types that are defined in the specified *package*. You can also only have one wildcard that terminates the specified *package*, as shown. For more information on object type names and packages, see Define and reference object type names on page 45.

Thus, to access a class with the type name acme.myObjs.Common.CommonObj using only its unqualified class name, you can define either of the following USING statements:

```
USING acme.myObjs.Common.*.
USING acme.myObjs.Common.CommonObj.
```

The first statement (a package USING statement) allows you to reference all classes and interfaces defined in the acme.myObjs.Common package using only their class or interface names. The second statement (a type-name USING statement) allows you to access the single specified type defined in the same acme.myObjs.Common package using only its class name (CommonObj). A type-name USING statement is useful when you want to limit class or interface name references to a specified class or interface within a package.

You can include multiple USING statements in a source file to allow unqualified name access to class and interface types defined in multiple packages. If you reference a class or interface type that is defined with the same name in multiple packages, ABL uses the first class or interface type name match that it encounters in order of the specified USING statements, starting with all type-name USING statements (type-name priority), and following with all package USING statements, until an appropriate match is found.

Compare class definition files and procedure source files

A class definition file has many similarities to a procedure file but there are some differences that justify their having different source filename extensions. Both class definition files and procedure files define a main block in which state and behavior are defined. Procedure files have no rigid syntax, and can begin with any ABL statement. However, the main block of a class definition file must be defined either between a starting CLASS statement and a terminating END CLASS statement (the CLASS block) or between a starting INTERFACE statement and a terminating END INTERFACE statement (the INTERFACE block).

The CLASS or INTERFACE statement is necessary to specify the object type name and other information required to support the class or interface definition. If a class definition file (with the .cls extension) does not contain a CLASS or INTERFACE statement, ABL generates a compilation error. Both class definition files and procedure files can have USING statements that allow you to specify unqualified class and interface names for the object types that they reference.

Procedure files allow data elements scoped to the procedure file to be defined in the main block using DEFINE statements outside of internal procedure and user-defined function definitions. Similarly, class definition files allow data members and properties scoped to the class file to be defined using DEFINE statements in the CLASS block outside of method definitions. However, unlike procedure files where the main block is, itself executable, the CLASS or INTERFACE block in class definition files is not executable. A CLASS block does not allow any executable code to appear outside a method definition or any other separately executable block that you can define within the CLASS block. An INTERFACE block cannot contain any executable code.

Therefore, unlike the executable code that can run in the main block when you instantiate a procedure object, there is no code that can run in the main block of a class when you instantiate a class-based object, except in a constructor. To execute code during class instantiation, you must add the code to a constructor, which is a special method designed to execute for a class when it is instantiated. The main block of a procedure file can also define procedure parameters to pass when instantiating the procedure object. For a class definition file, you instead define a constructor with parameters that you pass when instantiating the class. Class files, in fact, allow you to define multiple constructors for a class definition, each of which takes a different parameter list. Thus, unlike a procedure object, which can be instantiated with only one set of procedure parameters, you can define a class that can be instantiated with different sets of parameters, depending on the constructor that you use to instantiate the class.

Finally, you cannot start up an application using a class file, as specified using the startup procedure (-p) startup parameter. Even if your application is otherwise built entirely using classes, you must provide a startup procedure to at least instantiate and start up the main-line class for your application.

Define classes

Classes encapsulate state and behavior. Defining a class requires a mechanism to define the set of data members and properties that represent the class's state and a set of related methods and events that provide the behavior of the class. The members of a class comprise the type of the object defined by the class. As with other data types, the compiler verifies that references to a class and its members are consistent with its type. Another object type is an interface, which also has members, and provides a means to enforce a consistent definition for the corresponding members of similar class types. (For more information on defining interface types, see Define interfaces on page 93.) Class-based objects are running instances of classes. At run time there can be many instances of a class, each with its own state. There can also be object references of different types pointing to a single class instance that represents all these types. A valid reference type can be a class in its class hierarchy or an interface that it implements. For more information on how an object can represent more than one type, see Design objects: inheritance, polymorphism, and delegation on page 101.

After beginning a class definition with a CLASS statement, which specifies the class type and its relationship to any other class and interface types, you can define a user-defined class with the following types of elements:

- Data members and property members that define the state of the class
- Method and event members that define the behavior of the class, including special methods (accessors) associated with properties
- Optional constructors, which are special methods that are invoked when the class is instantiated
- One optional destructor, which is a special method that is invoked only when the object is destroyed
- Class-scoped handle-based objects that are not class members, but which can be referenced in the class, including static ABL widgets, streams, and work tables
- ON statements that are also not class members, which are used to define triggers for events on ABL widgets and other supported handle-based objects that are referenced in the class
- User-defined function prototypes, also not class members, which are used to specify user-defined functions in external procedures that are referenced in the class

Together with the CLASS statement, all of these elements provide the mechanisms for defining classes as described in the following sections:

- Define state in a class on page 50
- Define behavior in a class on page 51
- Define classes based on other classes on page 53

Define state in a class

Data can exist in the following forms within a class:

- Data members of the class
- Properties of a class
- ABL widgets, streams, and work tables that are referenced by class members
- Local variables of a method within the class

Data members and properties

You define data members and properties of a class in the main block of the class, outside of any method definitions for the class. Class data members and properties have an access mode that determines where and how you can access the data member or property. PRIVATE data members or properties are accessible from within the class where they are defined. An instance can access a PRIVATE data member or property of another instance if they are both instances of the same class. PROTECTED data members or properties are accessible from within the class where they are defined and in any subclass of the defining class. PUBLIC data members or properties are accessible both within the class hierarchy and from outside the class hierarchy where they are defined. To fully encapsulate the data members of a class, you typically make each data member PRIVATE and access it indirectly through a PUBLIC property or PUBLIC method, as appropriate.

You can define data members as a wide variety of ABL data elements, including variables, temp-tables, ProDataSets, and other handle-based objects. Of these data members, only variables can be PUBLIC. Variables and some other data members can be defined as object references to other class-based objects that the class instantiates. You can define a property as any data type that a method or user-defined function can return as a value, including an object reference or ABL array.

You define data members and properties of a class in the main block of the class, outside of any method definitions for the class. Class data members and properties have an access mode that determines where and how you can access the data member or property. PRIVATE data members or properties are accessible from within the class where they are defined. An instance can access a PRIVATE data member or property of another instance if they are both instances of the same class. PROTECTED data members or properties are accessible from within the class where they are defined and in any subclass of the defining class. PACKAGE-PRIVATE variable data members can be accessed from within the class and any class within its package. PACKAGE-PROTECTED variable data members can be accessed from within the class, any class within its package, and from within any subclass that inherits the class. PUBLIC data members or properties are accessible both within the class hierarchy and from outside the class hierarchy where they are defined. To fully encapsulate the data members of a class, you typically make each data member PRIVATE and access it indirectly through a PUBLIC property or PUBLIC method, as appropriate.

You can define data members as a wide variety of ABL data elements, including variables, temp-tables, ProDataSets, and other handle-based objects. Of these data members, only variables can be PACKAGE-PRIVATE, PACKAGE-PROTECTED, or PUBLIC. Variables and some other data members can be defined as object references to other class-based objects that the class instantiates. You can define a property as any data type that a method or user-defined function can return as a value, including an object reference or ABL array.

Class-scoped handle-based objects

Any ABL static widgets, streams, and work tables that you define for a class are not class members that can be inherited by a subclass, but they provide resources that are privately scoped to the class and available for access by all member methods and triggers defined in the class. However, note that handles to ABL widgets, like handles to all handle-based objects, can be stored as variable data members.

Local variables and other data elements of methods

Local variables and other data elements defined within a method do not have an access mode and are always scoped to, and only accessible from within, the method where they are defined. Thus, they are only available during execution of the defining method.

Define behavior in a class

The behavior within a class is defined using methods and events.

Methods of a class

Methods encapsulate executable code for a class, much like internal procedures or user-defined functions for a procedure-based object. Like data members and properties, methods have an access mode, which defines where and how you can call the method. As for data members and properties, the access mode of a method determines the context within an application where the method is visible and available for execution. You can call private methods from within the class in which they are defined. An instance can access a private method of another instance as long as they both belong to the same class. You can call protected methods from within the class hierarchy where they are defined and in any subclass of the defining class. An instance can call a protected method of a second instance of a class that is at the same level or in a super class in the class hierarchy. You can call public methods both from within the class hierarchy and from outside the class hierarchy where they are defined.

Classes support four types of methods:

- Named methods, which you can define as needed and which you invoke by name, similar to procedures or user-defined functions. A method must also specify a return type, but that return type can be VOID, which means that no value is returned by the method. Methods can have zero or more parameters.
- Accessors, which are specified for a property when you define it and which are invoked when you read or write the property.
- Constructors, which you can define as needed and which can be invoked when you instantiate the class. You can provide one or more constructors (overloaded with different parameter lists) when the class needs to initialize data for an object during class instantiation. Providing overloaded constructors allows you to instantiate the same class with different initial data. If you do not provide a constructor, ABL provides a default constructor in order to instantiate the class. The default constructor has no parameters, and if you provide a constructor with no parameters, ABL treats this constructor as the default constructor.

- A destructor, which you can define as needed and which is automatically invoked during garbage collection or when you delete the class instance explicitly. You can provide a destructor when the class needs to free resources or do other cleanup work when a class instance is destroyed. If a destructor is not provided, ABL provides a default destructor in order to delete the object.

Note: Classes also support built-in methods on events of a class (*event methods*). You can manage class events by invoking these methods on them.

Events of a class

In addition to the various types of methods, you can define events as members of a class. A class event can cause one or more specified methods or internal procedures to execute when the defining class *publishes* the event in response to a run-time condition that some method of the class identifies. These methods and internal procedures are referred to as *event handlers*. Any consumer of the class that defines a class event can add (*subscribe*) or remove (*unsubscribe*) such handlers for the event.

As with other members of a class, a class event can be inherited by any derived class, depending on its access mode (`PRIVATE`, `PROTECTED`, or `PUBLIC`). However, access to a class event outside of the class where it is defined only allows you to subscribe event handlers to it using the built-in `Subscribe( )` event method and to unsubscribe event handlers from it using the built-in `Unsubscribe( )` event method. Within the defining class, you can also subscribe and unsubscribe handlers for a event, but you can publish a class event using the built-in `Publish( )` event method only from within the class where it is defined and implemented, regardless of its access mode.

You can subscribe an event handler to a private event from within the class that defines the event. An instance can also subscribe an event handler to a private event of another instance if both instances are from the same class. You can subscribe an event handler to a protected event from within the class that defines the event and from within any class that inherits the defining class. An instance can subscribe an event handler to a protected event from a second instance that is at the same level or in a super class in the class hierarchy. You can subscribe to public events both from within the class hierarchy and from outside the class hierarchy where they are defined.

The class that defines an event controls how and when the event is published. However, an application that consumes this class can subscribe or unsubscribe event handlers to the event from anywhere in the application that the event access mode allows. In addition, because you can both subscribe and unsubscribe event handlers, you can have different handlers registered to execute for the same event when it is published at different times. Thus, you can dynamically change application behavior in response published class events.

A class event also has a signature defined for it with zero or more method parameters. Any event handler subscribed to a class event must have a signature that is compatible with the method signature defined for the event.

In addition to the various types of methods and events of a class, you can also define triggers (`ON` statements) that specify behavior (handler code) for handle-based object events on widgets that you define in a class. Triggers defined in a class are privately scoped to the class and handle events only for widgets activated by methods of the defining class. These triggers thus support the behavior of these class methods. Like static widgets, streams, and work tables, triggers that are part of a class definition are not class members that can be inherited by a subclass, nor can they be affected by an `APPLY` statement executed in another class definition.

Define classes based on other classes

A user-defined class (defined using the CLASS statement) can extend another class by inheriting it. In ABL, a class can inherit from at most one other class, which in turn can inherit from another class. This single inheritance model makes the non-PRIVATE data members, properties, methods, and events of the super class appear as if they are part of the class being defined. If a class is marked as FINAL, it cannot be inherited (used as a super class for another class). The different classes that make up a class hierarchy are stored in separate class files, both in their source code form and their r-code form. However, at both compile time and run time, all the parts of a class hierarchy are treated as a single unit, and must all run in the same ABL session. Thus, an object is an instantiation of a class hierarchy, not simply the class that is instantiated. This object can also represent (have the class type of) any class in that hierarchy, and can be referenced as any one of these classes. For more information on class hierarchies and inheritance, see Class hierarchies and inheritance on page 102.

A class can also be abstract. An abstract class is just like a non-abstract class except for three features:

- It must be defined with the ABSTRACT option.
- It cannot be instantiated, but must be inherited by a derived class; therefore, it cannot be marked FINAL.
- In addition to being able to implement any other class member or element, it can declare abstract properties, methods, and events, each of which must be implemented by a derived class. The implementing class must be the first derived non-abstract class, unless a derived abstract class has already implemented the abstract member.

An abstract property, method, or event is declared with a prototype that defines only type information for the specified class member. No storage or code implementation is associated with an abstract member declaration. The abstract prototype declaration simply creates the requirement that a class member of the specified type must be implemented in a derived class. Otherwise, you can access and manipulate abstract members (according to their type) in a manner similar to implemented members, because an implementation is guaranteed to exist at run time.

A user-defined class can also implement methods, properties, and events whose prototypes are declared by an interface (defined in the INTERFACE statement). An interface is not a class in itself, but similar to a class, it does represent an object type (interface type). An interface type specifies the prototypes of properties, methods, and events for classes to implement, similar to abstract members of a class type. Thus, an interface does not specify any method, property, or event implementations itself, but it only provides a particular set of method, property, or event prototypes that different classes can implement as required. In fact, it is a compiler error for a class not to provide an implementation for every method, property, or event prototype specified in an interface that it implements. An implementing class can meet this requirement by inheriting method, property, or event implementations from a super class, as well as by defining its own method, property, or event implementations.

Again, although an interface is not a class, it does specify the type of an object. Thus, an object instantiated from a class that implements an interface can be referenced as an instance of that interface type as well as an instance of its class type. If a class implements several interfaces, an object instantiated from that class can also be referenced as an instance of each interface type implemented by that class, including any inherited interface types. However, note that you cannot instantiate an interface the way you can instantiate a non-abstract class. You can only reference a class that you have already instantiated as an instance of an interface type that the class implements.

Note that while similar in application, the key difference between interface members and abstract members is that abstract members define prototypes in an abstract class that must be inherited and implemented by a derived class. So, abstract members participate fully in any class hierarchy where they are defined, and can be referenced at the same points in a class hierarchy as any other member. Interface members can only be referenced through the interface type where they are defined or through a class that implements these interface members.

Note: Both abstract classes and interfaces can be used to guarantee the API for applications that implement a delegation model. For more information, see Use delegation with classes on page 130.

Use the CLASS construct

The CLASS construct represents all of the statements that can comprise a class definition. The class definition specifies a main block, starting with a CLASS statement that identifies the class. After any USING statements, the first compilable line in a class definition file that defines a class must be this CLASS statement. The last compilable line in a class definition must be the END CLASS statement, which terminates the main block of the class.

Note: The CLASS statement is only valid in a file with the .cls extension.

This is the syntax for defining a class.

Syntax

```
CLASS class-type-name [ INHERITS super-type-name ]
  [ IMPLEMENTS interface-type-name [ , interface-type-name ] . . . ]
  [ USE-WIDGET-POOL ] [ ABSTRACT | FINAL ]
  [ SERIALIZABLE ] :
    [data-member . . . ]
    [property . . . ]
    [constructor . . . ]
    [method . . . ]
    [event . . . ]
    [destructor]
    [class-scoped-handle-object . . . ]
    [trigger . . . ]
    [udf-prototype . . . ]
END [ CLASS ] .
```

Element descriptions for this syntax diagram follow:

class-type-name

The object type name for the class definition, specified using the following syntax:

```
[ " ] [package.] class-name [ " ]
```

package

A period-separated list of string components that comprise a package name that, along with *class-name*, uniquely identifies the defined class among all accessible classes and interfaces in your application environment.

Note that you cannot specify Progress as the first component of the package name for any ABL user-defined class. For example, Progress.Inventory is an invalid package name for a user-defined class and results in a compiler error.

class-name

The class name, which must match the filename of the class file containing the class definition.

For more information on this syntax, see Define and reference object type names on page 45.

For a class definition, if the class is defined in a package, you must specify *package*, even if the class definition file contains a USING statement that specifies the fully qualified object type name for the class or its *package*. For more information, see Reference an object type name without its package on page 48.

INHERITS *super-type-name*

The immediate super class that this class inherits, where *super-type-name* is the class type name of this super class. Logically, the super class's non-private members can be thought of as part of the current class definition. The super class can be a class that in turn inherits from another class. When a class is defined using the INHERITS phrase, the class being defined is referred to as a subclass. All of the super classes of this subclass constitute the class hierarchy for this class definition.

The syntax for *super-type-name* is the same as for *class-type-name*. However, if a USING statement specifies a fully qualified *super-type-name* or its package, you can specify the super class using its unqualified class name. For more information, see Reference an object type name without its package on page 48. Based on this information, ABL must be able to locate each specified super class definition file (either the source file or the r-code file) in order to compile or run the user-defined class. Also, the specified super class cannot be defined as FINAL.

If you do not specify this option, the class inherits directly from the ABL root class, Progress.Lang.Object. For more information on this built-in class, see Use the root class: Progress.Lang.Object on page 91.

IMPLEMENTS *interface-type-name* **[** , *interface-type-name* **] . . .**

One or more interfaces that this class implements. The current class hierarchy must implement all the properties, methods, and events declared in each interface specified as *interface-type-name*, including all properties, methods, or events inherited from super interfaces.

The syntax for *interface-type-name* is the same as for *class-type-name*. However, if a USING statement specifies a fully qualified *interface-type-name* or its package, you can specify the interface using its unqualified interface name. For more information, see Reference an object type name without its package on page 48. Based on this information, ABL must be able to locate the specified interface class file (either the source file or the r-code file) in order to compile or run this user-defined class.

USE-WIDGET-POOL

Creates one or two unnamed widget pools, one that is scoped to the current class instance for handle-based objects associated with instance members of the class, and a separate unnamed

widget pool scoped to the class type for handle-based objects associated with static members of the class. The specified instance unnamed pool serves as the default widget pool for all dynamic widgets and other dynamic handle-based objects that the class instance creates, unless there is a more locally-scoped unnamed widget pool in effect when the handle-based object is created. The AVM puts a dynamic handle-based object into the current default widget pool unless the statement that creates it uses the `IN-WIDGET-POOL` option.

The AVM deletes the associated class unnamed widget-pool (specified using this option) when each instance of the class is deleted. The specified static unnamed pool serves as the default widget pool for all handle-based objects created in the context of any static members of the class and persists for the duration of the session. For more information on static class members, see Use static members of a class on page 186.

If a class definition does not include the `USE-WIDGET-POOL` option, the AVM follows the same rule for choosing the default widget pool as in procedures:

- If one or more locally-scoped unnamed widget pools have been created, the AVM uses the most locally-scoped unnamed widget pool.
- If no locally-scoped unnamed widget pools have been created, the AVM uses the session unnamed widget pool.

For more information on using widget pools with classes, see Use widget pools on page 223.

`ABSTRACT`

If specified, this class is defined as abstract, allowing it to define abstract class members (as well as non-abstract class members), including properties, methods, and events. You cannot instantiate an abstract class, and whether or not it defines abstract members, it must be inherited by another class; therefore, an abstract class cannot be defined with the `FINAL` option.

The first non-abstract subclass that inherits from this abstract class must also implement any abstract members that are inherited in its class hierarchy. An implemented member is any inherited abstract member that is defined with the `OVERRIDE` option and without the `ABSTRACT` option. For more information, see Define properties within a class on page 62, Define methods within a class on page 71, and Define events within a class on page 77.

`FINAL`

If specified, this class cannot be inherited. That is, it cannot be specified as a super class for the `INHERITS` option of another class definition.

`SERIALIZABLE`

Indicates instances of the class can be:

- passed between an application server and a remote ABL client
- serialized to a binary or JSON stream

All classes in the hierarchy must be marked as `SERIALIZABLE`, so using this option on a class that inherits from a non-serializable class will raise a compile-time error. The class types of any data members that are themselves class-based objects must also be `SERIALIZABLE`. The `SERIALIZABLE` option cannot be used with ABL-extended .NET classes.

For additional information on the restrictions on serialization, see Pass object reference parameters on page 159. For the basics of serializing an object to a stream, see Serialization and deserialization on page 213.

data-member

A data member definition. A class can define zero or more data members of the class. For more information, see Define data members within a class on page 58.

property

A property definition. A class can define zero or more properties of the class. For more information, see Define properties within a class on page 62.

method

A method definition. A class can define zero or more methods of the class. For more information, see Define methods within a class on page 71.

event

A class event definition. A class can define zero or more events of the class. For more information, see Define events within a class on page 77.

constructor

A constructor definition. A class can define zero or more constructors for the class. For more information, see Define class constructors on page 84.

destructor

A destructor definition. A class can define one destructor for the class. For more information, see Define the class destructor on page 88.

class-scoped-handle-object

A static ABL widget, stream, or work-table definition. A class can define zero or more of these static handle-based objects to provide private resources for a class definition. For more information, see Define class-scoped handle-based objects on page 91.

trigger

A trigger definition. A class can define zero or more `ON` statements that specify triggers for a class. Each `ON` statement specifies a response to a specific set of handle-based object events, mostly from interactions with ABL widgets. For more information, see Use handle-based object events in classes on page 221.

udf-prototype

A user-defined function prototype. If you invoke a user-defined function in a procedure object handle from within the method of a class, you must provide the function prototype and a handle to the running external procedure where the user-defined function is defined. You must also provide the function prototype with the `IN SUPER` option if the user-defined function is defined in a super procedure for the session. This is the same prototype and handle reference that you must provide in any procedure that references an external user-defined function. For more information, see the reference entry for the `FUNCTION` statement in *ABL Reference*.

You can terminate a `CLASS` statement with either a period (`.`) or a colon (`:`). The *data-member*, *property*, *constructor*, *method*, *destructor*, *trigger*, and *udf-prototype* specifications can appear in any order.

The following sample shows an abstract class definition:

```
CLASS acme.myObjs.Common.CommonObj ABSTRACT:
...
END CLASS.
```

The following sample shows a class definition that inherits from the previous class, its super class:

```
USING acme.myObjs.Common.*.
CLASS acme.myObjs.CustObj INHERITS CommonObj:
...
END CLASS.
```

Note the `USING` statement that allows an unqualified reference to the `CommonObj` class name to specify its class type. However, the class type specified for the class definition must be fully qualified with its package.

The following sections, and other sections in this book, make reference to and build on this sample class definition (referred to as "the sample class"). This class and other associated class and interface definitions in this and other topics (referred to as "a sample class" or "a sample interface") sometimes represent partial or modified implementations of the sample classes that are fully presented in a later topic. For more information, see Compare constructs in classes and procedures on page 225.

Define data members within a class

Data members define instance data of a class. They must be defined in the main block of the class. You can define variables, buffers, temp-tables, queries, ProDataSets, and data-sources as data members of a class, with restrictions as noted below. You can also define certain types of data as properties. For more information, see Define properties within a class on page 62.

Data members of a class are defined using the standard ABL `DEFINE` statements, with the addition of an optional access mode for each data member. The access mode is valid only for data member definitions in the main block of a class definition file (not for local method data, for example). Temp-tables and ProDataSets must be defined as data members of the class. They cannot be defined within a method of a class. This restriction is equivalent to external procedures, where temp-tables and ProDataSets cannot be defined within internal procedures or user-defined functions.

This is the syntax for defining a variable data member.

Syntax

```
DEFINE [ PRIVATE | PACKAGE-PRIVATE | PROTECTED | PACKAGE-PROTECTED | PUBLIC ][
STATIC ]
  [ SERIALIZABLE | NON-SERIALIZABLE ]{ VARIABLE }
  data-member-name data-member-definition .
```

This is the syntax for defining handle-based data members, which cannot be defined as PUBLIC:

```
DEFINE [ PRIVATE | PACKAGE-PRIVATE | PROTECTED | PACKAGE-PROTECTED ][ STATIC ]
  [ SERIALIZABLE | NON-SERIALIZABLE ]
  { BUFFER | TEMP-TABLE | QUERY | DATASET | DATA-SOURCE }
  data-member-name data-member-definition .
```

Element descriptions for these data member syntax diagrams follow:

[PRIVATE | PACKAGE-PRIVATE | PROTECTED | PACKAGE-PROTECTED | PUBLIC]

The access mode for the data member. The default access mode is PRIVATE.

Only data members in the CLASS block can have an Access modes. Local variables defined within methods cannot have an access mode and are accessible only from within the method, similar to the way local variables defined within an internal procedure are accessible only within the defining internal procedure.

[STATIC]

Specifies the scope of the data member. By default, data members of a class are instance data members, meaning that one copy of a given data member is created and scoped to each instance of the class in which it is defined. However, data members can also be defined as static by specifying this keyword. ABL creates only one copy of a static data member on first reference to the class type in which it is defined, scopes that data member to the class type, and makes it available through that class type throughout the ABL session without the need for a class instance. Unless specified otherwise, any reference to a data member in this manual is assumed to be a reference to an instance data member.

For more information on static data members, see Use static members of a class on page 186.

Note: A static data member, which is scoped to a class type, is distinct from a static widget or other static handle-based object. A static handle-based object, such as a menu, temp-table, or ProDataSet, is a compile-time defined and bound object that can also be implemented as an equivalent dynamic handle-based object, which can be created and bound at run time only. Thus, you can define a static handle-based object, such as a temp-table, that is also a static data member, and you can define a HANDLE variable as a static data member, which can, in turn, reference a dynamic or static handle-based object. Typically, the usage of the term, *static*, can be understood from the context. Otherwise, OpenEdge content attempts to clarify the meaning in any ambiguous context.

[SERIALIZABLE | NON-SERIALIZABLE]

Class-based objects that are defined as serializable (using the SERIALIZABLE option in the CLASS statement) can be passed as parameters in remotes call between the application server and ABL clients and can be serialized to binary or JSON format. By default, both passing an object as a parameter and serializing an object to binary via the Progress.IO.BinarySerializer class include all data members regardless of access mode. However, for JSON serialization via Progress.IO.JsonSerializer, only public data members are serialized. To include a protected or private variable, temp-table, or ProDataSet during JSON serialization, SERIALIZABLE must be added to the definition. See the Serialize( ) method (JSON) entry in *ABL Reference* for more information.

Use the NON-SERIALIZABLE option to exclude a given variable from parameter passing between an application server and ABL client or from the serialization process via the `Progress.IO.BinarySerializer` or `Progress.IO.JsonSerializer` class. A variable marked as NON-SERIALIZABLE is assigned its initial value when the object is deserialized.

Note: The SERIALIZABLE option can only be used with variables, temp-tables, and ProDataSets. Using it with buffers, queries, or data sources will raise a compiler error.

`data-member-name`

The name of the variable, buffer, temp-table, query, ProDataSet, or data-source. This *`data-member-name`* must be unique within the class hierarchy according to its particular class member namespace. For example, a variable data member must be unique within a namespace that includes the names of properties, variable data members, and class events. For more information, see Namespaces for class members on page 83.

Variable data members defined within a class can be named the same as an ABL reserved keyword. In all contexts, a variable data member whose name is a reserved keyword must be qualified when referenced, either by an object reference (which can be `THIS-OBJECT`) or if it is static, with a static type name. For more information on accessing data members, see Access data members and properties on page 153. For more information on using static type-name syntax to reference static variable data members, see Access static members on page 186.

`data-member-definition`

The remaining syntax available for defining the variable, buffer, temp-table, query, ProDataSet, or data-source.

Note that some of these data members are complex data members. A *complex data member* is a data member, such as a ProDataSet, that must be defined together with one or more component data members. In general, the component data members of an associated complex data member must be defined with an equal or less restrictive access mode than the associated complex data member. For example, if you define a `PRIVATE` ProDataSet, the temp-tables and buffers of the ProDataSet can be defined with the `PROTECTED` access mode. Also, the component data members of a complex static data member definition can consist **only** of other static data members that are defined with a compatible access mode in the current class hierarchy. However, the component data members of a complex instance data member can consist of **both** instance and static data members.

For example, adding four data members to the sample class definition results in this code:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj:
  DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.

   DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
      FIELD RecNum AS INTEGER
      FIELD CustNum LIKE Customer.CustNum
      FIELD Name LIKE Customer.Name
      FIELD State AS CHARACTER.

  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.
  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
...
END CLASS.
```

For PUBLIC and PROTECTED definitions the data member name is available to the class that defines it and to all subclasses of this class. The definition of a PUBLIC or PROTECTED data member does not need to be repeated in the subclass or the referencing class. The use of the data member name in a subclass refers to the data member defined in the subclass or to data members defined in a super class within its class hierarchy. There is no separate copy of the data member when it is referenced from a subclass. If a subclass references a data member from a super class, the compiler accesses the data member definition in the super class to identify its type information. If the data member definition is repeated in the subclass, this results in a compiler error.

When a PUBLIC instance data member is accessed outside of its class hierarchy, it is accessed through an object reference, using the *object-reference*:*data-member-name* syntax. The data member is defined in the referenced class and no separate copy is made in the referencing class. That is, the referencing class does not repeat the definition for a data member that it references in another object. A data member definition of the same name in the referencing class is treated as an entirely separate data member from the one in the referenced class.

For example, assuming the previous class definition for acme.myObjs.CustObj suppose we define a new class acme.myObjs.NECustObj that inherits and references the data members of its super class acme.myObjs.CustObj. If the class CustObj defines behavior for handling Customer data in the application, the new subclass NECustObj can be thought of as a specialization of that class that defines additional or overridden behavior just for Customer records for the New England area.

In the following variation of a sample class, NECustObj, notice that both the PUBLIC and PROTECTED data members, iNumCusts and ttCustomer, are inherited from CustObj and accessible within its definition:

```
USING acme.myObjs.*.

CLASS acme.myObjs.NECustObj INHERITS CustObj:

  METHOD PRIVATE VOID DisplayCust ( ):
    iNumCusts = iNumCusts + 1.
      FOR EACH ttCustomer:
         DISPLAY ttCustomer.
      END.
  END METHOD.

END CLASS.
```

The following is another class, MyMain, that uses the NEW statement to create an instance of the acme.myObjs.NECustObj class and assign its object reference to the variable rNECust. In this case, only the PUBLIC data member iNumCusts of NECustObj is accessible to this class:

```
USING acme.myObjs.*.

CLASS MyMain:

  DEFINE PRIVATE VARIABLE rNECust AS CLASS NECustObj NO-UNDO.

  METHOD PRIVATE VOID DisplayCust ( ):
    rNECust = NEW NECustObj( ).
    rNECust:iNumCusts = rNECust:iNumCusts + 1.
    /* Can not access PROTECTED ttCustomer */
  END METHOD.

END CLASS.
```

Comparison with procedure-based programming variables and data elements

The variables and other data elements defined in the main block of an external procedure are private without having an explicit access mode. That is, by default, they are scoped to and accessible from elements of the procedure, including any of its internal procedures or user-defined functions, and they cannot be directly accessed from other external procedures. In the same way, data members defined with no access mode in a class are PRIVATE and accessible from all methods defined in the class; and they cannot be directly accessed from other classes, whether inside or outside of the class hierarchy, or from external procedures.

Procedures can define SHARED data elements, which are scoped to and accessible from other external procedures. For SHARED data elements, the data definitions must be defined as NEW SHARED in one external procedure and these same data definitions must be repeated as SHARED in each called external procedure that accesses them. This is necessary because the compiler does not look at other external procedures when compiling a procedure.

Classes do not support SHARED data elements. However, SHARED data elements in procedures roughly correspond to PROTECTED and PUBLIC data members in classes. PROTECTED and PUBLIC data members in classes provide wider access without the need to repeat the data member definitions as is required for SHARED data elements in procedures. However, data members must be referenced with respect to the instantiated object where they are defined. So, PROTECTED data members can only be referenced within the same class hierarchy of a single instantiated object where they are defined; and PUBLIC data members can be referenced like PROTECTED data members, within the class hierarchy, and also from outside the class hierarchy. However, PUBLIC data members can only be referenced outside the class hierarchy where they are defined by using a qualifying prefix associated with the defining class. For an instance data member, the qualifying prefix is an object reference to an instance of the defining class, and for a static data member, the qualifying prefix is the type name of the defining class. These referential restrictions both identify class and object relationships to the data members and allow the compiler to validate data member references for consistency based on these relationships.

Define properties within a class

You can define certain types of data for a class as properties. Properties are always class members and must be defined in the main block of the class. The data types for properties are restricted to those that you can specify as the return type of a method. You must define all other types of class data as data members or class-scoped handle-based objects. For more information, see Define data members within a class on page 58 and Define class-scoped handle-based objects on page 91.

Unlike data members, the DEFINE statement for defining properties provides a built-in mechanism for encapsulating data using special methods called accessors. The accessors you define determine if a given property can be read, written, or both. Accessors can also directly access four general categories of data:

1. Default memory for the property that has the same data type as the property definition
2. Any other accessible data members of the class hierarchy where the property is defined
3. Any PUBLIC data that is available from other classes, as appropriate
4. Any internal data defined within the accessors themselves

Depending on how you implement these accessors, when you access the property, you can directly access the property's default memory, or you can indirectly access and process other data elements of various types that are available to the accessors. Thus, property accessors can encapsulate state and behavior similar to a method. No matter how you implement a property's accessors, you can access a property in exactly the same way as you access an equivalently defined variable data member.

Although a property accessor can access data of any type, the data that a property accesses is typically defined with the same data type as the property itself, which is the most common and effective use of a property definition. To effectively encapsulate types of data that you cannot define explicitly as a property data type, like temp-tables, you can define named methods with interfaces designed to handle these data types. For more information on defining methods, see Define methods within a class on page 71.

By default, a property is an instance member that is defined for access, according to its access mode, only within or through the instance of the class in which it is defined. A property can also be defined as static, which associates the property with the class type (not an instance) and makes it available, according to its access mode, throughout the ABL session without the need for a class instance. For more information on accessing static properties, see Access static members on page 186.

You can also define an abstract property prototype in an abstract class type definition, and you can define an interface property prototype in an interface type definition. A property prototype defines its data type and certain other options, but no implementation. Both types of property prototype define a property that must be implemented appropriately in the class hierarchy where it is defined. For more information on defining interface property prototypes, see Define interfaces on page 93 and Use the INTERFACE construct on page 93.

DEFINE PROPERTY statement

This is the syntax for defining a property using the `DEFINE PROPERTY` statement:

Syntax

Element descriptions for this syntax diagram follow:

[`PRIVATE` **|** `PACKAGE-PRIVATE` **|** `PROTECTED` **|** `PACKAGE-PROTECTED` **|** `PUBLIC` **]**

The access mode for the property. The default property access mode is `PUBLIC`. `PRIVATE` properties are accessible from within the class where they are defined. An instance can access a `PRIVATE` property of another instance if they are both instances of the same class. `PROTECTED` properties are accessible from within the class where they are defined and in any subclass of the defining class. An instance can access a `PROTECTED` property of a second instance of a class that is at the same level or in a super class in the class hierarchy. A `PACKAGE-PRIVATE` property can be accessed from within the class and any class within its package. A `PACKAGE-PROTECTED` property can be accessed from within the class, any class within its package, and from within any subclass that inherits the class.`PUBLIC` properties are accessible both within the class hierarchy and from outside the class hierarchy where they are defined. For more information on accessing properties, see Access data members and properties on page 153.

An abstract property prototype cannot be defined as `PRIVATE`, and an interface property prototype can only be defined as `PUBLIC`.

Note: If the property is both readable and writable, a more restrictive access mode can be set for either reading or writing the property (but not both) using the *`accessor-spec`*.

[`STATIC` **]**

Specifies the scope of the property. By default, properties of a class are instance properties, meaning that one copy of a given property is created and scoped to each instance of the class in which it is defined. However, properties can also be defined as static using this keyword. ABL creates only one copy of a static property on first reference to the class type in which it is defined, scopes that property to the class type, and makes it available through that class type for the duration of the ABL session. Unless specified otherwise, any reference to a property in this manual is assumed to be a reference to an instance property.

You cannot define a static property as abstract.

For more information on static properties, see Use static members of a class on page 186.

[ABSTRACT]

Declares the property as abstract using an abstract property prototype. Each abstract property prototype is declared by a single DEFINE PROPERTY statement, with one or both of the two possible accessors specified, but with no accessor implementations. This means that appropriate access mode and data type information is specified for each accessor, but the accessor code block must be empty. Abstract properties can be defined as either PROTECTED or PUBLIC. These property prototypes cannot specify the INITIAL or STATIC option. An abstract property has no storage allocated for it to hold an initial value and it can only be an instance property.

An abstract property must be implemented in some class (abstract or non-abstract) that derives from the class that defines the abstract property (see the). The class that implements an abstract property must fully define a corresponding property that matches this property prototype, but without the ABSTRACT option. It must define at least the accessors specified in the prototype, but it can also define any accessor that is not specified in the prototype. In addition the implemented property can have a less restrictive access mode, if available (PUBLIC if the abstract property is PROTECTED).

You can only define an abstract property prototype in an abstract class definition.

[OVERRIDE]

[SERIALIZABLE | NON-SERIALIZABLE]

Class-based objects that are defined as serializable (using the SERIALIZABLE option in the CLASS statement) can be passed as parameters in remotes call between the application server and ABL clients and can be serialized to binary or JSON format. By default, both passing an object as a parameter and serializing an object to binary via the Progress.IO.BinarySerializer class include all data members regardless of access mode. However, for JSON serialization via Progress.IO.JsonSerializer, only public data members are serialized. To include a protected or private property during JSON serialization, SERIALIZABLE must be added to the definition. See the Serialize( ) method (JsonSerializer) entry in *ABL Reference* for more information.

Use the NON-SERIALIZABLE option to exclude a given property from parameter passing between an application server and ABL client and from the serialization process via the Progress.IO.BinarySerializer or Progress.IO.JsonSerializer class. A property marked as NON-SERIALIZABLE is assigned its initial value when the class is deserialized.

You can also use the NON-SERIALIZABLE option to make a class serializable that otherwise would not be because one or more of its members are not serializable. For example, if a property itself is a class-based object that is not serializable, adding the NON-SERIALIZABLE keyword to that property will allow the class that contains it to be defined as SERIALIZABLE. See Pass object reference parameters on page 159 for full details on what can and cannot be serialized.

Note: Use the SERIALIZABLE option only with properties that are both readable and writable. This option is not for use with properties defined as ABSTRACT or as part of an interface.

property-name

The property name, which identifies either the property or its default memory, depending on the context. This *property-name* must be unique within the class hierarchy according to its particular

class member namespace. A property name must be unique within a namespace that includes the names of properties, variable data members, and class events. For more information, see Namespaces for class members on page 83.

Properties defined within a class can be named the same as an ABL reserved keyword. In all contexts, a property whose name is a reserved keyword must be qualified when referenced, either by an object reference (which can be `THIS-OBJECT`) or if it is static, with a static type name. For more information on accessing properties, see Access data members and properties on page 153. For more information on using static type-name syntax to reference static properties, see Access static members on page 186.

type-spec

The data type of the property and its default memory, specified using this syntax:

```
AS { primtive-type-name | [ CLASS ] object-type-name }
   [ EXTENT [ constant ] ]
```

Any data type that you specify must be one that is allowed as a return type for a method, including a built-in ABL primitive type (*primtive-type-name*), or a class or interface type (*object-type-name*). If you specify a class or interface type, *object-type-name* can be a fully qualified class name (specifying the package). If you also specify the fully qualified class name or its package with a `USING` statement, *object-type-name* can be the unqualified class or interface name. For more information on referencing object type names, see Define and reference object type names on page 45.

The optional EXTENT [*constant*] defines the property as an array of data elements, where the element data type is specified by either the AS *primitive-type-name* option or the AS *object-type-name* option. This option can specify an array property as either determinate (has a defined number of elements) or indeterminate (has an undefined number of elements). To define a determinate array property, specify the `EXTENT` option with the *constant* argument. This optional argument is an integer constant value that represents the number of elements in the property array. To define an indeterminate array property, specify the `EXTENT` option without the *constant* argument. This option participates in defining the data type of the property. For more information about arrays, see Work with one-dimensional arrays in *Basic ABL*. If you do not use the `EXTENT` option (or you specify *constant* as 0), the property is not an array property.

```
[ INITIAL { constant | [ constant [ , constant ] . . . ] } ]
```

This option specifies an initial value for an implemented property, depending on the data type. It allows you to define a single value for a scalar property or all of the element values for an array property, specified as a comma-separated list within the required square brackets (`[]`).

You cannot specify this option for an abstract or interface property prototype.

```
[ NO-UNDO ]
```

Similar to data members, if a transaction is undone in which the property is accessed, this option retains the latest change to the value of property default memory during the transaction.

accessor-spec

One or two accessors defined for the property that indicate if the property is readable, writable, or both.

This syntax specifies two accessors that allow the property to be both readable and writable:

```
[ PRIVATE | PACKAGE-PRIVATE | PROTECTED |
PACKAGE-PROTECTED ] get-accessorset-accessor
  | get-accessor [ PRIVATE | PROTECTED | PACKAGE-PRIVATE |
PACKAGE-PROTECTED ] set-accessor
```

The *get-accessor* represents an accessor that allows the property to be read, and the *set-accessor* represents an accessor that allows the property to be written. The optional access mode applies only to the immediately following accessor and overrides the specified property access mode. The other accessor always inherits the property access mode. If you specify an access mode for one accessor, it must be **more** restrictive than the property access mode that applies to the other accessor. However, note that if the property is abstract, the accessor access mode can never be PRIVATE, because it must be implemented in, and accessible from, an inheriting class.

So for example, if the property access mode is PROTECTED and you specify a PRIVATE access mode on the *set-accessor*, the *get-accessor* assumes the access mode of the property, which is PROTECTED. This specifies that the property can only be written from the defining class or an instance of the defining class. The property can be read only from the defining class, an instance of the defining class, a subclass of the defining class, or an instance of a subclass of the defining class.

If you want the property to be both read and written using the property access mode (the default), do not specify an accessor access mode.

This syntax specifies a single accessor that allows the property to be read:

```
get-accessor
```

This accessor allows the property to be read according to the property access mode. If this is the only accessor that you specify, the property is read-only. You cannot override the property access mode for a read-only property.

This is the syntax for a *get-accessor*, which specifies how the property is read:

```
GET. | GET ( [ array-index-parameter ] ) : get-logic END [ GET ] .
```

You can define one of two types of accessors for reading the property:

GET.

A *simple* GET accessor (defined **without** an implementation). The property value is read directly from the current value of the default memory for the property. This is the only *get-accessor* syntax that you can specify for an abstract or interface property prototype.

```
GET ( [array-index-parameter] ) : get-logic END [ GET ].
```

A *coded* GET accessor (defined **with** an implementation). The property value is read from a value returned from *get-logic*. This *get-logic* can include any number and types of statements that are valid for a method. These statements can read and write the value of the default memory specified by *property-name* or the values of any other accessible data elements. If the property is defined as static, *get-logic* statements can access **only** other static members defined within the current class hierarchy in addition to its default memory and local accessor data; instance members defined within the current class hierarchy are inaccessible. However, if the property is defined as an instance member, *get-logic* statements can access static members as well as other instance members.

If the property is defined as an array using the EXTENT option, the *get-logic* always works on one element of the array at a time, whether the array is read with or without a subscript. If the array is read without a subscript, *get-logic* executes once for each element in the property array. If specified, the optional *array-index-parameter* passes the subscript value to the *get-logic* for the current element being accessed. This is the syntax:

```
[ INPUT ]array-index-name AS { INTEGER | INT64 }
```

The *array-index-name* provides a name for the index parameter, which can have one of the specified integer data types. Whatever retrieval operation you code for the *get-logic* can then access the appropriate array element of the property default memory by using *array-index-name* as the array subscript (*property-name*[*array-index-name*]).

To return a value to the property reader, you must use the RETURN statement. This statement can return any value compatible with the property's data type, regardless of the value of *property-name*. If you do not invoke a RETURN statement in *get-logic*, the property returns the Unknown value (?) to the reader.

Note: Any statement in *get-logic* that assigns a value to *property-name* invokes the property SET accessor to assign the value. However, any *get-logic* statements that read the value of *property-name* directly read the value most recently assigned to property default memory.

If a statement in *get-logic* raises an ERROR condition that is not handled using the NO-ERROR option, the condition can be handled by the associated block according to its ON ERROR definition or the presence of an appropriate CATCH statement. If the associated block is the accessor, itself, it can also handle the condition according to the default ON ERROR handling for method and procedure blocks or according to any specified ROUTINE-LEVEL ON ERROR UNDO, THROW statement.

A RETURN ERROR or UNDO, THROW in *get-logic* that raises ERROR beyond the GET accessor block, raises the ERROR condition on the statement that is reading the property. Any data elements (including property default memory) that have been changed by *get-logic* retain their latest values or revert to their values before this statement executed, depending on their respective NO-UNDO settings. Any optional RETURN-VALUE setting or error object can be appropriately accessed after control returns from the statement that is reading the property. For more information on handling errors returned from properties, see Raise and handle error conditions on page 197.

This syntax specifies a single accessor that allows the property to be writable:

```
set-accessor
```

This accessor allows the property to be written according to the specified property access mode. If this is the only accessor that you specify, the property is write-only. You cannot override the property access mode for a write-only property.

This is the syntax for a *set-accessor*, which specifies how the property is written:

```
SET. | SET ( parameter-definition [ , array-index-parameter ] ) :
  set-logic END [ SET ].
```

You can define one of two types of accessors for writing the property:

SET.

A *simple* SET accessor (defined **without** an implementation). The value written to the property is assigned directly to the default memory for the property. This is the only *set-accessor* syntax that you can specify for an abstract or interface property prototype.

SET (*parameter-definition* [, *array-index-parameter*]) : *set-logic* END [SET].

A *coded* SET accessor (defined **with** an implementation). The value written to the property is assigned to the parameter specified by *parameter-definition*, which has the following syntax:

```
[ INPUT ] parameter-name AS
  { primtive-type-name | [ CLASS ] object-type-name }
```

The INPUT parameter (*parameter-name*) holds the value being written to the property as the property is being set, but does not by itself set the property value. The data type that you specify for the parameter (*primitive-type-name* or *object-type-name*) must match the data type you have defined for the property. How the property value is set depends entirely on the statements in *set-logic*. These statements can include any number and types of statements that are valid for a method. Thus, *set-logic* statements can access the value being written to the property (*parameter-name*), and they can read or write the value of property default memory (*property-name*) or the values of any other data elements that are available to the accessor like any method in the class definition. However, for the property value to be set, at least one statement must assign the value of *parameter-name* to either *property-name* or to some other data element maintained by the SET accessor. If the property is defined as static, *set-logic* statements can access **only** other static members defined within the current class hierarchy in addition to its default memory and local accessor data; instance members defined within the current class hierarchy are inaccessible. However, if the property is defined as an instance member, *set-logic* statements can access static members as well as other instance members.

If the property is defined as an array using the EXTENT option, the *set-logic* always works on one element of the array at a time, whether the array is written with or without a subscript. If the array is written without a subscript, *set-logic* executes once for each element in the property array. If specified, the optional *array-index-parameter* passes the subscript value to the *set-logic* for the current element being accessed. This is the syntax:

```
[ INPUT ] array-index-name AS { INTEGER | INT64 }
```

The *array-index-name* provides a name for the index parameter, which can have one of the specified integer data types. Whatever retrieval operation you code for the *set-logic* can then access the appropriate array element of the property default memory by using *array-index-name* as the array subscript (*property-name*[*array-index-name*]).

Note: Any statement in *set-logic* that reads *property-name* reads the value returned by the property GET accessor. However, any *set-logic* statements that assign a value to *property-name* directly assign that value to property default memory.

If a statement in *set-logic* raises an ERROR condition that is not handled using the NO-ERROR option, the condition can be handled by the associated block according to its ON ERROR definition or the presence of an appropriate CATCH statement. If the associated block is the accessor, itself, it can also handle the condition according to the default ON ERROR handling for method and procedure blocks or according to any specified ROUTINE-LEVEL ON ERROR UNDO, THROW statement.

A ***RETURN ERROR*** or UNDO, THROW in ***>set-logic*** that raises ERROR beyond the SET accessor block, raises the ERROR condition on the statement that is reading the property. Any data elements (including property default memory) that have been changed by *set-logic* retain their latest values or revert to their values before this statement executed, depending on their respective NO-UNDO settings. Any optional RETURN-VALUE setting or error object can be appropriately accessed after control returns from the statement that is writing the property. For more information on handling errors returned from properties, see Raise and handle error conditions on page 197.

The following example shows a property definition and its access within another sample class definition, `acme.myObjs.CreditObj`:

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.

CLASS acme.myObjs.CreditObj:

  DEFINE PUBLIC PROPERTY CustCreditLimit AS DECIMAL INITIAL ? NO-UNDO
    /* GET: Returns the credit limit of the current Customer.       */
    /* If there is no current Customer, it returns Unknown (?).     */
    GET .
    /* SET: Raises the credit limit for Customers in good standing. */
    /* Current increase is $1,000.                                  */
    PROTECTED SET (INPUT piCL AS DECIMAL):
      IF Customer.Balance > piCL THEN DO:
        CustCreditLimit = Customer.Creditlimit.
        UNDO, THROW NEW Progress.Lang.AppError( "Over Limit" ).
      END.
      ELSE
        ASSIGN
          Customer.Creditlimit = piCL + 1000
          CustCreditLimit      = Customer.Creditlimit.
    END SET.
  ...

  METHOD PUBLIC VOID CheckCustCredit ( ):
    /* Invokes the CustCreditLimit property SET accessor */
    IF AVAILABLE Customer THEN
      CustCreditLimit = Customer.Creditlimit.
    ELSE
      UNDO, THROW NEW Progress.Lang.AppError( "No Customer" ).
  END METHOD.

END CLASS.
```

In this definition, the property `CustCreditLimit` is publicly readable, but writable only within the definition of `CreditObj` or any subclass that might be defined. (The `CreditObj` of a particular instance is also writable from another instance of the same class or subclass.) This property definition uses its default memory (`DECIMAL`) to store the property value and provides an implementation for only one of its accessors (the `SET`), which interacts with the `Customer` buffer currently available in `CreditObj` (initialization code not shown). Note that the `SET` accessor throws an error object for an application condition and the public method, `CheckCustCredit( )`, defined in the same class writes the property value that might trigger the error. Because the property is defined as `NO-UNDO`, the `SET` value for the property default memory is retained, even when it raises `ERROR` in the caller. Any error object is then automatically thrown out of the `SET` accessor because the class is defined with a `ROUTINE-LEVEL ON ERROR UNDO, THROW` statement.

For more information on accessing properties, see Access data members and properties on page 153. For more information on handling property error conditions, see Raise errors within a property on page 201.

Comparison with procedure-based programming user-defined functions

In procedures, the closest equivalent to a property is a user-defined function. You might define one function equivalent to the `GET` accessor that returns a value, and another function (or internal procedure) equivalent to the `SET` accessor that accepts the setting value as an `INPUT` parameter.

Define methods within a class

Methods define the behavior of a class. A by default, a method is an instance member that is defined for access, according to its access mode, only within or through the instance of the class in which it is defined. A method can also be defined as static, which associates the method with the class type (not an instance) and makes it available, according to its access mode, throughout the ABL session without the need for a class instance. For more information on calling static methods, see Access static members on page 186.

Instance methods can access any data members, properties, and other methods of the class including non-private (`PUBLIC` or `PROTECTED`) data members and properties of each super class in the class hierarchy. Method definitions are only valid in a class and cannot be defined in a procedure. Methods also have an access mode that defines where and how the method can be called. The possible access modes are `PRIVATE`, `PROTECTED`, and `PUBLIC`, and `PUBLIC` is the default. `PRIVATE` methods are accessible from within the class where they are defined. An instance can access a `PRIVATE` method of another instance if both instances are of the same class. `PROTECTED` methods are accessible from within the class where they are defined and in any subclass of the defining class. An instance can access a `PROTECTED` method of a second instance of a class that is at the same level or in a super class in the class hierarchy. `PUBLIC` methods can be invoked by the class defining them, by any class that inherits from that class, and by any class or procedure that instantiates that class using its object reference. For more information on calling `PUBLIC` methods, see Call instance methods from outside a class hierarchy where they are defined on page 144.

A method can return a value, including a `CHARACTER`, `CLASS`, `COM-HANDLE`, `DATE`, `DATETIME`, `DATETIME-TZ`, `DECIMAL`, `HANDLE`, `INT64`, `INTEGER`, `LONGCHAR`, `LOGICAL`, `MEMPTR`, `RAW`, `RECID`, or `ROWID`; where `CLASS` specifies an object reference to a class. To return a specific value, use the `RETURN` statement from within the method. A method can also specify `VOID`, which means that the method does not return a value. The code within a method is made up of ABL statements much the same as in a procedure. Methods can also use the syntax described in this manual that is restricted to classes and cannot use certain syntax that is relevant only for procedures. For example, if the method returns a value, it cannot invoke any statement that blocks for input, such as `UPDATE`, `SET`, `PROMPT-FOR`, `CHOOSE`, `INSERT`, `WAIT-FOR`, and `READKEY`. You can also raise `ERROR`, from within a method using a `RETURN ERROR` or a `UNDO, THROW`, and optionally return a `RETURN-VALUE` setting or a specified error object to the caller. For more information on handling errors returned from methods, see Raise and handle error conditions on page 197.

You can also define an abstract method prototype in an abstract class type definition, and you can define an interface method prototype in an interface type definition. A method prototype defines its name, return type, and certain other options, but no implementation. Both types of method prototype define a method that must be implemented appropriately in the class hierarchy where it is defined. For more information on defining interface method prototypes, see Define interfaces on page 93 and Use the INTERFACE construct on page 93.

A subclass can define a method using the same name as a method in its super class using the `OVERRIDE` option. This new definition overrides the method of the same name and scope in the super class. Only named methods can be overridden, not constructors or destructors. The method name, return type, number of parameters, and each corresponding parameter type must match between the super class method and the subclass method. The access modes of the methods must match or be overridden by a less restrictive access mode. For example, a `PUBLIC` subclass method can override a `PROTECTED` super class method. In addition, only an instance method can override another instance method, and only a static method can override another static method. To implement an abstract method, you **must** override it with an instance method defined in some class derived from the class that defines the abstract method.

Overriding works differently for instance and static methods. Any class that inherits an instance method that is overridden higher up in the class hierarchy can access only two implementations of the method. It can call and execute the implementation that is inherited or defined by its immediate super class using the SUPER system reference. No other implementation of the overridden method is available above the implementation that is inherited or defined by its super class. Otherwise, the implementation of the method that executes when the class calls the same method (without SUPER) is always the most derived implementation in the instance class hierarchy, whether that implementation is defined in the calling class or overridden by one or more subclasses. For more information on overriding instance methods, see Override methods within a class hierarchy on page 107.

For a static method, any override is really a re-definition of a super class implementation. A class that calls a static method always executes the nearest implementation defined in or above it in the class hierarchy. However, unlike an instance method, there can be no further override of a static method derived from the calling class, because there is no class instance to contain a subclass that can further override the method. In addition to calling the nearest implementation of an overridden static method in its own class hierarchy, a class can call other implementations of the method in any other class. Unlike an instance method of a given class hierarchy, where only the most-derived implementation executes, it is possible to directly call and execute each and every implementation of an overridden static method in the same class hierarchy, as well as in other classes that derive from the current class hierarchy. For more information on defining and accessing overridden static methods, see Define static method overrides and overloadings on page 192 and Call overloaded static methods on page 190.

A class can define a method using the same name as another method defined in the class or a super class, as long as the two methods have different signatures. This new definition overloads any other method of the same name defined in the class. Thus, overloading provides a means to define several methods that perform the same function, but require different calling sequences. The only requirement is that ABL must be able to disambiguate the signatures of overloaded methods. ABL can distinguish many overloaded method signatures at compile time, but it also supports certain method overloadings that it disambiguates only at run time. Note that the STATIC option does not participate in method overloading, so you cannot define both an instance and a static method with identical signatures. For more information on method overloading, see Overload methods and constructors on page 113. For more information on how static methods participate in method overloading, see Define static members on page 190.

A method in a class can run any external procedure using the standard RUN statement. It can also run an internal procedure or user-defined function using a procedure object handle.

Methods are of course very similar to the internal procedures and user-defined functions of an ABL procedure. Methods share some of the same restrictions of internal procedures and user-defined functions. When a method has a VOID return type, the methods can contain any statements allowed within an internal procedure. When the method returns a data type the method is limited to the set of statements allowed for user-defined functions. This limitation exists because methods with a return value can be used in an expression, and certain statements are not allowed during expression evaluation.

As in an internal procedure or user-defined function, local variables defined within methods are scoped to the method. Their values do not persist across method invocations but are re-initialized on every call to the method. If a local variable in a method has the same name as a data member of property for the class, the class's data member or property is shadowed by the local variable's definition. If a local variable definition in a method shadows a data member or property of the same name in its class or a super class, there is no way for the method to access the data member or property of the class.

DEFINE METHOD statement

This is the syntax for defining a class-based method using the METHOD statement:

Syntax

```
METHOD {method-modifiers }{ VOID |return-type}method-name
      ( [parameter[ , parameter]...] ) { : | . }[method-body
END [ METHOD ] . ]
```

Element descriptions for this syntax diagram follow:

method-modifiers

A list of options that modify the behavior of the method. They can appear in any order as specified in the following syntax:

```
[ PRIVATE | PACKAGE-PRIVATE | PROTECTED  | PACKAGE-PROTECTED |PUBLIC ][
STATIC | ABSTRACT ]
[ OVERRIDE ][ FINAL ]
```

```
[ PRIVATE | PACKAGE-PRIVATE | PROTECTED | PACKAGE-PROTECTED | PUBLIC
]
```

The access mode for the method. The default method access mode is PUBLIC. PRIVATE methods are accessible from within the class where they are defined. An instance can access a PRIVATE method of another instance if they are both instances of the same class. PROTECTED methods are accessible from within the class where they are defined and in any subclass of the defining class. An instance can access a PROTECTED method of a second instance of a class that is at the same level or in a super class in the class hierarchy. A PACKAGE-PRIVATE method can be called from within the class and any class within its package. A PACKAGE-PROTECTED method can be called from within the class, any class within its package, and from within any subclass that inherits the class. You can call a PUBLIC method from within the class that defines the method, from within any class that inherits the defining class, and from any class or procedure that references the defining class. For more information on calling methods, see Call class-based methods on page 140.

An abstract method prototype cannot be defined as PRIVATE, and an interface method prototype can only be defined as PUBLIC.

```
[ STATIC ]
```

Specifies the scope of the method. By default, methods of a class are instance methods, meaning that a given method is defined and available for each instance of the class in which it is defined. However, methods can also be defined as static using this keyword. ABL defines a given static method on first reference to the class type in which it is defined, scopes that method to the class type, and makes it available through that class type for the duration of the ABL session. Unless specified otherwise, a reference to a method in this manual is assumed to be a reference to an instance method.

You cannot define a static method as abstract.

For more information on static methods, see Use static members of a class on page 186.

[ABSTRACT]

Declares the method as abstract using an abstract method prototype. Each method prototype is declared by a single METHOD statement, with a signature, but with no implementation and no END METHOD statement. Abstract methods can be defined as either PROTECTED or PUBLIC. These method prototypes cannot specify the STATIC option. An abstract method can only be an instance method. Abstract method prototypes can be overloaded and can overload implemented method definitions. Note that a constructor or destructor cannot be abstract.

An abstract method must be implemented in some class (abstract or non-abstract) that derives from the class that defines the abstract method. The class that implements this abstract method must fully define a corresponding method that matches this method prototype, but without the ABSTRACT option. In addition the implemented method can have a less restrictive access mode, if available (PUBLIC if the abstract method is PROTECTED).

You can only define an abstract method in an abstract class definition.

[OVERRIDE]

Specifies that this method overrides a method defined in a super class in the class hierarchy. The compiler verifies that this method has the same return type, the same or a less restrictive access mode, the same scope (instance or static), and the same number, modes, and data types of parameters as a method of the same name in a super class. If any of these tests fail, the compiler generates an error.

Note that to implement an inherited abstract method, a method definition overrides the abstract method. However, an abstract method prototype can also override an inherited method that is already implemented. In this case, the method is redefined as abstract and must be implemented, again, in a further derived class. This is different from the definitions of properties and events, which can never override an implemented (non-abstract) property or event, respectively.

[FINAL]

If specified, this method cannot be overridden. Any class that inherits from this class cannot define a method with the same method name as this method.

Note: The FINAL option is permitted but irrelevant on a PRIVATE method, or on any method within a class that itself has been defined as FINAL. A subclass can define a method with the same definition as a PRIVATE method in its super class. However in this case, the subclass has simply defined its own method and is not overriding the super class's method.

VOID | *return-type*

The *return-type* is the data type of the value returned by the method. The data types that can be returned by a method are the same data types supported for return values by user-defined functions: CHARACTER, CLASS, COM-HANDLE, DATE, DATETIME, DATETIME-TZ, DECIMAL, HANDLE, INT64, INTEGER, LONGCHAR, LOGICAL, MEMPTR, RAW, RECID, ROWID; where CLASS specifies an object reference to a class defined as either a class or interface type. Like user-defined function return types, the return-type can also include the EXTENT option to define a one-dimensional array of a listed type. Unlike a user-defined function, a method can return VOID, which means the method does not return a value. (In ABL, you can ignore the data type of a user-defined function simply by invoking it as a statement.)

To set the return value for a method, you must use the RETURN statement with a value of the same data type as *return-type*. There is no implicit type conversion.

method-name

The name of the method. The method name must be unique among all non-overloaded methods within the class and cannot be the same as the class name. This *method-name* must also be unique among all non-overloaded methods within the class hierarchy according to its particular class member namespace. For more information, see Namespaces for class members on page 83.

If the *method-name* is the same as the name of a non-private method in a super class that also has the same signature, the method definition **must** also include the OVERRIDE option to override the method in the super class. If the *method-name* is the same as the name of any other non-private method in the class hierarchy or any other private method in the same class definition, and the other method has a different signature, the current method definition **must not** include the OVERRIDE option, as it overloads the other method with its different signature.

Methods defined within a class can be named the same as an ABL reserved keyword. In all contexts, a method whose name is a reserved keyword must be qualified when referenced, either by an object reference (which can be THIS-OBJECT) or if it is static, with a static type name. For more information on calling methods, see Call class-based methods on page 140. For more information on using static type-name syntax to call static methods, see Access static members on page 186.

([*parameter* [, *parameter*] ...]) { : | . }

Any parameters that you define for the method. If this method overloads another method definition of the same name, ABL must be able to disambiguate the two methods by the number, data types, or modes of their parameter definitions (their signatures). For more information on the syntax of *parameter* and how to define overloaded method signatures, see the Parameter definition syntax reference entry in *ABL Reference*.

You can terminate the METHOD statement with a colon (:) or period (.). You typically use a colon to introduce the *method-body* of an implemented method and use a period to terminate the definition of an abstract or interface method prototype.

[*method-body* END [METHOD] .]

The logic of an implemented method, which can be composed of any ABL statements currently allowed within an internal procedure, plus the syntax that is restricted to classes and their methods. An implemented method must include a *method-body*, followed by an END statement, even if it contains no statements (is empty). An abstract or interface method prototype cannot specify a *method-body* at all.

One statement specifically for use in the *method-body* of an instance is a call to an instance method defined in a super class using the SUPER system reference. Also, in the *method-body* of a static method, you can **only** access other static members defined within the current class hierarchy in addition to local method data; instance members defined within the current class hierarchy are inaccessible. However, if the method is defined as an instance member, *method-body* statements can access static members as well as other instance members.

If the method has a non-VOID return type, like a user-defined function, you must return the value for any defined return type using the RETURN statement.

Like a procedure or user-defined function, you can raise ERROR on the statement that invokes the method using an UNDO, THROW available in several ABL constructs to throw an error object that can be caught and accessed in the method or in its caller. Also like a procedure, but **unlike** a

user-defined function, you can use a RETURN ERROR available in several ABL constructs to return ERROR to the caller. This RETURN ERROR can also include a string value for setting the RETURN-VALUE function in the caller or specify an error object that can be caught and accessed in the caller.

You cannot use any statements or ABL elements in the *method-body* that are only relevant in procedures, such as a reference to the THIS-PROCEDURE system handle.

Adding a method to the acme.myObjs.CustObj sample class definition results in this code:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj:

  DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.

  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD RecNum  AS INTEGER
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name    LIKE Customer.Name
    FIELD State   AS CHARACTER.

  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.

  METHOD PUBLIC CHARACTER GetCustomerName (INPUT piRecNum AS INTEGER):
      FIND ttCustomer WHERE ttCustomer.RecNum = piRecNum NO-ERROR.
      IF AVAILABLE ttCustomer THEN
         RETURN ttCustomer.Name.
      ELSE DO:
         rMsg:Alert("Customer number" + STRING(ttCustomer.RecNum) +
               "does not exist").
         RETURN ?.
      END.
  END METHOD.
END CLASS.
```

Adding an abstract method to the super class, acme.myObjs.Common.CommonObj, results in this code:

```
USING acme.myObjs.Common.*.

CLASS acme.myObjs.Common.CommonObj ABSTRACT:

  METHOD PROTECTED ABSTRACT CLASS MsgObj MessageHandler
      (INPUT iObjType AS CHARACTER).

END CLASS.
```

Implementing the abstract MessageHandler( ) method in the CustObj class results in this code:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj:

  DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.

  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD RecNum  AS INTEGER
```

```
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name    LIKE Customer.Name
    FIELD State   AS CHARACTER.

  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.

  METHOD PUBLIC CHARACTER GetCustomerName (INPUT piRecNum AS INTEGER):
    FIND ttCustomer WHERE ttCustomer.RecNum = piRecNum NO-ERROR.
    IF AVAILABLE ttCustomer THEN
      RETURN ttCustomer.Name.
    ELSE DO:
      rMsg:Alert("Customer number" + STRING(ttCustomer.RecNum) +
                 "does not exist").
      RETURN ?.
    END.
  END METHOD.

  METHOD PROTECTED OVERRIDE CLASS MsgObj MessageHandler
      (INPUT iObjType AS CHARACTER):
      RETURN NEW MsgObj (iObjType).
  END METHOD.

END CLASS.
```

Define events within a class

Class events, along with associated internal procedures and class methods that are subscribed as handlers for these events, contribute to defining the behavior of a class. A class event allows a class to provide notification of a particular condition by publishing the event when the condition is identified. Publishing the event then causes one or more event handlers that are subscribed to the event to execute. These event handlers can include any compatible method or internal procedure. Each time the event is published, the currently subscribed event handlers execute one after the other, and in no guaranteed order. After the final event handler completes, execution returns to the next statement after the statement that published the event.

A class event is a class member that you must define like any data member, property, or method. Thus, a class event can be inherited by any derived class, depending on its access mode. However, access to a class event outside of the class where it is defined only allows you to subscribe event handlers to it. You can publish a class event only from within the class where it is defined and implemented, regardless of its access mode.

When you define a class event, you also define a method signature (which is always `VOID`) for any event handlers that you subscribe to the event. For methods that you subscribe to, ABL type checks this event signature at compile time, like any method signature. However, for internal procedures that you subscribe, the AVM checks signature compatibility only at run time. Note that you pass the parameters for an event when you publish it, and the data flow through these parameters depends on the modes of the parameters (`INPUT`, `OUTPUT`, or `INPUT-OUTPUT`) and how many event handlers you have subscribed to the event. The values of `OUTPUT` and `INPUT-OUTPUT` parameters are determined by the last handler to run. In the case of `INPUT-OUTPUT` parameters, each handler invoked for the published event has access to the value set by the previous handler. For more information on how data flows through the signatures of published class events, see Publish class events.

By default, a class event is an instance member that is defined for access, according to its access mode, only within or through the instance of the class in which it is defined. A class event can also be defined as static, which associates the event with the class type (not an instance) and makes it available, according to its access mode, throughout the ABL session without the need for a class instance.

For more information on accessing static events, see 'Access static members' in *Develop Object-oriented ABL Applications*.

You can also define an abstract event prototype in an abstract class type definition, and you can define an interface event prototype in an interface type definition. An event prototype defines an event without an implementation that cannot be published. The syntax for an event prototype is largely identical to the syntax of the implemented event. However, the effect of implementing an event prototype defines the event in a class where it can be published to the application. Both types of event prototype (abstract and interface) must be implemented appropriately in the class hierarchy where they are defined. For more information on defining abstract event prototypes, see the syntax description.

For more information on defining interface event prototypes, see Define Interfaces and Use the interface construct in *Develop Object-oriented ABL Applications*.

ABL provides three built-in event methods (`Publish( )`, `Subscribe( )`, and `Unsubscribe( )`) to publish and manage event handler subscriptions for a class event.

For more information on publishing and subscribing to class events, see Publish and subscribe to class events in *Develop Object-oriented ABL Applications*.

DEFINE EVENT statement

This is the syntax for defining a class event using the `DEFINE EVENT` statement.

Syntax

```
DEFINE [event-modifiers] EVENT event-name
    {[ SIGNATURE ] VOID ( [parameter[ , parameter]...] )
    |[ DELEGATE ][ CLASS ]dotNet-delegate-type} .
```

Element descriptions for this syntax diagram follow:

event-modifiers

A list of options that modify the behavior of the event. They can appear in any order as specified in the following syntax:

```
[  PRIVATE | PROTECTED | PACKAGE-PRIVATE | PACKAGE-PROTECTED | PUBLIC ][
 STATIC | ABSTRACT ]
    [ OVERRIDE ]
```

```
[ PRIVATE | PACKAGE-PRIVATE | PROTECTED | PACKAGE-PROTECTED |PUBLIC
]
```

The access mode for the event, which determines the context where you can subscribe event handlers to the event. The default access mode is `PUBLIC`. You can subscribe an event handler to a `PRIVATE` event from within the class that defines the event. An instance can also subscribe an event handler to a `PRIVATE` event of another instance if both instances are from the same class. You can subscribe an event handler to a `PROTECTED` event from within the class that defines the event and from within any class that inherits the defining class. An instance can subscribe an event handler to a `PROTECTED` event from a second instance if the first instance is in the same class as the second instance or from a subclass of the second instance. A `PACKAGE-PRIVATE` event can have a handler subscribed to it by the defining class and any

class within its package. A PACKAGE-PROTECTED event can have a handler subscribed to it by the defining class, any class within its package, and from within any subclass that inherits the class.You can subscribe an event handler to a PUBLIC event from within the class that defines the event, from within any class that inherits the defining class, and from any class or procedure that references the defining class. For more information on subscribing event handlers to class events, see Specify handler subscriptions for class events on page 168.

An abstract event prototype cannot be defined as PRIVATE, and an interface event prototype can only be defined as PUBLIC.

[STATIC]

Specifies the scope of the event. By default, events of a class are instance events, meaning that a given event is defined and available for each instance of the class in which it is defined. However, class events can also be defined as static using this keyword. ABL defines a given static class event on first reference to the class type in which it is defined, scopes that event to the class type, and makes it available through that class type for the duration of the ABL session. Unless specified otherwise, a reference to an event in this manual is assumed to be a reference to an instance event.

You cannot define a static event as abstract.

For more information on static events, see Use static members of a class on page 186.

[ABSTRACT]

Declares the event as abstract using an abstract event prototype. Each event prototype is declared by a single DEFINE EVENT statement, with a signature. Abstract events can be defined as either PROTECTED or PUBLIC. These event prototypes cannot specify the STATIC option. An abstract event can only be an instance event.

An abstract event must be implemented in some class (abstract or non-abstract) that derives from the class that defines the abstract event. The class that inherits and implements this abstract event must define and override the event with a definition that matches this abstract event prototype, but without the ABSTRACT option. The implemented event can also have a less restrictive access mode, if available (PUBLIC if the inherited abstract method is PROTECTED). Note that you cannot use the Publish( ) event method to publish a class event that has been defined as abstract, but only from within the class definition that inherits and implements the abstract event.

Thus, defining an event as abstract ensures that the event is inherited by some other class, and that only the class that implements the inherited abstract event can publish it. An abstract event can also be inherited by another abstract class without implementing it. You can override and redefine an inherited abstract event within an abstract class without implementing it in order to specify a less restrictive access mode for the event. You can also subscribe event handlers to any accessible abstract event within or outside of the class that defines the event, because the event is always implemented by an inheriting class at run time.

You can only define an abstract event in an abstract class definition.

[OVERRIDE]

Specifies that the event definition overrides an inherited abstract event. This definition can implement the inherited abstract event, or it can redefine the event as abstract as long as the defining class is also abstract. The compiler verifies that this event definition has the same

name, the same or a less restrictive access mode, and the same number, modes, and data types for its parameters as the overridden abstract event. Event parameters can also be specified using two different syntax models, and the overriding event must use the same syntax model to specify its parameters as the inherited abstract event.

Note that you cannot override any implemented event that you inherit. Also, if the current class definition is abstract, it is not necessary to override an inherited abstract event that you want to remain abstract, unless you want to specify a less restrictive access mode to be implemented by a subclass. If any of these tests fail, the compiler generates an error.

event-name

The name of the class event. This *event-name* must be unique within the class hierarchy according to its particular class member namespace. A class event name must be unique within a namespace that includes the names of properties, variable data members, and class events. For more information, see Namespaces for class members on page 83.

Events defined within a class can be named the same as an ABL reserved keyword. In general, you only reference an event name to either publish the event or to subscribe or unsubscribe an event handler to the event. In all contexts, an event whose name is a reserved keyword must be qualified when referenced, either by an object reference (which can be `THIS-OBJECT`) or if it is static, with a static type name. For more information on publishing and managing event handler subscriptions to events, see Publish and subscribe to class events on page 167. For more information on using static type-name syntax to reference static events, see Access static members on page 186.

[SIGNATURE **]** VOID **[***parameter***[** , *parameter***]...]**

Defines a signature for the class event specified as an ABL method signature. This signature specifies how to pass parameters to event handlers when you publish the event. This signature is type checked at compile-time, like any other method signature, but the signatures of internal procedures that you subscribe as event handlers are checked for compatibility only at run time. Note that a class event signature is always `VOID`, which means that any method subscribed as an event handler must also be defined as `VOID`. For more information on the syntax of *parameter* and how to define ABL method signatures, see the Parameter definition syntax reference entry in *ABL Reference*. For more information on how data flows through the signatures of published class events, see Publish class events on page 170.

[DELEGATE **][** CLASS **]***dotNet-delegate-type*

Defines a signature for the class event specified as a .NET event signature. You must specify a .NET event signature using a .NET *delegate type* (*dotNet-delegate-type*), which is a special .NET class type that .NET uses to define the signatures of all .NET events. The primary use for defining ABL class events with .NET event signatures is to override and implement .NET abstract events or to implement events declared in .NET interfaces. However, you can also define ABL class events with .NET event signatures that have no particular connection to a .NET event. An ABL class event signature defined with a .NET delegate type is type checked at compile-time, as with any ABL class event signature, and the corresponding signatures of event handlers must be similarly run-time compatible. Note that you cannot implement an abstract event defined with an ABL method signature using a class event defined with a .NET event signature, nor the reverse, even if the signatures pass corresponding parameters of the same mode and type. For more information on defining and using ABL class events that have .NET event signatures, see the information on working with .NET events in *Use GUI for .NET Objects in ABL Applications*.

The following modified fragments from the sample classes show the addition of an `OutputGenerated` class event and its use, starting with its definition, in the `CommonObj` abstract class, as an abstract event together with an abstract method (`PublishOutputGenerated( )`) for publishing the event:

```
CLASS acme.myObjs.Common.CommonObj ABSTRACT:

...

  DEFINE PUBLIC ABSTRACT EVENT OutputGenerated
      SIGNATURE VOID (pcOutputType AS CHARACTER).

  METHOD PROTECTED ABSTRACT VOID PublishOutputGenerated
    (INPUT pcOutputType AS CHARACTER).

...

END CLASS.
```

The `CustObj` derived class implements both the event and its publishing method, which it also calls to publish the event. The listed `printObj( )` method can call `Publish( )` directly because the event is defined and implemented in the same class definition. However, another class derived from `CustObj` (`NECustObj`, not shown) must call `PublishOutputGenerated( )` in order to publish the event. So, `CustObj` also calls `PublishOutputGenerated( )` for consistency, which also helps the entire class hierarchy take advantage of any updates to how the event is published in the future:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj:

...

  DEFINE PUBLIC OVERRIDEEVENT OutputGenerated
    SIGNATURE VOID (pcOutputType AS CHARACTER).

  METHOD PROTECTED OVERRIDE VOID PublishOutputGenerated
    (INPUT pcOutputType AS CHARACTER):
    OutputGenerated:Publish(pcOutputType).
  END METHOD.

...

  METHOD PUBLIC VOID printObj ( ):
    OUTPUT TO PRINTER.
    DISPLAY dtTimestamp.
    FOR EACH ttCustomer:
      DISPLAY ttCustomer.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to printer").
  END METHOD.

...

END CLASS.
```

The Main class instantiates CustObj, subscribes its own event handler for the OutputGenerated event, and calls the printObj( ) method that publishes the event (through a call to PublishOutputGenerated( )):

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:

...

  DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.

...

  CONSTRUCTOR PUBLIC Main ( ):

...

    /* Create an instance of the CustObj class */
    rCustObj = NEW CustObj ( )

    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.

  /* Event handlers for each Customer class instance */
  METHOD PRIVATE VOID OutputGenerated_CustObjHandler
          (pcOutputType AS CHARACTER):
      MESSAGE pcOutputType "for all customers." VIEW-AS ALERT-BOX.
  END METHOD.

...

  /* ObjectInfo processes information about the Customer object */
  METHOD PUBLIC VOID ObjectInfo (piInfoCount AS INTEGER):

  ...

    IF piInfoCount <> ? AND piInfoCount > 1 THEN
      rCustObj:printObj(piInfoCount).
    ELSE
      rCustObj:printObj( ).

  ...

  END METHOD.
END CLASS.
```

Comparison with procedure-based programming named events

For procedures, the equivalent of a class event is a named event, which like all procedure resources are weakly typed. Named events have no compile-time definition and are only given a name when they are published at run time. You publish and manage event handler subscriptions for named events using the PUBLISH, SUBSCRIBE, and UNSUBSCRIBE ABL statements, which correspond in function to the Publish( ), Subscribe( ), and Unsubscribe( ) event methods for class events.

The event handlers for named events are always internal procedures (never methods). Thus, you can publish a named event from within a class as long as you publish it on behalf of an external procedure to which the event is attributed, in the same way that you can publish the event from within an external procedure that is different from the external procedure to which the event is attributed. Such a named event is associated with the specified procedure to which the event is attributed, and is not a member of the class that executes the `PUBLISH` statement. This simply allows you to publish a named event with previously developed event handlers directly from within the class without having to call a separate procedure to do it. For any new development of user-defined events using classes, class events provide the more natural mechanism to implement event-driven applications.

Namespaces for class members

ABL defines six distinct namespaces for naming class members, including methods, variables/properties/events, buffers/temp-tables, ProDataSets, queries, and data-sources.

Each class member must have a unique name within its namespace and within its class hierarchy, depending on the access mode and any overriding or overloading of the member. Without overriding or overloading, each non-private namespace member must have a unique name among all other non-private namespace members across the entire class hierarchy where it is defined. Similarly, each private namespace member must have a unique name among all namespace members within its local class definition and among all non-private namespace members defined above it in the class hierarchy. That is, private namespace member names defined higher in a class hierarchy do not affect namespace member names defined lower in the hierarchy. Namespace members that support overriding and overloading can have their names duplicated within their namespace according to the overriding or overloading that is supported for a given member type.

Note: Every temp-table has a default buffer of the same name. You cannot define a buffer of the same name as a temp-table, even within a procedure file. Because of this, the buffer and temp-table are considered to be within the same namespace.

In addition to the namespaces already described, there are two additional implicit members of a class that participate in namespaces: database tables and table (buffer) fields. Because of syntactic similarities, references to database tables and fields can conflict with type-name references to static members of a class.

Tables

Whenever an ABL file refers to a database table, an implicit buffer of the same name as the table is created. For class files this implicit buffer is logically created as a private instance member. These implicit buffers overlap the buffer/temp-table namespace.

Fields

Whenever a database table or temp-table is in scope, ABL allows unqualified access to the fields of the table, provided there is no ambiguity. For example, if the client session is connected to the `sports2020` database, the application has access to the `Customer` table fields when the table reference is in scope:

```
FIND FIRST Customer NO-LOCK.
MESSAGE Name.
```

These field references overlap the variable/property/event namespace.

Abbreviations

ABL supports abbreviated names for table and buffer fields, as in this example:

```
FIND FIRST Cust. /* Dropped the ending "omer" */
MESSAGE Tele.    /* Dropped the ending "phone" */
```

These abbreviated references overlap their respective namespaces.

Static class members

Static type-name syntax, when it includes a package name, uses the same syntax as references to tables and fields, as shown in the following table.

Table 3: Table and static type-name syntax naming convention

Tables	Static type-name syntax
`table.field`	`package.class`
`database.table.field`	`package.subpackage.class`

This causes the namespaces of static type-name references to overlap with the namespaces of tables, buffers, and fields. For more information on static type-name syntax, see Access static members on page 186.

Note: The class member namespace design was added to make it easier to migrate your application from procedural-based to object-oriented. Your application might have taken advantage of the fact that ABL allows a number of data members to be named the same. This feature allows the same capability within a user-defined class.

Define class constructors

A constructor of a class is a special method that gets called when a class is instantiated using the `NEW` function. A constructor for a class has the same name as the class name. Unlike ordinary methods, a constructor definition is identified by the `CONSTRUCTOR` statement. In addition, constructors cannot have a return type.

You are not required to define a constructor. ABL provides a default instance constructor with no parameters for any ABL class that does not explicitly define one. You can also define multiple instance constructors for a class that are overloaded with different parameter signatures. If you choose to define an instance constructor without parameters, that constructor becomes the default instance constructor for the class. You can only define one static constructor for a class. ABL also provides a default static constructor for a class if it needs one and you do not define one. For more information on static constructors, see Use static members of a class on page 186. For more information on instance constructor overloading, see Overload methods and constructors on page 113.

In a class, it is an instance constructor that executes when the class is instantiated. Remember that the main block of a class is not allowed to have any executable statements. This restriction assures that all initialization of the class is done inside its instantiating constructor. The presence of a constructor as an explicitly-defined, special-purpose method allows you to take advantage of instantiation mechanisms, such as invoking a specific chain of constructors in a class hierarchy when the class is instantiated.

The access mode for an instance constructor can be PRIVATE, PROTECTED, or PUBLIC. PUBLIC is the default. In order for a class to be instantiated using the NEW function, the class must support either an explicitly defined PUBLIC constructor or rely on the built-in default constructor. When a class or procedure creates an instance of a class with the NEW function, it is effectively telling the AVM to run a particular constructor in the other class, as specified by its unique signature in the class. (For more information, see Create a class instance on page 136.)

If no instance constructor of a class is PUBLIC, the class cannot be referenced by the NEW function outside of its own definition, or the definition of one of its subclasses, because it has not defined itself as publicly accessible. You might want to define a PROTECTED constructor if the class can only be created as part of an object hierarchy. (For an example of a PROTECTED constructor, see Construct an object on page 116.) When you define every constructor of a class as PROTECTED, that class cannot be directly instantiated by another class. It can only be instantiated by one of its subclasses. A PROTECTED constructor is appropriate where a class defines standard behavior to be inherited by one or more other classes, but where the super class, by itself, does not provide a complete class definition.

Any class constructor that is PRIVATE can only be invoked from another constructor of the defining class (using the THIS-OBJECT statement) or from an instance of the defining class. You might define a PRIVATE constructor where you want to encapsulate behavior shared and accessed only by other constructors during class instantiation. From outside the class hierarchy, you can never explicitly invoke a constructor. When you instantiate a class, a PUBLIC constructor is invoked implicitly, as specified by its signature. It is illegal to explicitly invoke a constructor from any method other than from another constructor in the same class (using the THIS-OBJECT statement) or from a constructor of a subclass (using the SUPER statement).

Like any method, you can raise ERROR on the statement that invokes a constructor. Any instance constructor that executes a RETURN ERROR or an UNDO, THROW that returns ERROR from the constructor block immediately halts class instantiation, destroys any class instances that have already completed construction in the class hierarchy, and raises the ERROR condition on the statement that instantiated the class. After the statement that instantiated the class returns control to the instantiating block, any RETURN-VALUE setting or specified error object is available for access. Constructors can also raise ERROR when the AVM cannot instantiate the class because of run-time failures such as missing class files or other broken references. For more information on handling errors returned from constructors, see Raise and handle error conditions on page 197.

CONSTRUCTOR statement

This is the syntax for defining a class constructor:

Syntax

```
CONSTRUCTOR [ PRIVATE | PROTECTED | PACKAGE-PRIVATE | PACKAGE-PROTECTED |PUBLIC |
 STATIC ]class-name
  ( [parameter[ , parameter]...] ):
  constructor-body
END [ CONSTRUCTOR ] .
```

Element descriptions for this syntax diagram follow:

[PRIVATE **|** PACKAGE-PRIVATE **|** PROTECTED **|** PACKAGE-PROTECTED **|** PUBLIC **|** STATIC **]**

Specifies the access mode for an instance constructor or STATIC to specify the static constructor. The default access mode for an instance constructor is PUBLIC. PRIVATE instance constructors can be invoked by another constructor defined in the class using the THIS-OBJECT statement. A PRIVATE constructor can also be invoked by an instance of the defining class. PROTECTED instance constructors can be invoked by another constructor defined in the class and by a constructor defined in a subclass of the defining class using the SUPER statement. A PROTECTED constructor can be invoked by an instance of the defining class or a subclass. A PACKAGE-PRIVATE constructor can be called from the defining class or any class within its package. This is done via the SUPER statement within another constructor or using the NEW statement or function elsewhere. A PACKAGE-PROTECTED constructor can be called from the defining class, from a subclass, or from any class within its package. This is done via the SUPER statement within another constructor or using the NEW statement or function elsewhere.PUBLIC constructors can be invoked by another constructor defined in the class, by a constructor defined in a subclass of the defining class, and by other classes and procedures that instantiate the defining class using the NEW function, NEW statement, or DYNAMIC-NEW statement.

You can define only one static constructor for a class type. If the class type contains other static members, this defined static constructor (or the default static constructor, if you do not define one) runs once in an ABL session at some point after the first reference to its defining class type, but before any instance constructor of that class type runs. Unless specified otherwise, a reference to a constructor in this manual is assumed to be a reference to an instance constructor. For more information on static constructors, see Use static members of a class on page 186.

class-name

The name of the class stripped of any relative path information. This is the *class-name* portion of the *class-type-name* defined by the CLASS statement.

[*parameter* **[** *, parameter* **]...]**

Any parameters that you define for an instance constructor. You cannot define parameters for a static constructor. If this instance constructor overloads another instance constructor definition in the class, ABL must be able to disambiguate the two constructors by the number, data types, or modes of their parameter definitions (their signatures). For more information on the syntax of *parameter* and how to define overloaded constructor signatures, see the "Parameter definition syntax" in *ABL Reference*.

constructor-body

The logic of the constructor, which can be composed of any statements currently allowed within a procedure block along with syntax restricted to methods. In an instance constructor only, additional statements allowed include the SUPER statement, which invokes an instance constructor defined in the immediate super class, and the THIS-OBJECT statement, which invokes another instance constructor defined in the current class. In a static constructor, you can **only** access other static members defined within the current class hierarchy in addition to local method data; instance members defined within the current class hierarchy are inaccessible. However, in an instance constructor, *constructor-body* statements can access static members as well as other instance members. In general, the constructor logic is typically used to initialize accessible data members and properties of the class.

For instance constructors only, if the constructor is for a subclass and every constructor in the immediate super class takes parameters (there is no default), the first executable statement in the constructor must be an explicit call to a constructor of the immediate super class using the SUPER

statement. If there is a super class constructor that does not take parameters (serving as a defined default), an explicit call from the subclass is optional. If you do not explicitly invoke a super class constructor, the AVM automatically invokes the super class's default constructor before executing any statements in the subclass constructor. If you invoke a `RETURN ERROR` or an `UNDO, THROW` that returns `ERROR` from a constructor, the AVM automatically calls the class destructor, which halts class instantiation and deletes any classes that have completed construction in the class hierarchy. For more information on how the AVM handles super class and subclass constructors in a class hierarchy, see Construct an object on page 116. For more information on coding the logic for a static constructor, see Define static members on page 190.

Adding a constructor to the `acme.myObjs.CustObj` sample class definition results in the following code:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj:

  DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.

  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD RecNum AS INTEGER
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name LIKE Customer.Name
    FIELD State AS CHARACTER.

  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.

  CONSTRUCTOR PUBLIC CustObj ( ):
      rCreditObj = NEW CreditObj ( ).
      iNumCusts  = 0.

      /* Fill temp table and get row count */
      FOR EACH Customer WHERE Customer.CreditLimit > 50000:
         CREATE ttCustomer.
         ASSIGN
            iNumCusts          = iNumCusts + 1
            ttCustomer.RecNum  = iNumCusts
            ttCustomer.CustNum = Customer.CustNum
            ttCustomer.Name    = Customer.Name
            ttCustomer.State   = Customer.State.
      END.
      rMsg = MessageHandler("acme.myObjs.CustObj").
  END CONSTRUCTOR.

  METHOD PUBLIC CHARACTER GetCustomerName (INPUT piRecNum AS INTEGER):
    FIND ttCustomer WHERE ttCustomer.RecNum = piRecNum NO-ERROR.
    IF AVAILABLE ttCustomer THEN
      RETURN ttCustomer.Name.
    ELSE DO:
      rMsg:Alert("Customer number" + STRING(ttCustomer.RecNum) +
                 "does not exist").
      RETURN ?.
    END.
  END METHOD.

END CLASS.
```

Comparison with procedure-based programming main blocks of persistent procedures

An instance constructor for a class is similar to the main block of a persistent procedure. Parameters passed to a class's constructor are similar to parameters passed to a persistent procedure when it is run. When you instantiate a class using the NEW function, the specified constructor executes as part of the class constructor hierarchy and returns to the caller with the object reference to the instantiated class. The caller can then use this object reference to access public data members and properties and to call public methods on this class-based object. When a procedure is first run PERSISTENT, the code in its main block executes and returns to the caller, setting the procedure object handle. The caller can then access any variables that it shares globally with the procedure object and can use the procedure object handle to run internal procedures and user-defined functions in the procedure object.

Define the class destructor

The destructor of a class is a special method that is called automatically by the AVM when a class instance is deleted by garbage collection, by a DELETE OBJECT statement, or class instantiation is halted by a RETURN ERROR or UNDO, THROW. Thus, you never call a destructor directly. The destructor of a class has the same name as the class name. Unlike ordinary methods, a destructor definition is identified by the DESTRUCTOR statement. Destructors have no return type, are always PUBLIC, and cannot have any parameters. Destructors also are instance members. Note that there is no static destructor.

A class is not required to have a destructor. If a class has not defined a destructor for the class, ABL provides a default destructor for the class.

For each class instance that you delete, its destructor is responsible for deleting any resources allocated during the execution of that class instance. Deleting the class instance automatically deletes all dynamic handle-based objects that are created in any default unnamed widget pool that is specified for the class. (For more information, see Use widget pools on page 223.) Otherwise, you can code the destructor to explicitly delete resources that have otherwise been created by the class instance.

The DELETE OBJECT statement can already be used to delete dynamic visual and data objects as well as persistent procedure instances. What is different about classes with a destructor is that a class is given an opportunity to do necessary cleanup work when it is deleted. The value of the destructor is that it is executed automatically when the class instance is terminated using the DELETE OBJECT statement or as a result of garbage collection.

At run time, the AVM frees all allocated memory associated with the object reference when it executes the DELETE OBJECT statement. Before doing this, the AVM invokes the destructor for each class in the object's class hierarchy, if one has been defined.

When the client session is shut down, the AVM deletes all remaining class instances, invokes the destructor for each class in the object's class hierarchy, and frees all resources associated with them. In addition, the AVM automatically deletes any ABL-related resources, such as persistent procedure, visual objects, and data objects.

If a destructor fails to clean up the resources that were allocated for the object, there is no mechanism for it to inform the caller (for example, a DELETE OBJECT statement) that it failed. Also, you can use a a RETURN ERROR or an UNDO, THROW in a destructor, but it is illegal to return an ERROR condition from the destructor to the caller.

DESTRUCTOR statement

This is the syntax for defining a class destructor:

Syntax

```
DESTRUCTOR [ PUBLIC ] class-name ( ):
  destructor-body
END [ DESTRUCTOR ] .
```

Element descriptions for this syntax diagram follow:

class-name

The name of the class stripped of any relative path information. This is the *class-name* portion of the *class-type-name* defined by the CLASS statement.

destructor-body

The logic of the destructor, which can be composed of any ABL statements allowed within a method block, except that any RETURN ERROR or UNDO, THROW cannot raise ERROR beyond the level of the destructor block. The destructor logic is normally used to free up any dynamic handle-based objects or other system resources in use by the class.

Adding a destructor to the sample class definition results in the following code:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj:

  DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.

  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD RecNum  AS INTEGER
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name    LIKE Customer.Name
    FIELD State   AS CHARACTER.

  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.

  CONSTRUCTOR PUBLIC CustObj ( ):
    rCreditObj = NEW CreditObj ( ).
    iNumCusts = 0.

    /* Fill temp table and get row count */
    FOR EACH Customer NO-LOCK WHERE Customer.CreditLimit > 50000:
      CREATE ttCustomer.
       ASSIGN
        iNumCusts          = iNumCusts + 1
        ttCustomer.RecNum  = iNumCusts
        ttCustomer.CustNum = Customer.CustNum
        ttCustomer.Name    = Customer.Name
        ttCustomer.State   = Customer.State.
    END.
    rMsg = MessageHandler("acme.myObjs.CustObj").
  END CONSTRUCTOR.

  ...

  DESTRUCTOR PUBLIC CustObj( ):
     EMPTY TEMP-TABLE ttCustomer.
  END DESTRUCTOR.

END CLASS.
```

In this definition, the clean-up work of the class destructor includes emptying a temp-table of all the records that the class creates for it. Note that all the other class instances that this class creates are automatically deleted through garbage collection. For more information on ABL garbage collection, see Garbage collection on page 98.

Comparison with procedure-based programming cleanup

With persistent procedures, the application must adhere to an enforced programming strategy in order to allow cleanup when the procedure exits. Typically, you do this by defining a trigger block to be executed `ON CLOSE OF THIS-PROCEDURE`, then take care to always delete a persistent procedure instance using the `APPLY "CLOSE"` statement rather than using the `DELETE OBJECT` statement. (This is the convention used in procedures generated by the AppBuilder.)

A destructor provides a uniform mechanism to handle such cleanup tasks in class-based objects without the need for special programming strategies.

Define class-scoped handle-based objects

Handle-based objects that include static widgets (visual objects), streams, and work-tables can only be defined in a class definition as PRIVATE and can only be referenced within the class file that defines them. Therefore, they are not members of the class in which they are defined, but are resources that can be referenced by other executable class elements, including properties, methods, and triggers.

This is the syntax for defining class-scoped handle-based objects:

Syntax

```
DEFINE [ PRIVATE ]
  { BROWSE | BUTTON | FRAME | IMAGE | MENU |
    RECTANGLE | STREAM | SUB-MENU | WORK-TABLE }
  object-name[object-definition] .
```

object-name

The name of the static widget, stream, or work-table.

The *object-name* must be unique among all handle-based objects of the same type in the defining class. This name cannot be an ABL reserved keyword.

object-definition

The remaining syntax available for defining the static widget, stream, or work-table.

The use of these objects within a class definition is much the same as their use within a procedure. For more information on using widgets within a class definition, see Define and use widgets in classes on page 194.

Use the root class: Progress.Lang.Object

Progress.Lang.Object is an ABL built-in class definition that is the ultimate super class (root class) for all classes. In other words, if a class does not inherit from another class it implicitly inherits from Progress.Lang.Object. Thus, at the top of every class inheritance chain is the Progress.Lang.Object class.

Progress.Lang.Object is a fully implemented (non-abstract) class that provides a common set of properties and methods that are available for an instance of any class.

Note: These properties and methods provide support for class-based objects that is similar to the support provided by the ABL built-in attributes and methods on procedure object handles.

The following table describes the common properties and methods on Progress.Lang.Object.

Table 4: Progress.Lang.Object public properties and methods

Property or method	Description
`PUBLIC NEXT-SIBLING AS CLASS Progress.Lang.Object`	A read-only property that contains an object reference to the next object in the linked list of instantiated objects for the session. You use this property together with the `FIRST-OBJECT` attribute on the `SESSION` system handle in order to walk the list from the first to the last instantiated object.
`PUBLIC PREV-SIBLING AS CLASS Progress.Lang.Object`	A read-only property that contains an object reference to the previous object in the linked list of instantiated objects for the session. You use this property together with the `LAST-OBJECT` attribute on the `SESSION` system handle in order to walk the list from the last to the first instantiated object.
`CONSTRUCTOR PUBLIC Object ( )`	Constructor for this class. You can instantiate `Progress.Lang.Object`, but its functionality is limited to the public properties and methods defined in this table. This class is normally used to define a variable or parameter that can represent any class-based object in ABL (because all class-based objects inherit from `Progress.Lang.Object`).
`METHOD PUBLIC CLASS Progress.Lang.Class GetClass ( )`	Returns an object reference to a built-in instance of `Progress.Lang.Class` that provides type information for the current class instance. Each ABL session contains a single instance of `Progress.Lang.Class` for each type of class-based object created in the session. The lifetime of this built-in object is controlled by the ABL session and therefore cannot be deleted. For more information on the built-in class, `Progress.Lang.Class`, see Reflection on page 205.
`METHOD PUBLIC CHARACTER ToString ( )`	Returns the type name of the object followed by a unique object identifier in the ABL session. This method is normally overridden by a subclass.
`METHOD PUBLIC LOGICAL Equals ( other-obj AS CLASS Progress.Lang.Object )`	Returns `TRUE` if two different object references point to the same object or are both the `Unknown value (?)`, and returns `FALSE` otherwise. The two compared object references include the current object reference and the object reference specified by *`other-obj`*. **Note:** This default behavior is equivalent to comparing two object references using the equals operator (`EQ` or `=`).
`METHOD PUBLIC CLASS Progress.Lang.Object Clone ( )`	Has no default behavior and returns the `Unknown value (?)` and an error message if it is invoked and not overridden by a subclass. The subclass should define a method to create a copy of an object and return a reference to the new copy.

Because every object ultimately derives from `Progress.Lang.Object`, you can define an object reference data element with the data type `Progress.Lang.Object` and set it to reference any class-based object that you create in an ABL session. As with any object reference, the use of a `Progress.Lang.Object` object reference is limited to the properties and methods defined for `Progress.Lang.Object`. However, you can cast any object reference to an appropriate subclass to access functionality in that subclass. For more information on casting, see Object reference assignment and casting on page 179.

Define interfaces

An interface declares prototypes for a set of instance properties, methods, and events that a class must define if the class definition implements the specified interface. An interface also represents a user-defined object type that defines only property, method, and event names and signatures, along with any temp-table and ProDataSet definitions used as parameters to those properties, methods, and events.

A class can implement zero or more interfaces. Thus, interfaces provide a way for a class to satisfy the requirements of more than one set of behaviors while only being able to inherit a single set of behaviors from its super class. If some behaviors represent a set of properties, methods, or events that other objects need to invoke or access, but for which there is no common implementation, an interface provides a reliable way of defining a uniform contract that all implementations of its properties, methods, and events must satisfy. Any caller can then count on the consistent definition of a set of properties, methods, and events and their signatures, even though the implementation of these same properties, methods, and events might differ completely from one class to another.

Note that an abstract class provides a similar mechanism for defining a contract that an implementing class must satisfy. However, an abstract class enforces its contract through inheritance by a subclass. Unlike an interface, an abstract class can define implemented members that it also provides to the inheriting class hierarchy. This allows an abstract class to enforce a common implementation among class hierarchies in which it participates, in addition to any abstract members that it defines. The class that inherits from an abstract class can also implement an interface that declares prototypes for both the abstract and implemented class members that it inherits.

Use the INTERFACE construct

The `INTERFACE` construct represents all of the statements that can comprise an interface definition. The interface definition specifies a main block, starting with an `INTERFACE` statement that identifies the interface. After any `USING` statements, the first compilable line in a class definition file that defines an interface must be this `INTERFACE` statement. The last statement in an interface definition must be the `END INTERFACE` statement, which terminates the main block of the interface. An interface compiles to r-code similar to a class.

Note: The `INTERFACE` statement is only valid in a file with the `.cls` extension.

The main block of the interface contains statements that specify the property, method, and event names and signatures that any class implementing the interface must define. Interfaces do not specify data members to implement, but can contain temp-table and ProDataSet definitions that are used for parameters in specified property, method, or event prototypes. No allocation is associated with these definitions. These temp-table and ProDataSet definitions must occur before any property, method, or event prototypes in the main block of the interface are specified. Any class implementing an interface with temp-table or ProDataSet parameters must repeat those definitions within the class.

When compiling a class that implements an interface with temp-table or ProDataSet parameters, the definition of a given temp-table or ProDataSet in the implementing class must be compatible with the definition of the corresponding temp-table or ProDataSet in the interface. The compiler's compatibility rules are the same as those enforced when passing temp-table or ProDataSet parameters:

- Compatible temp-tables must have the same number of fields, with each field matching in data type, extent, and position, but not in field name or any other field attribute. The temp-tables must have the same number of indexes, with index components matching in every way (except in field name), and the index names must also match. The temp-table names do not have to match.
- Compatible ProDataSets must have the same number of member buffers, in the same order, and the tables of those buffers must match in the same way as compatible temp-tables. Names of either the ProDataSet or its buffers do not have to match.

INTERFACE statement

This is the syntax for defining an interface:

Syntax

```
INTERFACE interface-type-name :
  [ INHERITS super-interface-name[ , super-interface-name]... ] :
  [{temp-table|ProDataSet}...]
  [property...]
  [method...]
  [event...]
END [ INTERFACE ] .
```

Element descriptions for this syntax diagram follow:

interface-type-name

The object type name for the interface definition is specified using the following syntax:

```
[ " ][package.]interface-name [ " ]
```

package

A period-separated list of string components that comprise a package name that, along with *interface-name*, uniquely identifies the interface among all accessible classes and interfaces in your application environment.

Note that you cannot specify `Progress` as the first component of the package name for any ABL user-defined interface. For example, `Progress.Inventory` is an invalid package name for a user-defined interface and results in a compiler error.

interface-name

The interface name, which must match the filename of the class file containing the interface definition.

For more information on this syntax, see Define and reference object type names on page 45.

For an interface definition, if the interface is defined in a package, you must specify *package*, even if the class definition file contains a USING statement that specifies the fully qualified object type name for the interface or its *package*. For more information, see Reference an object type name without its package on page 48.

INHERITS *super-interface-name* [, *super-interface-name*] ...

One or more interfaces that the definition for *interface-type-name* inherits. The property, method, and event prototypes defined by each *super-interface-name* thus become part of this interface definition, and multiple occurrences of the same *super-interface-name* or member prototype in the interface hierarchy are treated as a single occurrence. However, you can never specify *interface-type-name* as an inherited interface. In other words, there can be no cycle in an interface inheritance hierarchy (recursive inheritance). Otherwise, ABL raises a compile-time error.

The syntax for *super-interface-name* is the same as for *interface-type-name*. However, if a USING statement specifies a fully qualified *super-interface-name* or its package, you can specify the super interface using its unqualified interface name. For more information, see Reference an object type name without its package on page 48. Based on this information, ABL locates each specified super interface definition file (either the source file or the r-code file) in order to compile and run with this user-defined interface.

For more information on using interface inheritance, see Interface hierarchies and inheritance on page 121.

temp-table | *ProDataSet*

A static temp-table or ProDataSet definition. The interface must supply temp-table and ProDataSet definitions for any temp-tables and ProDataSets used as parameters to methods defined for the interface. These are data definitions only. Such temp-table or ProDataSet definitions cannot specify the STATIC option. The class that implements this interface must define an instance data member that matches each data definition in the interface. These temp-table or ProDataSet definitions cannot be inherited from a super interface; they are private to and apply only to the members of the interface in which they are defined.

property

A property interface prototype. Each property prototype is declared by a single DEFINE PROPERTY statement, with one or both of the GET and SET accessors specified, but with no accessor implementations. This means that appropriate access mode and data type information is specified for each accessor, but the accessor code block must be empty. All interface properties must be defined as PUBLIC, either explicitly declared or by default. These property prototypes cannot specify the INITIAL, STATIC, ABSTRACT, or OVERRIDE options. A property prototype has no storage allocated for it to hold an initial value and it can only be implemented as an instance property.

The class that implements this interface must fully define a corresponding property that matches each property prototype in the interface. It must at least define the accessors specified in the interface prototype, but it can also define any accessor that is not specified in the prototype. For more information on defining properties, see Define properties within a class on page 62.

method

A method interface prototype. Each method prototype is declared by a single METHOD statement with a signature, but with no implementation and no END METHOD statement. All interface methods must be defined as PUBLIC, either explicitly declared or by default. These method prototypes cannot

specify the STATIC, ABSTRACT, or OVERRIDE options. A method prototype can only be implemented as an instance method. Method prototypes can be overloaded and can overload implemented method definitions. Note that interfaces cannot contain a constructor or destructor.

The class that implements this interface must fully define a corresponding method that matches each method prototype in the interface. For more information on defining methods, see Define methods within a class on page 71.

event

A class event interface prototype. Each event prototype is declared by a single DEFINE EVENT statement with a signature. All interface events must be defined as PUBLIC, either explicitly declared or by default. These event prototypes cannot specify the STATIC, ABSTRACT, or OVERRIDE options. An event prototype can only be implemented as an instance event.

The class that implements this interface must fully define a corresponding event that matches each event prototype in the interface. Note that there is little difference between an interface event prototype and the event definition that implements it. However, as for any interface member, an event interface prototype ensures that a corresponding event member is defined in the class that implements the interface. Also note that while the syntax of the event implementation is no different from the interface event prototype, only the class definition that implements an event can publish the event to the application. For more information on defining events, see Define events within a class on page 77.

Use an interface definition

Interfaces cannot be instantiated, but they can be used as the data type to define an object reference data element, such as in a VAR and DEFINE VARIABLE statement or a DEFINE PARAMETER statement. This data element can then be assigned to hold an object reference to an instance of a class that implements the interface. Multiple classes can implement the same interface, which allows the classes to be treated in a common manner. An object reference to an interface can then be used to call these common methods, even though the underlying object might be of different classes. This is similar to having multiple classes inheriting from the same super class (with the exception that an interface itself provides no method implementation). Thus, an object reference to a super class can also be used to call the methods that are defined in the super class, even though the underlying object might be of different subclasses with varying method implementations.

The following example shows a simple interface definition, which is modified from the sample acme.myObjs.Interfaces.IBusObj interface with an additional Name property:

```
INTERFACE acme.myObjs.Interfaces.IBusObj:

  METHOD PUBLIC VOID printObj ( ).
  METHOD PUBLIC VOID printObj (INPUT pcCopies AS INTEGER).
  METHOD PUBLIC VOID logObj (INPUT pcFilename AS CHARACTER).

  DEFINE PUBLIC PROPERTY Name AS CHARACTER NO-UNDO
    GET.
    SET.

END INTERFACE.
```

Note that this interface declares two overloadings of the printObj( ) method.

The following fragment from the `acme.myObjs.CustObj` sample class shows how it might implement this interface, extended with the additional `Name` property:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj
                         IMPLEMENTS IBusObj:
  ...

  /* Must implement methods defined in the IBusObj interface */
  . . .
  /* First version of printObj prints a single copy of a report */
  METHOD PUBLIC VOID printObj( ):
      OUTPUT TO PRINTER.
      DISPLAY dtTimestamp.
      FOR EACH ttCustomer:
          DISPLAY ttCustomer.
      END.
      OUTPUT CLOSE.
      PublishOutputGenerated("One copy of report sent to printer").
  END METHOD.

/* Second version of printObj takes an integer parameter representing the
   number of copies to print. */
  METHOD PUBLIC VOID printObj (INPUT piCopies AS INTEGER):
    VAR INT iCnt.
    OUTPUT TO PRINTER.
    IF piCopies <> 0 THEN
    DO iCnt = 1 TO ABS(piCopies):
      DISPLAY dtTimestamp.
      FOR EACH ttCustomer:
        DISPLAY ttCustomer.
      END.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated(STRING(piCopies)
                           + " copies of report sent to printer").
  END METHOD.

  /* Method to log customer information */
  METHOD PUBLIC VOID logObj (INPUT pcFilename AS CHARACTER):
      OUTPUT TO VALUE(pcFilename).
      DISPLAY dtTimestamp.
      FOR EACH ttCustomer:
          DISPLAY ttCustomer.
      END.
      OUTPUT CLOSE.
      PublishOutputGenerated("One copy of report sent to "
                             + pcFilename + " file").
  END METHOD.

  DEFINE PUBLIC PROPERTY Name AS CHARACTER NO-UNDO
    GET():
      MESSAGE "Name property GET accessor" VIEW-AS ALERT-BOX.
      RETURN Name.
    END GET.
    SET(INPUT cValue AS CHARACTER):
      MESSAGE "Name property SET accessor: value" cValue VIEW-AS ALERT-BOX.
      Name = cValue.
    END SET.

  ...

END CLASS.
```

Note that while a property interface declaration may include a GET and SET accessor, or a GET accessor, or a SET accessor, you cannot force the property to **not** have a particular accessor in an implementing class. However, you can force the accessor to be missing when an instance of the class is used through an interface reference. You can do this by omitting the accessor in the property interface definition. Even though the accessor might be implemented in the class property, when used through an interface reference it will appear that the accessor implementation does not exist.

Garbage collection

Any class-based object, which ultimately derives from `Progress.Lang.Object`, is automatically 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 the 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 pass it to another procedure, where you can continue to manage the object until you finally delete it or it is deleted by garbage collection.

You can assign an object reference to an ABL data element in one of the following ways:

- When you assign the object reference returned by a `NEW` function (in the context of a `NEW` statement) to an ABL data element that is defined as a compatible object type
- When you pass an invocation of the `NEW` function as an argument to a routine `INPUT` parameter defined as a compatible object type

If you invoke the `NEW` function as part of an expression (**not** in a `NEW` statement) in which the resulting object reference is never assigned to an ABL data element, the AVM automatically handles the deletion of this object reference when it is no longer in use (sometime after the statement where the expression appears completes execution). Such an object reference is thus consumed by the expression and is no longer accessible to your application.

For more information on obtaining object references from the `NEW` function and the `NEW` statement, see Create a class instance on page 136.

To disable automatic garbage collection of class-based objects, use the No Garbage Collection (`-nogc`) client startup parameter. By default, the AVM automatically deletes class-based objects when there are no remaining references to the object. When `-nogc` is specified, the application assumes the responsibility of deleting these objects explicitly, using the `DELETE OBJECT` statement. If `-nogc` is specified, ABL objects that are not explicitly deleted are deleted when the session ends, and the AVM does not invoke any destructors for the class. However, `-nogc` does not prevent .NET garbage collection.

Garbage collection does not apply to handle-based objects.

An object may be deleted at any time using the `DELETE OBJECT` statement and the presence of outstanding references will not inhibit the destruction of the object. If you want to ensure that an object is deleted immediately, without waiting for garbage collection to occur, then use the DELETE OBJECT statement on page 98.

DELETE OBJECT statement

This is the syntax for deleting an instance of a class:

Syntax

```
DELETE OBJECT object-reference [ NO-ERROR ] .
```

object-reference

A data element containing a reference to an instantiated object.

For example, to delete the instance of the sample class, execute the DELETE OBJECT statement, as shown:

```
DELETE OBJECT myCustObj.
```

Note: If you use DELETE OBJECT on an object reference but there is still another reference to that object, the other reference will become invalid. Therefore, unless you are sure that the object is no longer needed, it is best to allow garbage collection to delete class-based objects. You just need to be sure that any reference no longer needed is set to the Unknown value (?), or will naturally go out of scope. There is one case where garbage collection will not occur and that is when there is a circular reference (for example, object A has a reference to object B and object B has a reference to object A). In this case, a link in the chain needs to be broken in order for garbage collection to free the objects involved. To break the chain, set one of the references to the Unknown value (?) or use DELETE OBJECT on the reference. But again, this is not appropriate if there are other references that need to remain valid.

For more information on deleting class-based objects, see Delete an object on page 121.

Note: Within a constructor, you can also use DELETE OBJECT THIS-OBJECT to abort class instantiation, which sets the returned object reference to the Unknown value (?). However, Progress Software Corporation recommends that you use RETURN ERROR or UNDO, THROW to return ERROR from the constructor. This both aborts class instantiation and raises an ERROR condition that you can handle in the instantiating context. For more information on class instantiation, see Construct an object on page 116. For more information on raising ERROR within a constructor, see Raise errors within an instance constructor on page 203.

During garbage collection and when a DELETE OBJECT statement executes, the AVM frees all allocated memory associated with the object reference and invokes the destructor for each class in the object's class hierarchy, if one has been defined. An application can use the destructor for a class to release any resources that it has acquired during the execution of the object instance.

OpenEdge includes a performance tuning feature for ABL class-based applications that controls how the AVM deletes objects. The Re-usable Objects Cache (-reusableObjects) startup parameter specifies the number of deleted class objects that the AVM stores for later re-initialization. By default, -reusableObjects is set to 25. When you use -reusableObjects, the AVM transfers the deleted object for most ABL classes to a re-usable objects cache. If your application causes the AVM to instantiate the same class later, the stored object is re-initialized and removed from the cache. The re-initialized object has a new UUID and the same initial data as a new instance of the class. The re-use of the object saves much of the overhead of instantiating a class.

Note: The cache does not store .NET classes, .NET-derived ABL classes, classes with static elements, or classes compiled during your session.

For more information on the re-usable objects cache, see the DELETE OBJECT statement in *ABL Reference* and the Re-usable Objects Cache (`-reusableObjects`) startup parameter in *Startup Command and Parameter Reference.*

When the client session is shut down, the AVM deletes all remaining class instances, invokes the destructor for each class in the object's class hierarchy (unless you specified `-nogc`), empties the re-usable object cache, and frees all resources associated with the classes.

Note: Deleting an instance of a class has no effect on any static members that have been initialized for the class type. static members of a class persist for the duration of an ABL session. For more information, see Initialize and delete static members on page 192.

3

Design objects: inheritance, polymorphism, and delegation

Other topics introduced the basic syntax that supports classes in ABL, this content provides more conceptual information on the structure and operation of class hierarchies, and shows how you might use classes in an application.

For details, see the following topics:

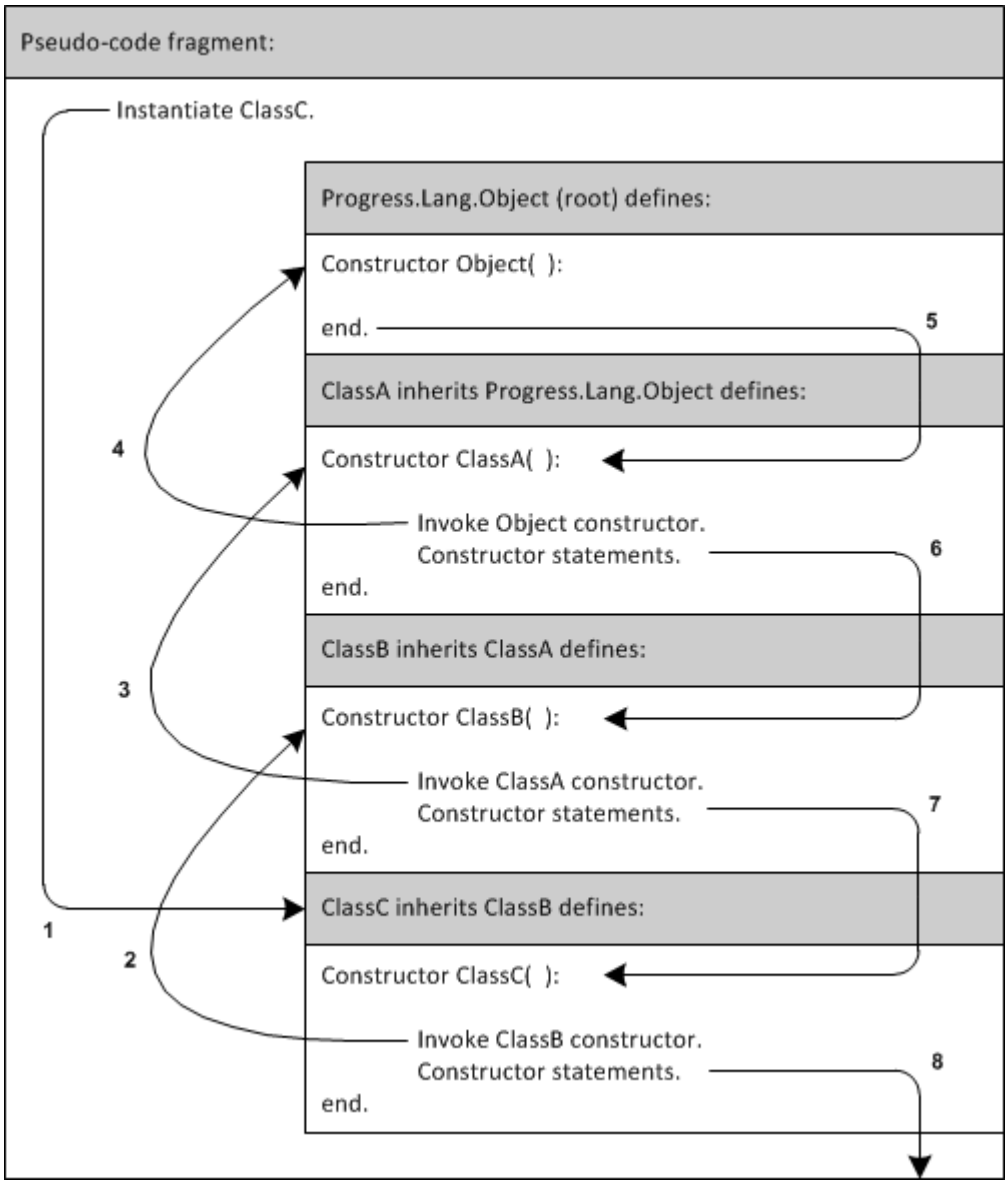
- Class hierarchies and inheritance
- Interface hierarchies and inheritance
- Use polymorphism with classes
- Use delegation with classes

Class hierarchies and inheritance

One of the great powers of object-oriented programming is inheritance, which facilitates code reuse and polymorphism (see Polymorphism on page 21). The CLASS construct supports single inheritance of classes. This inheritance allows a class (as a derived class or subclass) to extend another class (its super class). The subclass inherits all the non-private data members, properties, methods, and events of the super class so they appear as if they are part of the subclass. The super class itself might be a derived class that extends its own super class, forming a class hierarchy from the resulting series of super class and subclass relationships. Thus, through this chain of inheritance, a derived class inherits data and behavior from all the super classes in its class hierarchy, including the root class, the super class at the top of the hierarchy. The root class of a hierarchy is the class that does not inherit from any other class. In ABL, the root class of all classes is the built-in, non-abstract class, `Progress.Lang.Object`.

The following figure shows the sequence of class construction and inheritance during instantiation of a class hierarchy.

Figure 6: Instantiation of a class hierarchy



The numbered arrows show the order in which execution occurs during construction of the class hierarchy.

So, an instantiated class (an object) actually represents a hierarchy of classes, with the root class (`Progress.Lang.Object`) at the top and the instantiated class as the most derived class at the bottom (ClassC). When a class is instantiated, the specified constructor for each class in its class hierarchy is executed as part of the instantiation process. The AVM first invokes the constructor in the instantiated class (ClassC) of the object. The first action of the instantiated class's constructor must be to invoke a constructor in its immediate super class (ClassB), whether implicitly (for the default constructor) or explicitly (especially if parameters must be passed).

This super class constructor, in turn, invokes a constructor in its own immediate super class (ClassA). This chain of super class constructor invocation continues to the root class (`Progress.Lang.Object`). When the root class's constructor is invoked, it executes to completion. Execution then returns to its caller, its immediate subclass (ClassA), so that the subclass constructor can complete. This sequence continues until the constructor in the most derived subclass (the originally instantiated class, ClassC) completes. In this way, although instantiation of a class always starts at the bottom of its class hierarchy, the object representing this hierarchy is constructed starting from the root class at the top.

Also, note that one or both of ClassA and ClassB can be either abstract or non-abstract classes. So, in an ABL class hierarchy, only the instantiated class (ClassC in the figure) **must** be fully implemented (non-abstract).

Note: The invocation of static constructors in a class hierarchy follows a different initialization sequence. For more information, see Use static members of a class on page 186.

Classes and strong typing

Classes in ABL are strongly-typed. Strong typing means that the class definition and any references to the class are checked for validity at both compile time and run time. As described in Object-oriented programming and ABL on page 9, compile-time validation is one of the key advantages of programming with classes. Note also, that the data type of an object can be represented as any class type (abstract or non-abstract) in its class hierarchy, and as any interface type that classes in the hierarchy implement. This type representation determines how the object can be used.

In order to support strong typing, either the source files or r-code for all classes in an application must be appropriately available at compile time. Thus, when you are compiling a class that extends a class hierarchy, all the classes and interfaces for the hierarchy must be available either as source files or as r-code files.

In addition, the compilation of a subclass forces the recompilation of any class or interface in its class hierarchy. Since a class is built from all of the classes and interfaces in its hierarchy it is important to make sure that the entire hierarchy is up to date. If any class or interface in the hierarchy cannot be compiled, the compilation of the subclass fails.

Caution: When you compile a class, although ABL automatically compiles all super classes in its class hierarchy, it does not know about and cannot automatically compile any of the subclasses of the class you are compiling. Thus, when you change a super class, you must compile the class and all subclasses of the class to ensure that all objects with this class in their hierarchy inherit the updated class members. ABL, by itself, has no knowledge of these subclasses, so you must keep track of them manually or by using configuration management tools.

The compiler can handle compilation of a set of classes in any order as long as either the source or r-code can be found for all referenced classes. The compilation of a class can cause many other class files to be examined, and the compilation of a procedure that uses classes can cause those class files to be examined as well.

For more information on class compilation, see Compile class definition files on page 248.

Class hierarchies and procedure hierarchies

A class can inherit from (extend) only one other class. However, a class can be extended by any number of other classes (see). A class hierarchy represents all the classes that a class inherits either directly or indirectly through multiple levels of inheritance. The root of a class hierarchy is a class that does not extend any other class. The entire class hierarchy is treated as a single object type. You can logically think of a class as the merging of its own members with all of the non-private members of all the classes above it in the hierarchy, including the members of the built-in root class, `Progress.Lang.Object`.

To illustrate this, consider the hierarchy for the sample user-defined classes (see Sample classes on page 225), where `acme.myObjs.Common.CommonObj` is the top user-defined super class (base class), `acme.myObjs.CustObj` inherits from `acme.myObjs.Common.CommonObj`, and `acme.myObjs.NECustObj` inherits from `acme.myObjs.CustObj`, represented as follows:

```
Progress.Lang.Object              <--- Top of hierarchy
  acme.myObjs.Common.CommonObj
    acme.myObjs.CustObj
      acme.myObjs.NECustObj       <--- Bottom of hierarchy
```

A class definition can override a non-private method in its class hierarchy by providing a method definition with the same name, return type, number of parameters, and corresponding parameter types as the method in its class hierarchy. For example the sample `acme.myObjs.CustObj` class has a method `GetCustomerName( )` that returns the name of a customer, and the `acme.myObjs.NECustObj` class overrides the `GetCustomerName( )` method to provide both the name and E-mail address of the customer. In addition, any non-private data is available to all subclasses of the super class that defines it.

Also, the acme.myObjs.Common.CommonObj class is abstract and all of its abstract members (which can include properties, methods, or class events) must be overridden and implemented by the first derived non-abstract class, in this case `acme.myObjs.CustObj`. For example, `acme.myObjs.CustObj` overrides and implements three abstract members, including the `OutputGenerated` event and the two methods, `PublishOutputGenerated( )` and `MessageHandler( )`.

Finally, the data type of an object can be referenced as any class or interface type that is part of its class hierarchy. For example, an object instantiated as the `acme.myObjs.NECustObj` class can be referenced as the `acme.myObjs.Common.CommonObj` class. However, if it is so referenced, only the accessible class members defined within `CommonObj` or its inherited class hierarchy can be referenced using the object. Note that if `CommonObj` is an abstract class (as in the sample classes), you can also reference any of its accessible abstract members, because they are guaranteed to be implemented in some derived class.

Comparison with procedure-based programming super procedures

There are parallels between a class hierarchy and the super procedure mechanism in ABL. Super procedures provide some aspects of inheritance and delegation. For example, if the classes `acme.myObjs.Common.CommonObj`, `acme.myObjs.CustObj`, and `acme.myObjs.NECustObj` were implemented as ABL procedures, then the startup code for `NECustObj.p` could run `CustObj.p` and `CommonObj.p` persistently, and use the `ADD-SUPER-PROCEDURE` handle-based method to create a super procedure chain linking the three running procedure instances. The procedures could implement some of the same internal procedures or functions that correspond to the original class methods, and pass control from one level to the next higher level using the `RUN SUPER` statement or the `SUPER` built-in function. As described in a previous section, a key aspect of this technique is that all the associations are established at run time, so that the ABL compiler has no opportunity to validate the correctness of references between the procedures, as it can with the corresponding classes. Of course, without strong typing, procedures cannot be defined as abstract or implement interfaces. So, the typical way to manage different implementations of the same code base for procedures is to compile them with different versions of include (`.i`) files.

Class inheritance provides this functionality through more conventional object-oriented syntax, in a framework of strong typing and compile-time validation. Using one technique or the other is a choice that you have to make when you begin the design of an application module that is made up of multiple related procedures or classes. Whatever your choice, you cannot use the super procedure mechanism in classes, and you cannot build a class hierarchy from procedures.

For a comparison of class-based and procedure objects separately implemented using corresponding classes and procedures, see Compare constructs in classes and procedures on page 225.

Method scoping within a class hierarchy

A PRIVATE method can be called by the defining class. An instance can call the PRIVATE method of another instance if they are both from the same class. A PROTECTED method can be called by the defining class and any of its derived classes. A PACKAGE-PRIVATE method can be called from within the class and any class within its package. A PACKAGE-PROTECTED method can be called from within the class, any class within its package, and from within any subclass that inherits the class. A PUBLIC method can be called from any code (in a class or a procedure) that has access to a class instance. The default access mode is PUBLIC. When declaring an interface method prototype, the access mode for this method must be PUBLIC (the default). When defining an abstract method, the access mode for the method cannot be PRIVATE.

Existing A class can access all of the PUBLIC and PROTECTED methods of its super class as well as any methods that it defines. A super class that, in turn, inherits from another class inherits all its super class's PUBLIC and PROTECTED methods and so on up to the root of the hierarchy. Therefore when a class inherits all of the PUBLIC and PROTECTED methods of its super class, it inherits all of the PUBLIC and PROTECTED methods available all the way to the top of the class hierarchy.

While a subclass can access the PUBLIC and PROTECTED methods of any of its super classes, the reverse is not true for instance methods. You cannot invoke instance methods within a class that are defined only by a subclass of that class. Methods first defined in a subclass are simply not visible to any of its super classes, even though the subclass exists in the same class hierarchy as its super classes for a given object. Likewise, methods defined as PRIVATE in a super class are not visible to its subclasses. Because the default access mode for methods is PUBLIC, a method is always accessible to a subclass unless you specifically define it as PRIVATE. Therefore, a key benefit of classes is the PROTECTED access mode, which allows you to define an interrelated set of method definitions that are accessible within a class hierarchy, but which are invisible to any procedures or other classes that are outside that class hierarchy.

Note that an instance can call a PROTECTED method of a second instance that is at the same level or in a super class in the class hierarchy, and an instance can call a PRIVATE method of another instance if both are the same class.

PACKAGE-PRIVATE and PACKAGE-PROTECTED access modes are another level of access control for methods. For more information, see "Access modes" in *Object-oriented ABL Developer Guided Journey*.

Comparison with procedure-based programming internal procedures and functions

By comparison, internal procedures and functions are always PUBLIC by default, which means that they can be executed from within a super procedure stack (through the RUN SUPER statement or SUPER built-in function), or from unrelated procedures. You can define internal procedures and functions as PRIVATE to restrict access to the defining procedure alone.

Data member and property scoping within a class hierarchy

Classes support the same definitions for variables, buffers, queries, temp-tables, and ProDataSets as procedures do, with one difference. That difference is in the access mode. Because of this difference, these built-in data types are referred to as data members when instances are defined for use at the class block level. For more information on what access modes are valid for each type of data member definition, see Define data members within a class on page 58. By default, all data members are PRIVATE.

Properties provide another means to define instances of certain data types for use at the class block level. Properties use the same access modes as data members. However, because properties are typically used to encapsulate access to class data from outside the class hierarchy, the default access mode for properties is PUBLIC.

As with methods, the PROTECTED access mode defines data members and properties that are accessible throughout the class hierarchy but not from outside it, and the PRIVATE access mode defines data members and properties that are accessible only from within the defining class.

Note that an instance can access a PROTECTED data member or property of a second instance that is at the same level or in a super class in the class hierarchy, and an instance can access a PRIVATE data member or property of another instance if both are the same class.

Comparison with procedure-based programming variables

Variables and other types of data definitions are also implicitly PRIVATE when used in procedures. You can achieve something like the PROTECTED capability using procedures by defining variables and other types of data as NEW SHARED in a parent procedure and as SHARED in a subprocedure. Shared data elements require that the definitions be repeated as SHARED in each subprocedure that shares them, which is typically done using an include file containing all of the definitions, so that they are guaranteed to match in all procedures that use them. The match between the NEW SHARED and corresponding SHARED definitions is verified at run time. However, the PROTECTED access mode also allows you to access data members and properties defined throughout the hierarchy without having to repeat the definitions in each class that uses them, and access to these PROTECTED class members is verified at compile time.

PUBLIC data members and properties really have no correspondence in procedure-based data, except for PUBLIC data members and properties that are also static. PUBLIC static class data elements are similar to GLOBAL SHARED data elements in procedures, because both are scoped to the session and cannot be deleted. For more information on static data members and properties, see Use static members of a class on page 186.

Event scoping within a class hierarchy

A PRIVATE event can have a handler subscribed to it only by the defining class. An instance can subscribe a handler to a private event of another instance if both instances are of the same class. A PROTECTED event can have a handler subscribed to it by the defining class and any of its derived classes. A PACKAGE-PRIVATE event can have a handler subscribed to it by the defining class and any class within its package. A PACKAGE-PROTECTED event can have a handler subscribed to it by the defining class, any class within its package, and from within any subclass that inherits the class. A PUBLIC event can have a handler subscribed to it from any code (in a class or a procedure) that has access to a class instance. For more information, see "Access modes" in *Object-oriented ABL Developer Guided Journey*.

Comparison with procedure-based programming PUBLIC and PRIVATE events

By comparison, named events are always PUBLIC by default, which means that handlers can be subscribed (through the SUBSCRIBE statement) to named events from within an owning super procedure stack or from any code that has a handle to the procedure object that owns the event and its handler procedures. You can also define internal procedures as PRIVATE to restrict handler subscriptions for named events to the defining procedure.

Override data within a class hierarchy

It is invalid to have a data member or implemented property in a subclass with the same name as any PUBLIC, PACKAGE-PROTECTED, PROTECTED, or PACKAGE-PRIVATE data member or implemented property in one of its super classes. This is true regardless of whether the data types of the two data members or properties match, because data members and properties share the same namespace. Thus, it is impossible to define data in a subclass that might override (shadow) or conflict with data in a super class without returning an error from the compiler.

Remember that this restriction applies to definitions at the level of the class. It is permissible to define a variable within a method that duplicates (and therefore shadows) a data member or property of the same name defined outside the method. Essentially, defining a variable in a method with the same name as a data member or property in the class hierarchy makes the data member or property invisible and directly inaccessible from within the method.

Note that you **must** override and implement any abstract properties in a non-abstract subclass that inherits them. However, if the inheriting subclass is abstract, you can, at your option, override any inherited abstract property and designate it again as abstract. The only difference you can make in creating a new abstract property definition is to specify a less restrictive access mode, overriding a protected abstract property to make it public. Note, again, that you cannot override an implemented property and make it abstract.

Note, also, that you can read or write any accessible property, whether it is abstract or implemented at the point of access. When you read or write an abstract property, you are always accessing the override that implements the property in the class hierarchy.

For more information on accessing data members and properties inside and outside of a class hierarchy, see Access data members and properties on page 153.

Override methods within a class hierarchy

A class definition (subclass) that inherits from another class can define a method to override a PUBLIC, PACKAGE-PROTECTED, PROTECTED, or PACKAGE-PRIVATE method in its class hierarchy as long as it is not marked FINAL. The method must have a compatible access mode and the same method name, scope, return type, and number of parameters, and each corresponding parameter must be of the same mode (INPUT, OUTPUT, or INPUT-OUTPUT) and of the same data type as the method it is overriding. A compatible access mode means that the overriding method can have a less restrictive access mode than the overridden method. For example, a PUBLIC subclass method can override a PROTECTED super class method. The same scope means the methods must both be instance methods or both be static methods. The overriding method must also use the OVERRIDE keyword to assure the compiler that the override is intended. The method that is overridden does not need to be in the immediate super class. If ABL does not find a matching method in the immediate super class in the hierarchy, it searches further up the hierarchy. If no matching method is found in the hierarchy, ABL returns a compile-time error.

Note: The remainder of this section describes how overriding works for instance methods. For information on how overriding works for static methods, see Call overridden and super-class static methods on page 189.

Once a method has been overridden, all calls to that method invoke the method in the most derived subclass where it is overridden. This is true whether the method is called from outside (using an object reference) or from inside the class hierarchy of the object. Thus, you can call any accessible method, whether it is abstract or implemented at the point of invocation. For more information on calling methods inside and outside of a class hierarchy, see Call class-based methods on page 140.

Note that you **must** override and implement any abstract methods in a non-abstract subclass that inherits them. However, if the inheriting subclass is abstract, you can, at your option, override any inherited abstract method and designate it again as abstract. The only difference you can make in creating a new abstract method definition is to specify a less restrictive access mode, overriding a protected abstract method to make it public. Note, also, that unlike an implemented property, you **can** override an implemented method and make it abstract as long as the inheriting subclass is also abstract.

Override class events within a class hierarchy

It is invalid to have an implemented class event in a subclass with the same name as any `PUBLIC`, `PACKAGE-PROTECTED`, `PROTECTED`, or `PACKAGE-PRIVATE` class event in one of its super classes, regardless of whether the signatures of the two events match. Thus, it is impossible to define an event in a subclass that might override (shadow) or conflict with an implemented event in a super class without returning an error from the compiler.

Note that you **must** override and implement any abstract events in a non-abstract subclass that inherits them. However, if the inheriting subclass is abstract, you can, at your option, override any inherited abstract event and designate it again as abstract. The only difference you can make in creating a new abstract event definition is to specify a less restrictive access mode, overriding a protected abstract event to make it public. Note, again, like an implemented property, that you cannot override an implemented event and make it abstract.

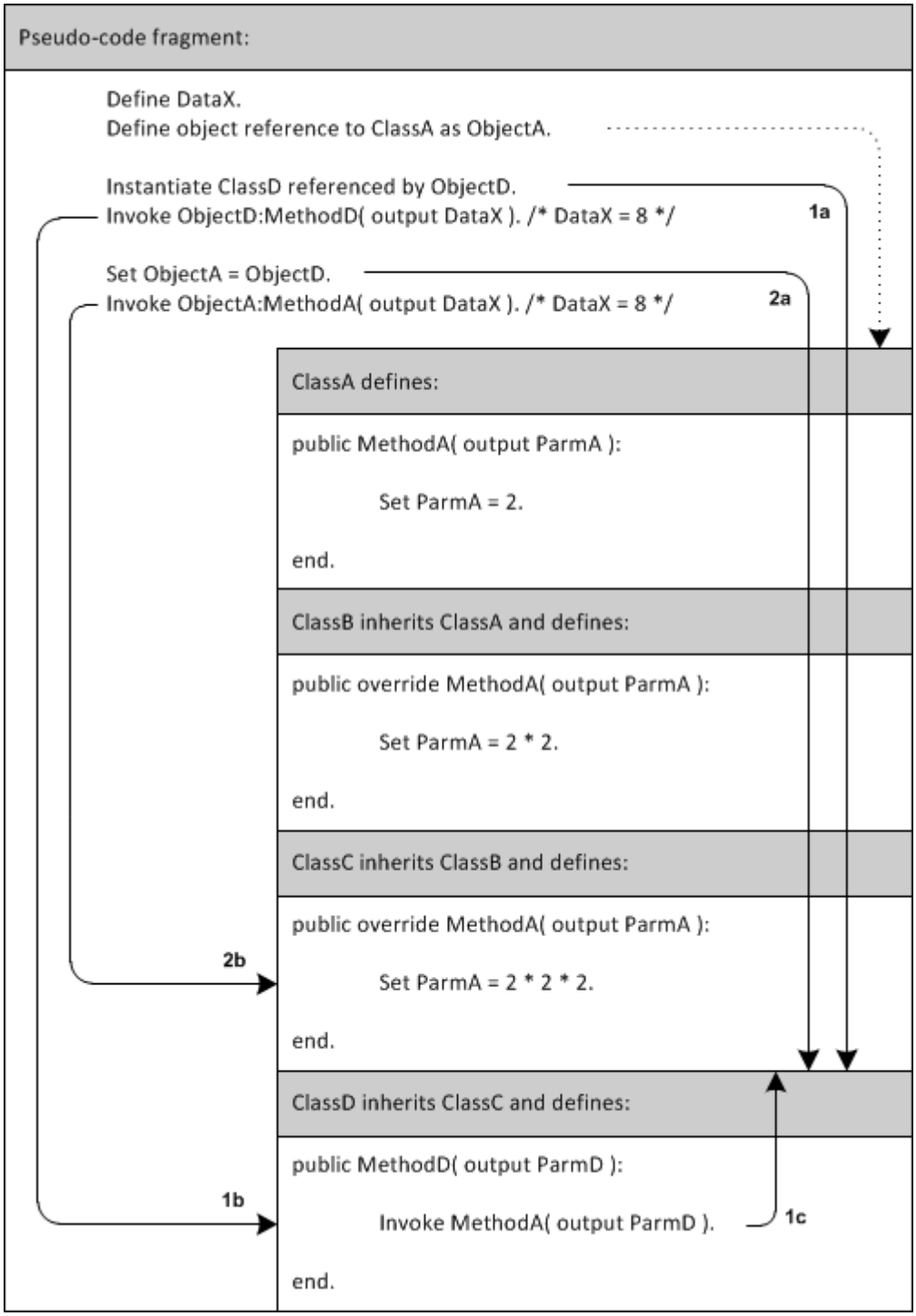
Note also that for abstract events, unlike for abstract properties or methods, the point at which you provide the non-abstract definition of the event determines where you can publish the event. However, you can subscribe a handler for an abstract event at all the same points in a class hierarchy where you can access an abstract property or method. Thus, you can subscribe a handler for any accessible event, whether the event is abstract or implemented at the point in a class hierarchy where you create the event handler subscription. However, you can publish an implemented event only within the class definition that implements it, which is the class that contains a `DEFINE EVENT` statement for the event without the `ABSTRACT` option. For more information on publishing and subscribing event handlers for a class event inside and outside of a class hierarchy, see Publish and subscribe to class events on page 167.

The following figure shows how a method override invoked on the defining super class executes within a class hierarchy, using pseudo-code to represent the behavior. The figure shows four classes forming an inheritance hierarchy with ClassA as the root class and ClassD as the most derived subclass. ClassA defines a MethodA( ), which is overridden, in turn, by ClassB, then by ClassC, which is also the most derived subclass that overrides the method.

This pseudo-code fragment defines an object reference (dotted arrow) to the ClassA data type (ObjectA), and according to the numbered arrows:

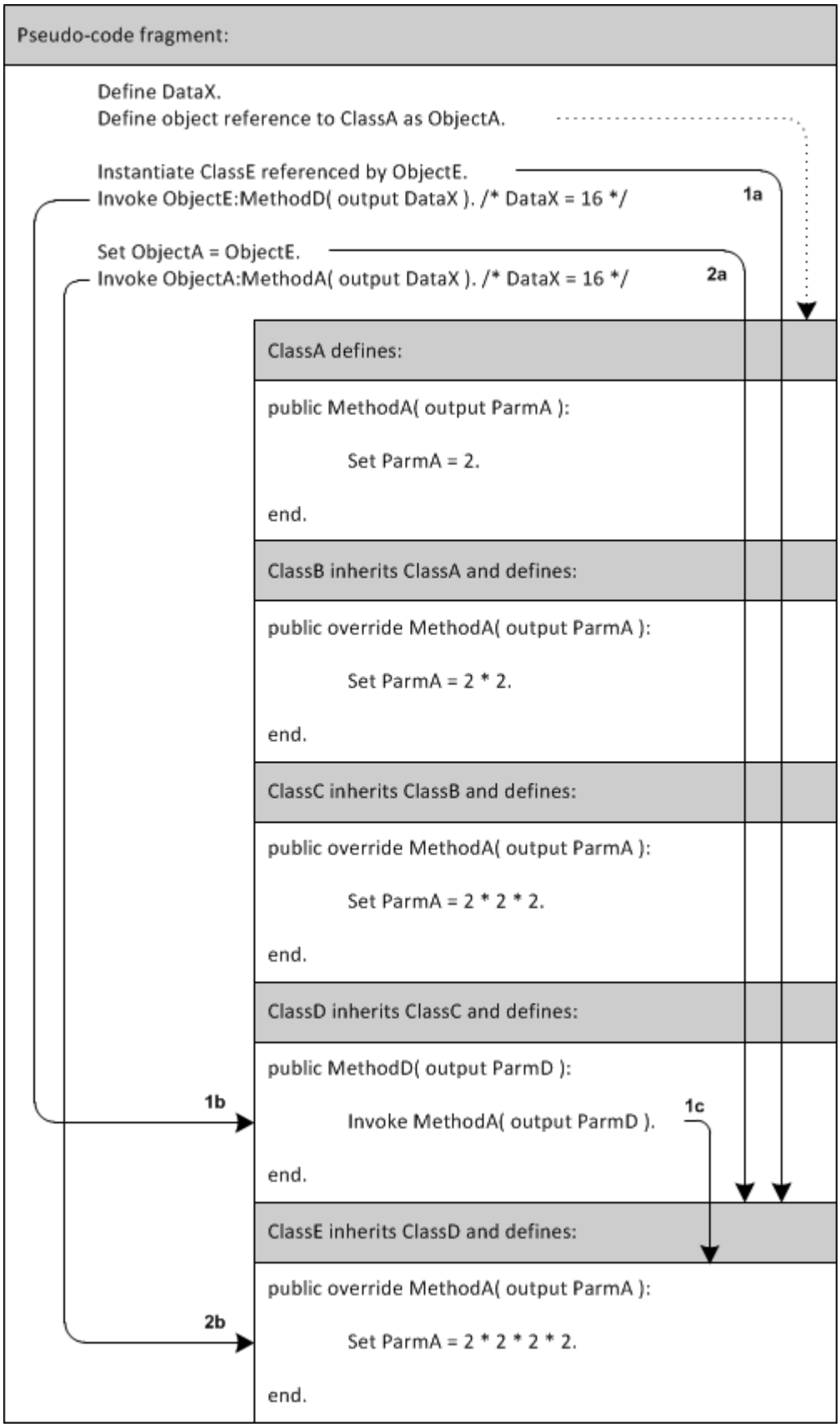
1. Instantiates ClassD, returning its object reference as ObjectD (**1a**) and executes MethodD( ) on ObjectD (**1b**). MethodD( ) then invokes MethodA( ) within the class hierarchy of the object. Because the most derived subclass in the object that overrides MethodA( ) is ClassC, ClassC's definition of this method executes (**1c**).
2. Assigns the ClassA object reference, ObjectA, to reference the ClassD instance (**2a**) and executes MethodA( ) on ObjectA (**2b**), which references the object as a ClassA instance and invokes the same MethodA( ) override (ClassC's) that was previously invoked through the ObjectD reference (**1c**).

Figure 7: Invoking an overridden method in a class



Now, suppose ClassE inherits from the ClassD and also overrides MethodA( ), as shown in the following figure.

Figure 8: Invoking an overridden method in a class extension



In this hierarchy, ClassE is the most derived subclass that overrides the method. So, as shown according to the numbered arrows, this pseudo-code:

1. Instantiates ClassE, returning its object reference as ObjectE (**1a**) and executes the inherited MethodD( ) on ObjectE (**1b**). MethodD( ) then invokes MethodA( ) within the class hierarchy of the object. Because the most derived subclass that overrides MethodA( ) is now ClassE, ClassE's definition of that method executes (**1c**).
2. Assigns the ClassA object reference, ObjectA, to reference the ClassE instance (**2a**) and executes MethodA( ) on ObjectA (**2b**), which references the object as a ClassA instance and invokes the same MethodA( ) override (ClassE's) that was previously invoked through the ObjectE reference (**1c**).

Note that, in general, you cannot invoke overrides of a method other than the override in the most derived subclass. For example, there is no way to directly access the MethodA( ) override defined in ClassB, even from within ClassB, because all direct calls to MethodA( ) anywhere inside or outside the class hierarchy execute the override in ClassE.

However, using the SUPER system reference, ABL allows any method in a subclass to invoke the behavior of a specified method in the nearest super class where it is defined. Thus, the MethodA( ) override defined in ClassE can invoke the behavior for the MethodA( ) definition that it overrides in ClassC. Similarly, MethodD( ) can use the SUPER system reference to invoke the behavior of the MethodA( ) defined in ClassC instead of invoking the override in the most derived subclass, ClassE. For more information on the SUPER system reference, see SUPER system reference on page 119.

If the super class's method is defined as PRIVATE, the method is not inherited by the subclass. If the subclass defines a method of the same name, it is not overriding the super class method because the super class's definition is not available to the subclass. In this case, the subclass method is entirely independent of the super class method. Because of this, it does not matter whether the methods match in their return type, access mode, or parameters. Also in this case, use of the OVERRIDE keyword on the method is not allowed.

If the super class's non-private method is defined as FINAL, no subclass can override that method. The compiler returns an error if you attempt to compile a subclass that has a method with the same name as a FINAL method in a super class, regardless if the signatures match.

The following sample class references and extends the sample subclass, `acme.myObjs.CustObj` (see Sample classes on page 225). This sample extension overrides the `GetCustomerName( )` method to return an E-mail address with the customer name:

```
USING acme.myObjs.*.

CLASS acme.myObjs.NECustObj INHERITS CustObj:

  DEFINE PRIVATE TEMP-TABLE ttEmail NO-UNDO
      FIELD RecNum AS INTEGER
      FIELD Name   AS CHARACTER FORMAT "X(20)"
      FIELD Email  AS CHARACTER FORMAT "X(20)".

  CONSTRUCTOR PUBLIC NECustObj (INPUT EmailFile AS CHARACTER):
    ...
    /* Code to initialize ttEmail:                          */
    ...
  END CONSTRUCTOR.

  /* Override method to always get customer name and email */
  METHOD PUBLIC OVERRIDE CHARACTER GetCustomerName
      (INPUT piCustNum AS INTEGER):
      VAR CHAR EmailName = SUPER:GetCustomerName (piCustNum).

      FIND FIRST ttEmail WHERE ttEmail.Name = EmailName NO-ERROR.
      IF AVAILABLE (ttEmail) THEN
         RETURN EmailName + ";" + ttEmail.Email.
      ELSE
         RETURN EmailName.
  END METHOD.

END CLASS.
```

This extension adds a new temp-table to provide the E-mail address, which is initialized by the constructor for `NECustObj`. Note that only methods can be overridden, constructors and destructors are implicitly `FINAL` and cannot be overridden in a subclass.

Method overriding supports polymorphism, because it allows you to call a method on a super class object reference, and the method invoked depends on the actual class instance that overrides the super class definition of that method. Thus, you can invoke different behavior using an identical method call, depending on the subclass object referenced by the super class object reference. At compile time, you do not necessarily know the class type of the object whose method override will execute. However, the method override that executes is always the method definition in the most derived class that defines the method in the hierarchy of that class type.

From a practical programming viewpoint, then, method overriding simplifies the code to execute different behaviors that might otherwise require long `CASE` or nested `IF` statements to select among them. The new behavior is selected simply by referencing an instance of a different class that overrides the same method call. For more information on how to use polymorphism with method overriding, see Use polymorphism with classes on page 125.

You can also define multiple methods with the same name that you invoke with different signatures, a practice known as overloading. Method overloading provides a mechanism that you can use together with polymorphism to identify specific method calls, but you need to know at compile time the particular signature of each overloaded method you want to call in the class hierarchy of the object. For more information on method overloading, see Overload methods and constructors on page 113.

Overload methods and constructors

A class definition can provide multiple methods, including constructors, that have the same name but different signatures. The signatures defined for overloaded methods and constructors must be different from one another in at least one variation in the number, type, or mode of their parameters. However, note that a static method cannot overload an instance method with an identical signature, because the method scope (determined by use of the `STATIC` option in the `METHOD` statement) does not count in the overloading.

Overloading thus allows a class to be instantiated, and allows multiple methods of a class with the same name to be called, using different arguments. Regardless of the arguments, a constructor always instantiates a class of the same type as any other overloaded constructor defined for the class, and a method of a given name typically invokes behavior that is similar to the behavior of every other overloaded method of the same name. In fact, it does not matter what behavior each method overloading provides, as long as the compiler can disambiguate the signatures at both compile time and run time. Thus, for methods, overloading provides a convenience, allowing you to conserve method names over some set of behaviors. For classes, constructor overloading provides both the convenience and power of being able to instantiate the same type of object using different sets of initial data. Note, however, that only instance constructors support overloading, as a class can have only one static constructor.

A class definition can overload its own methods and constructors. As a subclass, it can also inherit the overloaded methods (except the constructors) of a super class. It can also overload methods inherited from a super class, and it can override any existing overloadings of the methods that it inherits.

Method overloading provides a mechanism that you can use to invoke different behaviors in a similar way, somewhat like polymorphism (overriding). However, method overloading is much less powerful than polymorphism, because although you call a method of the same name, you need to specify a different signature to identify the different behavior, as compared to calling an identical overridden method on a different subclass instance to provide the different behavior polymorphically. Thus, method overloading represents an economy of notation (provided by conserving method names) rather than an economy of programming (provided by calling a single method), as represented by overriding. For more information on overriding, see Override methods within a class hierarchy on page 107.

Define overloaded methods and constructors

Although the use cases for overloading methods and constructors are different, the requirements for defining an overloaded method or constructor are almost the same. To define an overloaded method or constructor, you must do one of the following:

- Define a different number of parameters than any of the others.
- Ensure that at least one common parameter among them has a different mode (`INPUT`, `OUTPUT` or `INPUT-OUTOUT`) from any of the others.
- Ensure that at least one corresponding parameter among them has a different data type definition from any of the others. In addition to basic differences in data type, such as between different primitive types, different class or interface types, different temp-tables, and different ProDataSets, differences in data type can take the following forms:
  - For a data type that can be defined as an array, the difference can be whether it is defined as an array or not, whether it is defined as a determinate or indeterminate array, and whether two determinate arrays have different sizes (extents).
  - To differentiate static temp-table (`TABLE`) or static ProDataSet (`DATASET`) parameters, parameters of each type must differ in their defined schema.
  - A dynamic temp-table (`TABLE-HANDLE`) or ProDataSet (`DATASET-HANDLE`) parameter differs from any static temp-table or ProDataSet parameter, respectively. However, you can have only one method or

constructor that differs by specifying a TABLE-HANDLE rather than a TABLE parameter, or by specifying a DATASET-HANDLE rather than a DATASET parameter. In other words, because they have no schema associated with them at compile time, ABL cannot distinguish multiple TABLE-HANDLE parameters from each other or multiple DATASET-HANDLE parameters from each other.

- Finally, for methods, because method scope (specified by the STATIC option on the METHOD statement) does not participate in method overloading, you cannot define an instance and static method in a class that have the same method name and signature.

The following example shows a fragment of the sample class, acme.myObjs.CustObj, including the definitions of two overloaded methods distinguished by the number of parameters (one compared to none):

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj
                          IMPLEMENTS IBusObj:
...

  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD ...

...

  /* First version of printObj prints a single copy of a report */
  METHOD PUBLIC VOID printObj ( ):
    OUTPUT TO PRINTER.
    DISPLAY dtTimestamp.
    FOR EACH ttCustomer:
      DISPLAY ttCustomer.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to printer").
  END METHOD.

  /* Second version of printObj takes an integer parameter representing the
     number of copies to print. */
  METHOD PUBLIC VOID printObj (INPUT piCopies AS INTEGER):
    VAR INT iCnt.
    OUTPUT TO PRINTER.
    IF piCopies <> 0 THEN
    DO iCnt = 1 TO ABS(piCopies):
      DISPLAY dtTimestamp.
      FOR EACH ttCustomer:
        DISPLAY ttCustomer.
      END.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated(STRING(piCopies)
                           + " copies of report sent to printer").
  END METHOD.

...

END CLASS.
```

The following example shows a fragment of the sample class, Main, including the definitions of two overloaded constructors distinguished by the number of parameters (one compared to none):

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
```

```
USING acme.myObjs.Interfaces.*.

CLASS Main:

  VAR PRIVATE CLASS CustObj rCustObj.
  ...
  VAR PRIVATE CLASS HelperClass rHelperClass.
  VAR PRIVATE CHARACTER outFile.

  CONSTRUCTOR PUBLIC Main ( ):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass ( )

      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj ( )
      cOutFile     = "Customers.out".

    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.

  /* Second constructor takes a character parameter representing an input
     file of email addresses to instantiate a New England Customer object */
  CONSTRUCTOR PUBLIC Main (INPUT EmailFile AS CHARACTER):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass( )

      /* Create an instance of the NECustObj class */
      rCustObj     = NEW NECustObj (EmailFile)
      cOutFile     = "NECustomers.out".
    /* Subscribe OutputGenerated event handler for NECustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_NECustObjHandler).
  END CONSTRUCTOR.

...

END CLASS.
```

For more information on defining parameters for overloaded methods and constructors, see the Parameter definition syntax reference entry in *ABL Reference*.

Invoke overloaded methods and constructors

In many cases, invoking overloaded methods and constructors is the same as invoking non-overloaded methods and constructors, by providing the correct parameter list at compile time. However, for some types of parameter lists, the AVM uses a set of rules to disambiguate the correct method or constructor to call at run time. Thus, for certain data types, this results in a powerful form of overloading that depends on run-time conditions rather than coding at compile time to invoke the appropriate method. For more information, see Call an overloaded method or constructor on page 150.

Construct an object

You can construct a non-abstract class instance (object) by invoking the NEW functionon the type name of the class including any parameter list required by the specified constructor. (For more information on using the NEW function, see Create a class instance on page 136.) As described previously (see Class hierarchies and inheritance on page 102), when an object of a subclass is instantiated at run time, it is constructed from the top super class of the class hierarchy down. Instantiating from the top down is important because the subclass can reference public and protected methods in its super class during the instantiation process, and the AVM must make sure that a super class has been constructed before the subclass invokes any of its methods.

Construction of the class hierarchy is facilitated by two ABL elements, each of which performs a different, explicit role in building the class hierarchy, SUPER statement on page 116 and THIS-OBJECT statement on page 117.

SUPER statement

In order to enforce the construction of an object hierarchy from the top of the object hierarchy to the bottom, the very first action of the instantiating subclass constructor (as invoked by the NEW function) must either be an implicit call to the default constructor of the immediate super class or an explicit call to a constructor of the immediate super class using the SUPER statement. If all super class constructors have parameters (that is, there is no default super class constructor), the first executable statement in the instantiating subclass constructor must be either an explicit call to a super class constructor using the SUPER statement or an explicit call to an overloaded constructor using the THIS-OBJECT statement whose first statement explicitly or implicitly invokes a super class constructor. If the super class has no constructors defined or defines a constructor with no parameters (that is, it has a default constructor), an explicit call to a super class constructor is optional. Also, with a default super class constructor, if a subclass does not need to execute any statements of its own to initialize work when the subclass is instantiated, the subclass does not need to define a constructor. The NEW function automatically invokes the default constructor for the instantiated subclass when the NEW function is specified with no parameters, and the AVM automatically calls the default super class constructor as the initial action of the instantiating subclass constructor.

This is the syntax to invoke the immediate super class constructor using the SUPER statement:

Syntax

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

Element description for this syntax diagram follow:

[*parameter* **[** *, parameter* **]...]**

The parameters, if any, passed to the specified super class constructor. For more information on the syntax of *parameter*, see the Parameter passing syntax reference entry in *ABL Reference*.

Note: It is a compiler error to specify NO-ERROR on the SUPER statement. Any ERROR condition raised during execution of a constructor can be handled only by the statement that instantiates a class with the NEW function. For more information, see Raise and handle error conditions on page 197.

The syntax for the SUPER statement appears similar to the syntax for the SUPER built-in function, used to invoke user-defined functions in a super procedure. However, unlike the SUPER built-in function, which you must invoke inside an expression in a procedure, you can only invoke the SUPER statement as the first statement in a subclass constructor.

THIS-OBJECT statement

A super class constructor can only be called (explicitly or implicitly) once from the instantiating subclass constructor as its very first action. If all immediate super class constructors are defined with parameters (there is no default constructor), it is invalid for the instantiating constructor not to call a super class constructor, either explicitly, using the SUPER statement, or implicitly through an explicit call to an overloaded constructor using the THIS-OBJECT statement whose first action is ultimately an explicit call to a super class constructor using the SUPER statement. If the super class has a default constructor, there is no need for a subclass to use the SUPER statement to explicitly invoke a super class constructor, unless the application requires it.

Note: If you later define a constructor for the super class that takes parameters, but do not also explicitly define a default constructor for the super class that does **not** take parameters, all subclasses then must be updated to explicitly invoke the super class's new constructor with parameters.

This is the syntax to invoke an overloaded constructor (defined in the instantiated class) using the THIS-OBJECT statement:

Syntax

```
THIS-OBJECT ( [parameter[ , parameter]...] ) .
```

Element description for this syntax diagram follow:

[*parameter* **[** *, parameter* **]...]**

The parameters, if any, passed to the specified constructor. For more information on the syntax of *parameter*, see the Parameter passing syntax reference entry in *ABL Reference*.

Note: It is a compiler error to specify NO-ERROR on the THIS-OBJECT statement. Any ERROR condition raised during execution of a constructor can be handled only by the statement that instantiates a class with the NEW function. For more information, see Raise and handle error conditions on page 197.

If all immediate super class constructors are defined with parameters (there is no default super class constructor), the first statement of a constructor must either be the SUPER statement to invoke a super class constructor, or the THIS-OBJECT statement to call another constructor overloading with the current class.

Refer, again, to this sample class hierarchy:

```
Progress.Lang.Object             <--- Top of hierarchy
  acme.myObjs.Common.CommonObj
    acme.myObjs.CustObj
      acme.myObjs.NECustObj      <--- Bottom of hierarchy
```

A class or procedure that uses the NEW function to instantiate acme.myObjs.NECustObj invokes the NECustObj( ) constructor. This constructor must first execute its super class constructor. As previously noted, this can be either an explicit call—with the SUPER statement as the first executable statement—or an implicit call to the CustObj( ) constructor. The acme.myObjs.CustObj class, in turn, does the same for its super class constructor in acme.myObjs.Common.CommonObj. Because acme.myObjs.Common.CommonObj is the top of the user-defined class hierarchy, it is the first user-defined constructor to execute to completion.

But before the constructor for acme.myObjs.Common.CommonObj runs, the constructor for the built-in root class, Progress.Lang.Object, is executed, which executes standard startup behavior required by the AVM to instantiate classes at run time. The constructor for the top-level user-defined class does not ever need to explicitly invoke the constructor for Progress.Lang.Object (which relies on the default). Once the top-level user-defined constructor in acme.myObjs.Common.CommonObj completes, the CustObj( ) constructor is executed to completion followed by the NECustObj( ) constructor.

Note: If a class or procedure uses the NEW function to instantiate an additional acme.myObjs.NECustObj object, the r-code for all classes in the new object's class hierarchy is shared with the previously instantiated object, but the constructors for all classes execute again for the new object instance (possibly, with different data).

The following example adds a constructor to the sample subclass, acme.myObjs.NECustObj, described in Override methods within a class hierarchy on page 107. This constructor adds code to initialize the class temp-table, ttEmail, as shown:

```
USING acme.myObjs.*.

CLASS acme.myObjs.NECustObj INHERITS CustObj:

  DEFINE PRIVATE TEMP-TABLE ttEmail NO-UNDO
    FIELD RecNum AS INTEGER
    FIELD Name AS CHARACTER FORMAT "X(20)"
    FIELD Email AS CHARACTER FORMAT "X(20)".

  CONSTRUCTOR PUBLIC NECustObj (INPUT EmailFile AS CHARACTER):
   /* Because there are no parameters to the super class's constructor,
          this constructor call is optional */
     SUPER ( ). /* 8 */

    /* Code to initialize ttEmail: */
    . . .

  END CONSTRUCTOR.

  /* Override method to always get customer name and email */
  METHOD PUBLIC OVERRIDE CHARACTER GetCustomerName
    (INPUT piCustNum AS INTEGER):
    VAR CHAR EmailName.

    EmailName = SUPER:GetCustomerName (piCustNum).
    FIND FIRST ttEmail WHERE ttEmail.Name = EmailName NO-ERROR.
    IF AVAILABLE (ttEmail) THEN
      RETURN EmailName + ";" + ttEmail.Email.
    ELSE
      RETURN EmailName.
  END METHOD.

END CLASS.
```

Constructors can have an access mode of PUBLIC, PACKAGE-PROTECTED, PROTECTED, or PACKAGE-PRIVATE. The following example demonstrates the use of a PROTECTED constructor. The super class SuperOnly cannot be instantiated directly from outside the class hierarchy, because it does not have a PUBLIC constructor.

Class SuperOnly can only be instantiated by instantiating a class that inherits from it, such as MyPubClass:

```
CLASS SuperOnly:
  CONSTRUCTOR PROTECTED SuperOnly ( ):
    /* An attempt to use the NEW function for SuperOnly will work only from
       within the class hierarchy. This constructor can only be invoked by a
       subclass object's constructor. */
  END CONSTRUCTOR.
END CLASS.
```

```
CLASS MyPubClass INHERITS SuperOnly:
  CONSTRUCTOR PUBLIC MyPubClass ( ):
    SUPER( ). /* Invoke the Protected constructor */
    /* Perform operations to instantiate this class */
    ...
  END CONSTRUCTOR.
END CLASS.
```

Call up the class hierarchy

In addition to the SUPER statement and the THIS-OBJECT statement, two other ABL elements allow you to access objects in the class hierarchy, the SUPER system reference and the THIS-OBJECT system reference.

SUPER system reference

You can invoke the super class implementation of an instance method using the SUPER system reference. Executing a super class method using the SUPER system reference within a subclass causes control to switch to the version of the method in the next higher super class that implements it, bypassing any classes that do not implement the method. The SUPER system reference is normally used to invoke the super class behavior for a method from within the subclass method that overrides the invoked behavior, and there by construct a chain of behavior. However, it can also be used to invoke any other method in a super class. Therefore, the method name to invoke is part of the syntax.

Note: The SUPER system reference is different from the SUPER statement. You can only invoke the SUPER statement from within a constructor and it invokes a specified super class constructor. For more information, see SUPER statement on page 116.

This is the syntax to call a super class version of a method using the SUPER system reference:

Syntax

```
[ return-variable = ]
    SUPER:method-name ( [ parameter [ , parameter ] . . . ] ) [ NO-ERROR ]
```

Element descriptions for this syntax diagram follow:

return-variable

If the method returns a value that is assigned, the variable to hold the value returned by the method.

method-name

The name of the method to call in the class hierarchy. The first method found, starting from the immediate super class of the current class and searching upward in the class hierarchy, is the method executed. It might be anywhere in the class hierarchy starting from the immediate super class.

The name of the method can identify any valid method in the class hierarchy. Its name does not need to match the name of the method from which the call is made.

Note that the definition found for *method-name* in the super class hierarchy cannot be abstract. Otherwise, ABL raises a compiler error. Note that the intent and function of SUPER is to invoke the implementation of the method **definition** found in the super class hierarchy. If the method is abstract, its definition has no implementation and therefore cannot be executed.

[*parameter* **[** *, parameter* **]...]**

The parameters of the method. For more information on the syntax of *parameter*, see the Parameter passing syntax reference entry in *ABL Reference*.

NO-ERROR

Optionally redirects error processing when the method is called as a statement.

From within a class, any invocation of an overridden method without the use of the SUPER system reference invokes the most derived subclass's implementation of the method. From within a class, any invocation of an overridden method with the use of the SUPER system reference invokes the super class's implementation of the method. There is no direct access to the super class's implementation of the overridden method from outside the class.

The SUPER system reference does for class methods what the RUN SUPER statement does for internal procedures running in a persistent super procedure. Both provide access to code further up the execution stack. The RUN SUPER statement, which must appear in an internal procedure, runs the super procedure version of the current internal procedure.

THIS-OBJECT system reference

THIS-OBJECT is a system reference available for use within an instantiated class. At run time, it returns the currently running instance of the class as an object reference. You can optionally use THIS-OBJECT to access class members defined within the current class hierarchy.

The THIS-OBJECT system reference is different from the THIS-OBJECT statement. You can only execute the THIS-OBJECT statement within a constructor and it invokes another constructor in the same class definition.

This is the syntax to access local class members and class members defined within the current class hierarchy using the THIS-OBJECT system reference:

Syntax

```
THIS-OBJECT [ : class-member-reference]
```

class-member-reference

A reference to an instance variable data member, instance property, or instance method defined within the current class hierarchy. Any variable data member cannot be defined as an array (with an EXTENT).

Delete an object

A class-based, ABL object is either automatically deleted by the AVM during garbage collection when its reference count goes to zero, or manually deleted by the application using the `DELETE OBJECT` statement. In either case, the destructors (if any) for every class in the hierarchy are automatically run, from the bottom of the class hierarchy to the top.

Refer once again to this sample class hierarchy:

```
Progress.Lang.Object              <--- Top of hierarchy
  acme.myObjs.Common.CommonObj
    acme.myObjs.CustObj
      acme.myObjs.NECustObj       <--- Bottom of hierarchy
```

The `DELETE OBJECT` statement for an instance of `acme.myObjs.NECustObj` invokes the `NECustObj( )` destructor, if it has one, followed by the `CustObj( )` destructor, followed by the `CommonObj( )` destructor and finally the implicit destructor for the built-in root class, `Progress.Lang.Object`. The AVM executes all these destructors automatically. You do not use the SUPER statement in a destructor. Remember that a destructor is always public and it can have no parameters.

For more information about garbage collection and the `DELETE OBJECT` statement, see Garbage collection on page 98.

Unlike constructors, the destructors in the class hierarchy execute to completion from bottom to top. When an object is instantiated, code in its constructors must execute from the top down to initialize super class resources that might be referenced by a subclass. By contrast, the destructors have the opportunity to free resources created by a subclass before it terminates and passes control to its super class for further clean-up, which has no knowledge of or need to reference its subclass.

Interface hierarchies and inheritance

You can define an interface by inheriting the member prototypes from other interfaces as well as by defining member prototypes directly. Any class instance that implements the interface must implement the members of all super interfaces in the interface hierarchy as well as its own members. In addition, the class instance can be referenced as any of the super interface types as well as the interface type that inherits them. This provides an additional dimension to enable access to any object that implements the interface. The following topics detail how an interface hierarchy can be built, including limitations and guidelines for using it.

Multiple interfaces in a hierarchy

Each class implementing an interface is required to implement all members of the interface. However, a class inheriting from a class already implementing the same interface also inherits all of the implementations. Thus you need not explicitly implement the members.

Interface inheritance allows the same interface to appear multiple times within an interface hierarchy. That is, it is valid for an interface to inherit member prototypes from an interface that it already inherits from.

For example:

```
INTERFACE IMyX :
```

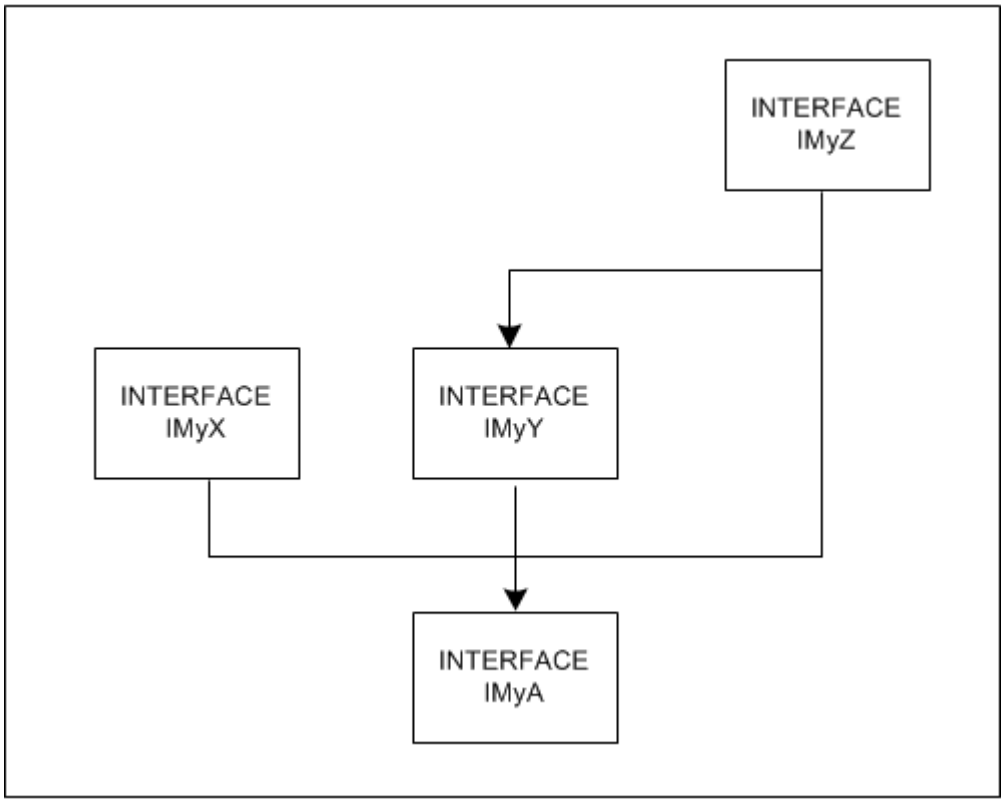
```
INTERFACE IMyY INHERITS IMyZ :
```

```
INTERFACE IMyZ :
```

```
INTERFACE IMyA INHERITS IMyX, IMyY, IMyZ :
```

The following figure depicts the interface hierarchy where interface `IMyZ` is inherited by both interface `IMyY` and `IMyA`.

Figure 9: Multiple Interfaces in hierarchy



It is not an error having interface `IMyZ` appear twice in the hierarchy. It is however, a compile time error to have an interface appear within its own hierarchy.

For example:

```
INTERFACE IMyJ INHERITS IMyJ :
```

The above example results in a compile time error.

Duplicate members in a hierarchy

Without interface inheritance, a class can implement more than one interface with multiple interfaces that include identical member signatures. In this case, the class is required to implement only one member that satisfies the identical signature in multiple interfaces.

With interface inheritance, the same rules apply. Multiple identical members appearing in different interfaces within the hierarchy are allowed and require only one implementation in the implementing class. Member types in the hierarchy that differ only by return type will generate compile time errors. Overloaded members may appear within the same interface, or in different interfaces within the hierarchy. This is comparable to existing rules with respect to overloading.

```
INTERFACE IMyTop:
  DEFINE PUBLIC PROPERTY myInteger AS INTEGER
    GET.
    SET.
END INTERFACE.
```

```
INTERFACE IMyBottom INHERITS IMyTop:
  DEFINE PUBLIC PROPERTY myInteger AS INTEGER
    GET.
    SET.
END INTERFACE.
```

```
CLASS MyImplementation IMPLEMENTS IMyBottom :
  DEFINE PUBLIC PROPERTY myInteger AS INTEGER
    GET():
      RETURN myInteger.
    END GET.
    SET(INPUT val AS INTEGER):
      myInteger = val.
    END SET.
END CLASS.
```

In the above example, it is allowed for interface `IMyBottom` to have its own definition of the existing property, `myInteger` that appears in `IMyTop` interface. Also, the class `MyImplementation` needs to implement only the single property that satisfies them both.

Use interface hierarchies

A class implementing an interface hierarchy does so in the same manner as a regular single interface. To the implementing class, the extended interface acts as a meta-interface that encapsulates all of the members of the interface hierarchy. This enhances the ability to implement more advanced interface polymorphism. You can implement any of the interfaces in the hierarchy.

For example:

```
INTERFACE ISedan  INHERITS IVehicle, ICar   :
```

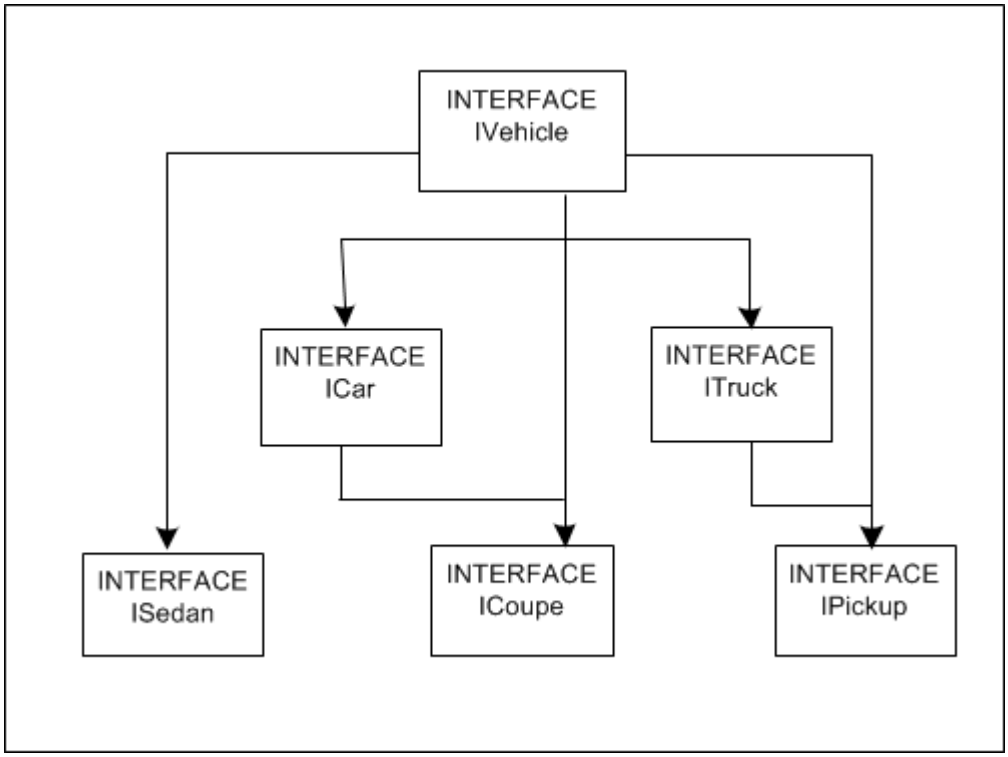
```
INTERFACE ICoupe  INHERITS IVehicle, ICar   :
```

```
INTERFACE IPickup INHERITS IVehicle, ITruck :
```

In the above example, three interfaces, `ISedan`, `ICoupe`, and `IPickup` inherit from the existing interfaces.

The following figure depicts the interface hierarchy where interface `ISedan`, `ICoupe`, and `IPickup` inherits member prototypes from the existing interfaces `IVehicle`, `ICar`, and `ITruck`.

Figure 10: Using interface hierarchy



The three classes that implement each extended interface are:

```
CLASS Altima  INHERITS Nissan IMPLEMENTS ISedan :
```

```
CLASS Mustang INHERITS Ford   IMPLEMENTS ICoupe  :
```

```
CLASS Tundra  INHERITS Toyota IMPLEMENTS IPickup :
```

An application making use of these types can choose to program to the interfaces via polymorphism.

For all three classes, both the function `TYPE-OF` and the `Progress.Lang.Class` method `IsA( )` will return true for `IVehicle`. The first two, `Altima` and `Mustang`, will both return true for `ICar`. Class `Tundra` will return true for `IPickup`.

```
PUBLIC METHOD LOGICAL handleVehicle(INPUT refVehicle AS CLASS IVehicle):

  IF TYPE-OF(refVehicle, "ICar") EQ TRUE
    THEN THIS-OBJECT:handleCar(refVehicle).

  ELSE IF TYPE-OF(refVehicle, "ITruck") EQ TRUE
    THEN THIS-OBJECT:handleTruck(refVehicle).
    ELSE THIS-OBJECT:handleGeneric(refVehicle).
```

The above example is of a client programming to the interfaces. All the three classes are valid as parameters to this method. The first two is passed into `handleCar( ) method`, while the third is passed to `handleTruck( )` method.

Use polymorphism with classes

Polymorphism is one of the most powerful advantages of object-oriented programming. Multiple subclasses that inherit from the same super class can override behavior in the super class by providing unique implementations for the methods defined in the super class. This allows each subclass to express a different behavior in response to the same method call on the super class. In effect, polymorphism allows the invocation of a given method on an object reference to a super class at compile time that is dispatched to the overriding method in the actual instantiated subclass at run time.

Thus, a super class defines a given method and a derived class overrides the method with its own behavior. As described previously, overrides of this method can be defined in additional subclasses of the initial overriding derived class to any depth in the class hierarchy of an object. However, the effective method override for any given class hierarchy is the override defined by the most derived subclass in the hierarchy.

To polymorphically invoke an overridden method on an object reference:

1. Instantiate a class whose hierarchy overrides the method defined in one of its super classes.
2. Obtain an object reference to the instantiated class whose class type is the super class that defines the method. Some ways of doing this include:

- Passing the object reference to the instantiated class as an argument to a method whose corresponding parameter is defined as the appropriate super class type.
- Assigning the result of the NEW function that instantiates the class to an object reference data element of the appropriate super class type.

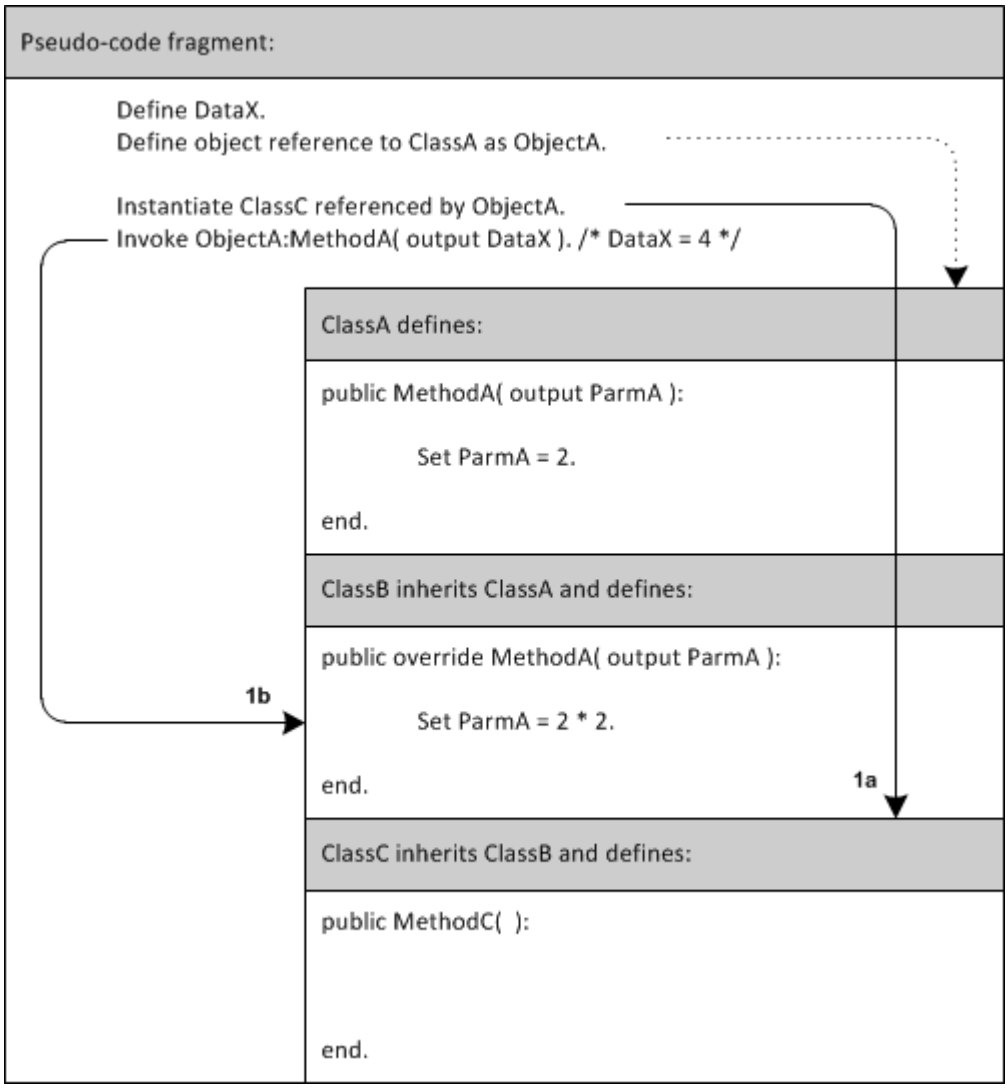
3. Invoke the method on this super class object reference.

The actual method that executes is the override defined in the most derived subclass within the object's class hierarchy.

The following figure shows how a method can be called polymorphically on the defining super class within a class hierarchy, using pseudo-code to represent the behavior. The figure shows three classes forming an inheritance hierarchy with ClassA as the root class and ClassC as the most derived subclass. ClassA defines a MethodA( ), which is overridden by ClassB, which is also the most derived subclass that overrides the method.

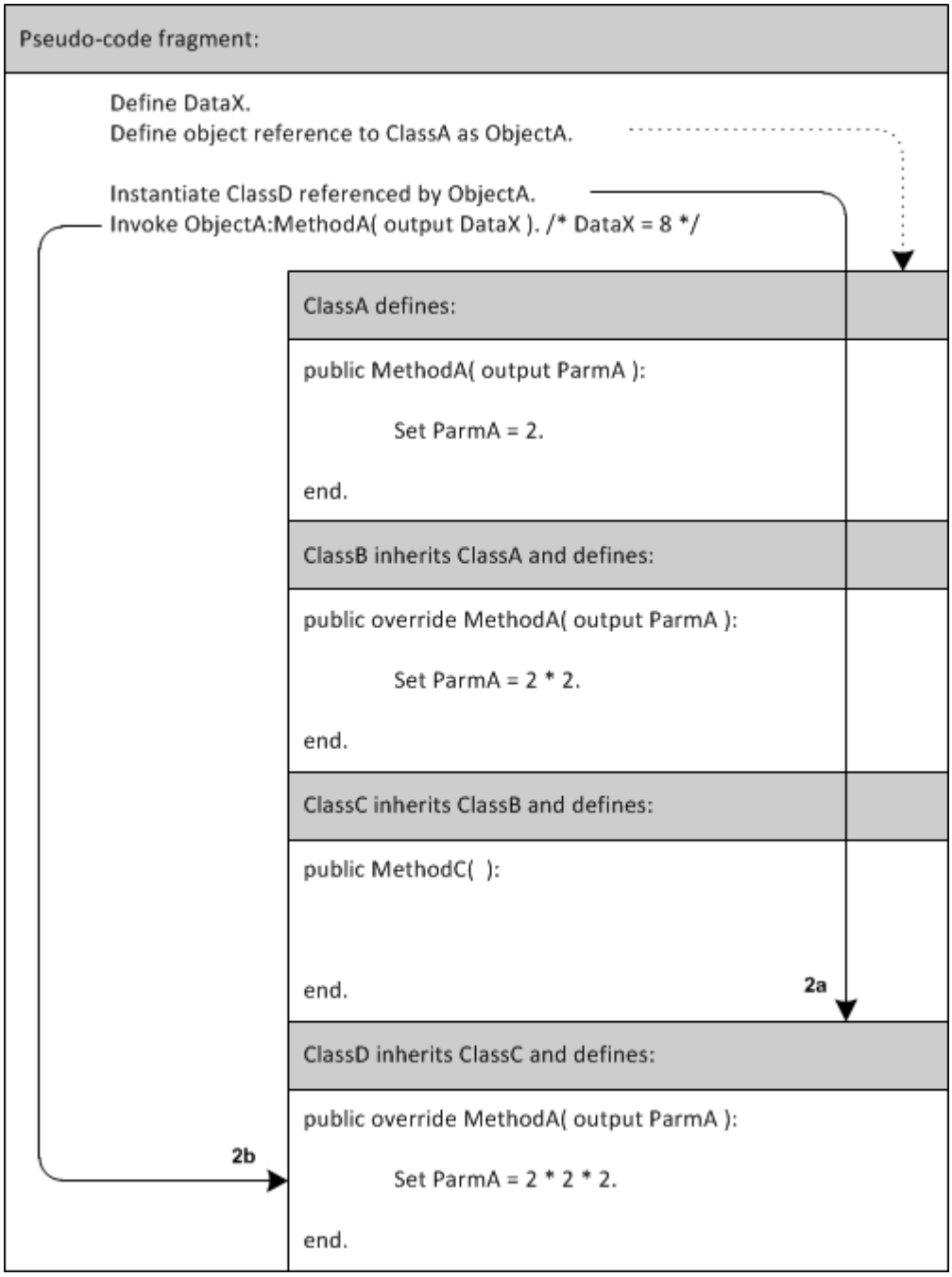
The pseudo-code fragment defines an object reference (dotted arrow) to the ClassA data type (ObjectA). As shown by the numbered arrows, the code then instantiates ClassC (**1a**), setting ObjectA to reference the instance, calls MethodA( ) on ObjectA. The method that executes is the override defined in the most derived overriding subclass, ClassB (**1b**).

Figure 11: Invoking a method polymorphically (one subclass)



The following figure shows another polymorphic method call. In this case, the object is instantiated from ClassD (**2a**), which extends the same class hierarchy and also overrides MethodA( ). So, the call to MethodA( ) on ObjectA executes the override defined in the most derived overriding subclass, ClassD (**2b**).

Figure 12: Invoking a method polymorphically (another subclass)



The following ABL classes show a practical example of polymorphism, where the class instances that define method overrides are passed to a common application method (`displayArea( )`) that, in turn, calls an overridden method (`calculateArea( )`) on a `ShapeClass` super class that originally defines the method.

In this example, `ShapeClass` is extended by both `CircleClass` and `RectangleClass`. The super class `ShapeClass` defines the method `calculateArea( )`, as shown:

```
CLASS ShapeClass:

  METHOD PUBLIC DECIMAL calculateArea ( ):
      /* Dummy routine */
     MESSAGE "If you got here someone did not override this method!"
         VIEW-AS ALERT-BOX.
  END METHOD.

END CLASS.
```

Both `RectangleClass` and `CircleClass` also define a `calculateArea( )` method. The `RectangleClass` uses the length and width, as shown:

```
CLASS RectangleClass INHERITS ShapeClass:

  VAR PRIVATE DECIMAL length.
  VAR PRIVATE DECIMAL width.

  CONSTRUCTOR PUBLIC RectangleClass
    (INPUT l AS DECIMAL, INPUT w AS DECIMAL):
    ASSIGN
      length = l
      width  = w.
  END CONSTRUCTOR.

  METHOD PUBLIC OVERRIDE DECIMAL calculateArea ( ):
       RETURN length * width.
  END METHOD.

END CLASS.
```

The `CircleClass` needs the radius only, as shown:

```
CLASS CircleClass INHERITS ShapeClass:

  VAR PRIVATE DECIMAL pi = 3.14159.
  VAR PRIVATE DECIMAL radius.

  CONSTRUCTOR PUBLIC ShapeClass (INPUT r AS DECIMAL):
    radius = r.
  END CONSTRUCTOR.

  METHOD PUBLIC OVERRIDE DECIMAL calculateArea ( ):
       RETURN pi * radius * radius.
  END METHOD.

END CLASS.
```

The following Main class demonstrates the polymorphic behavior of these classes:

```
CLASS Main:

 VAR CLASS RectangleClass rRectangle.
 VAR CLASS CircleClass rCircle.
  VAR DECIMAL width = 10.0.
  VAR DECIMAL length = 5.0.
  VAR DECIMAL radius = 100.0.

  CONSTRUCTOR PUBLIC Main ( ):
      rRectangle = NEW RectangleClass(width, length).
      rCircle    = NEW CircleClass (radius).
    displayArea( rRectangle ).
    displayArea( rCircle ).
  END CONSTRUCTOR.

  METHOD PUBLIC VOID displayArea( INPUT rShape AS CLASS ShapeClass ):
    MESSAGE rShape:calculateArea( ) VIEW-AS ALERT-BOX.
  END METHOD.

END CLASS.
```

Note that the first two definitions define rRectangle and rCircle variables as references to the RectangleClass and CircleClass, respectively. When each type of ShapeClass instance is created, the specific shape type is specified in the NEW function to *specialize* the general ShapeClass type.

To operate on these RectangleClass and CircleClass instances polymorphically, the displayArea( ) method specifies the general ShapeClass type as an INPUT parameter. This parameter can accept an object reference to any subclass of ShapeClass. This means that while only methods defined in ShapeClass can be invoked with this object reference, the implementation that is executed depends on the actual subclass instance that the ShapeClass object reference is pointing to. In this case, displayArea( ) invokes the ShapeClass method, calculateArea( ), in order to display the area of the particular ShapeClass object that is input, according to its instantiated class type.

The constructor for Main, then, instantiates an instance of RectangleClass and CircleClass and passes each one, in turn, to displayArea( ). Because of the polymorphic relationship to these specialized subclass instances of ShapeClass, the version of calculateArea( ) that is run is the version in each RectangleClass and CircleClass instance, respectively.

So, the use of polymorphism depends on the principle that a general domain exists over which a set of common operations can be applied and tailored to suit the requirements of more specialized subsets of that domain. In the previous example, the general domain is the domain of shapes and each specialized domain consists of different types of shapes, such as rectangles and circles. The subclass for each of these specialized shapes implements the same common set of operations to suit the requirements of its own specialized domain subset.

In the larger domain of business applications, one can similarly imagine many different uses for polymorphism. For example, in a general ledger system, you might have a super class for the domain of general accounts that provides a set of methods that operate on all accounts. You might then have one subclass representing asset accounts and another subclass representing liability accounts that both inherit from the general accounts class. In a general ledger system, some of the same methods inherited from the general accounts class would function as inverses of each other as implemented in the liability accounts class or the asset accounts class.

You might then create even more specialized subclasses that inherit from the asset accounts or liability accounts class. So, for example, in a hospital system, you might create a subclass of employee accounts that inherits from the liability accounts class and a subclass of patient accounts that inherits from the asset accounts class. Of course, there are many additional accounts that might be represented as subclasses of either the liability or asset accounts class in such a system.

Thus, by understanding the domain of a business environment and the specific problem that an application is intended to solve, you can virtually always represent the solution in terms of these polymorphic relationships between more general and more specialized domains. And you can likely implement any of the problem solutions for these domains using classes in ABL.

Use delegation with classes

Object-oriented design provides multiple models for facilitating code reuse. One model is inheritance, which is the model shown in most of the examples thus far. The other code reuse model is the delegation model. This topic demonstrates how you can use the delegation model in ABL. In the delegation model one class acts as a principal object for a set of related behavior that is implemented by other classes. This principal object is called a *container class*. The container class can create instances of other classes with the `NEW` function. These other classes that provide services to the container are called *delegate classes*. The container class executes delegate class behavior by forwarding method invocations (object messages) on to the delegate for processing. In this way, the principal object acts as a container for aggregated behavior in the sense that it is responsible for starting, managing, and using the other classes that provide services to the container, which in turn provides the services of all its delegates to users or clients of the container. In addition, each delegate class can be used by multiple containers to provide the same services for different purposes.

The association between a container class and a delegate class is somewhat looser than the strict compile-time definition of a class hierarchy, in which each class in the hierarchy explicitly states its position within the hierarchy as its first statement. There is nothing inherent in a class, no statement in a class, that specifically defines it as a container or a delegate. However, note that because of strong typing, the relationships and dependencies between a container and its delegates are verified and enforced at compile-time; the compiler references the code for each delegate of a container in order to validate all method calls to the delegate from the container. Any variable that holds an object reference must be defined explicitly for the object's class type. Therefore, the compiler knows exactly what data members, properties, methods, and events can be accessed through the object reference to a delegate.

To support the running of behavior by procedures and other classes through a container, the container must define a `PUBLIC` stub method for each method that is implemented by a delegate. In other words, a container class simply using behavior in one of its delegates does not make that delegate behavior available to a container's client unless the container exposes the behavior through a method of its own.

The following example demonstrates delegation. In the example, only the container is shown. The behavior of the delegates is not represented. The container class and both delegate classes, called `logToDB` and `logToFile`, must provide an implementation of the methods `openLog( )`, `writeLog( )`, and `closeLog( )`. You can use two natural mechanisms to enforce this design—an interface or an abstract class.

The following code shows an interface definition for the example:

```
INTERFACE Ilog:

  METHOD PUBLIC VOID openLog (pcName AS CHARACTER).
  METHOD PUBLIC VOID writeLog (pcTxt AS CHARACTER).
  METHOD PUBLIC VOID closeLog ( ).

END INTERFACE.
```

The container class itself implements the method prototypes in the `Ilog` interface. As you can see from the following sample code, the container's version of these methods invokes the same method in one or the other of its two delegates, which are created by the constructor. The container also implements a separate `setMode( )` method to specify the delegate behavior to use, which is initially `logToFile`. For example:

```
CLASS Container IMPLEMENTS Ilog:

  DEFINE PRIVATE VARIABLE rlogToDB   AS CLASS logToDB   NO-UNDO.
  DEFINE PRIVATE VARIABLE rlogToFile AS CLASS logToFile NO-UNDO.
  /* logMode = 1 (File) is the default */
  DEFINE PRIVATE VARIABLE logMode    AS INTEGER INITIAL 1 NO-UNDO.

  CONSTRUCTOR PUBLIC Container ( ):
    rlogToDB = NEW logToDB ( ).
    rlogToFile = NEW logToFile ( ).
  END CONSTRUCTOR.

  METHOD PUBLIC VOID setMode (INPUT piLogMode AS INTEGER)
    logMode = piLogMode.
  END METHOD.

  METHOD PUBLIC VOID openLog (INPUT pcName AS CHARACTER):
    IF logMode EQ 1 THEN
      rlogToFile:openLog (pcName).
    ELSE
      rlogToDB:openLog (pcName).
  END METHOD.

  METHOD PUBLIC VOID writeLog (INPUT pcTxt AS CHARACTER):
    IF logMode EQ 1 THEN
      rlogToFile:writeLog (pcTxt).
    ELSE
      rlogToDB:writeLog (pcTxt).
  END METHOD.

  METHOD PUBLIC VOID closeLog ( ):
    IF logMode EQ 1 THEN
      rlogToFile:closeLog ( ).
    ELSE
      rlogToDB:closeLog ( ).
  END METHOD.

END CLASS.
```

The following code shows an abstract class definition for the example:

```
CLASS LogAbs ABSTRACT:

  METHOD PUBLIC ABSTRACT VOID openLog (pcName AS CHARACTER).
  METHOD PUBLIC ABSTRACT VOID writeLog (pcTxt AS CHARACTER).
  METHOD PUBLIC ABSTRACT VOID closeLog ( ).

END CLASS.
```

The container class itself derives from the `LogAbs` abstract class. As you can see from the following sample code, the container class is almost identical to the version that implements the `ILog` interface, except that the method implementations specify the `OVERRIDE` option. For example:

```
CLASS Container INHERITS LogAbs:
```

```
  DEFINE PRIVATE VARIABLE rlogToDB   AS CLASS logToDB   NO-UNDO.
  DEFINE PRIVATE VARIABLE rlogToFile AS CLASS logToFile NO-UNDO.
  /* logMode = 1 (File) is the default */
  DEFINE PRIVATE VARIABLE logMode    AS INTEGER INITIAL 1 NO-UNDO.

  CONSTRUCTOR PUBLIC Container ( ):
    rlogToDB = NEW logToDB ( ).
    rlogToFile = NEW logToFile ( ).
  END CONSTRUCTOR.

  METHOD PUBLIC VOID setMode (INPUT piLogMode AS INTEGER)
    logMode = piLogMode.
  END METHOD.

  METHOD PUBLIC OVERRIDE VOID openLog (INPUT pcName AS CHARACTER):
    IF logMode EQ 1 THEN
      rlogToFile:openLog (name).
    ELSE
      rlogToDB:openLog (name).
  END METHOD.

  METHOD PUBLIC OVERRIDE VOID writeLog (INPUT pcTxt AS CHARACTER):
    IF logMode EQ 1 THEN
      rlogToFile:writeLog (pcTxt).
    ELSE
      rlogToDB:writeLog (pcTxt).
  END METHOD.

  METHOD PUBLIC OVERRIDE VOID closeLog ( ):
    IF logMode EQ 1 THEN
      rlogToFile:closeLog ( ).
    ELSE
      rlogToDB:closeLog ( ).
  END METHOD.

END CLASS.
```

Comparison with procedure-based programming compile time checking

The use of containers and delegates in classes is very different from procedures, where a RUN statement for another external or internal procedure cannot be checked until run time. The unanticipated run-time error that occurs when one procedure mistakenly runs another becomes instead a compile-time error when you use classes, which can be corrected before the application is ever executed. Thus, when using procedures to implement delegation, note that:

- The design and functionality is not very different from using classes. The difference is in the level of compile-time checking.
- A procedure container can use class instances as delegates.

4

Program with class-based objects

Once an instance of a class is created, or static class members are initialized, you can access the data members, properties, methods, and events of the class.

For details, see the following topics:

- Instantiate and manage class-based objects
- Verify the type and validity of an object reference
- Use built-in system and object reference elements
- Assign object references
- Compare objects
- Use static members of a class
- Define and use widgets in classes
- Use preprocessor features in a class
- Raise and handle error conditions
- Reflection
- Use the Progress.Lang.ParameterList class
- Identify classes in the Progress.Reflect package
- Serialization and deserialization

Instantiate and manage class-based objects

Each class or interface essentially represents a unique data type (object type). ABL supports built-in and user-defined class and interface types in much the same way as built-in primitive types (such as INTEGER). ABL also allows you to manage class-based objects in a similar fashion to handle-based objects (such as procedure, buffer, or query objects), but also supports automatic garbage collection for class-based objects. Thus, you can use class or interface types to define the data types of variables, properties, parameters, and return types. You can also use these data elements as object references to class instances that you create and that ABL (by default) automatically deletes from a session when references to these objects no longer exist.

Similar to ABL handle-based objects, you must create an instance of a class and obtain its object reference before you can reference its data members, properties, methods, or events.

Note: Classes also support static members that are associated with the class type rather than a class instance. To access static members, it is not necessary to create an instance of the class. For more information on accessing static class members, see Access static members on page 186.

Unlike handle-based objects, you do not reference a class instance using a handle (HANDLE data type). Instead, you use an object reference, which in turn cannot appear in any statement or function that expects a HANDLE data type. You can use any appropriate class or interface data type to define a data element that holds an object reference to a class instance that you create. The particular object type of the object reference determines the class members that you can reference for the given class instance. You can also store the object reference in a temp-table, but not in a database table. The initial value of any object reference you define is the Unknown value (?), which you can change only by assigning another appropriate object reference value.

If you instantiate multiple instances of a class, the ABL session maintains multiple instances of that class. However, as happens with persistent procedure instances, multiple instances of the same class share r-code in memory with a separate data segment for each instance.

Define an object reference variable or property

This is the simplified syntax to define an object reference variable using the VAR statement:

```
VAR [ access-mode ]
    [ CLASS ] object-type-name [ [ [ constant ] ] ]
    object-reference.
```

This is the syntax to define an object reference variable using the DEFINE VARIABLE statement:

```
DEFINE [access-mode ] VARIABLE object-reference
  AS [ CLASS ]object-type-name[ EXTENT constant ][ NO-UNDO ] .
```

This is the simplified syntax to define an object reference property:

```
DEFINE [ access-mode ] PROPERTY object-reference
  AS [ CLASS ] object-type-name [ EXTENT constant ] [ NO-UNDO ]
  accessor-definitions .
```

For the complete syntax, see the DEFINE PROPERTY statement.

Element descriptions for this syntax diagram follow:

Note: For more information on each option, see the specified references in this book and also the DEFINE VARIABLE, VAR, and DEFINE PROPERTY statement reference entries in *ABL Reference*. For variables, no other options apply, because they are not supported for object references.

access-mode

The optional access mode specifies where and how the variable or property can be accessed, and the available options depend on where the variable is defined (as a class data member or property, within a procedure, etc.). For more information on accessing class data members and properties, see Access data members and properties on page 153.

object-reference

The name of a variable or property that will hold an object reference value.

CLASS

The CLASS keyword is required if *object-type-name* conflicts with an abbreviation for a built in ABL data type, such as INTE (INTEGER). Otherwise, it can optionally be used to clarify the readability of the statement.

object-type-name

The type name of a class or interface. This can be the fully qualified object type name or the unqualified class or interface name, depending on the presence of an appropriate USING statement in the class or procedure file. For more information on object type names, see Define and reference object type names on page 45. For more information on the USING statement, see Reference an object type name without its package on page 48.

EXTENT *constant*

An object type name defined with an extent is an array of object references to that type. In other words, each element of the array can contain an object reference to an instance of the specified type. If constant is zero, then the object reference is not an array.

NO-UNDO

If specified, prevents the variable or property value from being undone if a transaction that changes it is rolled back.

accessor-definitions

One or two property accessors that indicate if the property is readable, writable, or both.

For more information on defining:

- Variables as data members, see Define data members within a class on page 58
- Properties, see Define properties within a class on page 62

The following example shows a fragment of the `Main` class from the sample classes that are fully implemented in Sample classes on page 225. This fragment defines several private variables to hold object references and filenames:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  VAR PRIVATE CHAR              cOutFile.
  VAR PRIVATE CLASS CommonObj   rCommonObj.
  VAR PRIVATE CLASS CustObj     rCustObj.
  VAR PRIVATE CLASS CustObj     rCustObj2.
  VAR PRIVATE CLASS HelperClass rHelperClass.
  VAR PRIVATE CLASS IBusObj     rIBusObj.

  CONSTRUCTOR PUBLIC Main ( ):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass ( )

      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj ( )
      cOutFile     = "Customers.out".

    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.
  ...
END CLASS.
```

Create a class instance

You can use any one of the following four ABL elements to create an instance of a class:

- `NEW` function
- `NEW` statement
- `DYNAMIC-NEW` statement
- `New( )` method of the `Progress.Lang.Class` class

At run time, the given function, statement, or method locates the specified class type on the `PROPATH` using the package information in the class type name (or as specified in an appropriate `USING` statement at compile time) and invokes the specified constructor to create a new instance of the class. The `NEW` function returns an object reference to the class instance (class-based object) it creates, which you can use directly in an expression to access a member of the class without assigning the object reference to a variable. The `NEW` statement and the `DYNAMIC-NEW` statement each instantiate the class and assign its object reference to a variable in a single operation. The `New( )` method of the `Progress.Lang.Class` class returns an object reference to the instantiated class as a `Progress.Lang.Object`.

The basic difference between them is that the DYNAMIC-NEW statement and the New( ) method allow you to instantiate the class from information evaluated at run time, while both the NEW statement and NEW function require that you specify the class type using a type name that is resolved at compile time. The New( ) method also provides the flexibility of using run-time information obtained through reflection to specify the class and constructor parameters (if any) that it uses to instantiate the class.

NEW and DYNAMIC-NEW statements

This is the syntax for instantiating a class using either the NEW statement or the DYNAMIC-NEW statement:

Syntax

```
object-reference = { NEW class-type-name | DYNAMIC-NEW expression }
  ( [parameter[ , parameter]...] ) [ NO-ERROR ] .
```

Element descriptions for this syntax diagram follow:

object-reference

The name of a data element (for example, variable, writable property, or parameter) appropriately defined as a class or interface type.

This *object-reference* must be defined as one of the following types:

- The same class as the class type specified by *class-type-name* or *expression*
- A super class of the class type specified by *class-type-name* or *expression*
- An interface implemented by the class type specified by *class-type-name* or *expression*

Note: If *object-reference* is a temp-table field, its object type name can only be Progress.Lang.Object, the type name of the ABL root class. For more information on the ABL root class, see Use the root class: Progress.Lang.Object on page 91.

NEW *class-type-name*

Where *class-type-name* is the type name of the class to instantiate; it cannot be an interface type name. This can be the fully qualified type name or the unqualified class name, depending on the presence of an appropriate USING statement in the class or procedure file. For more information on forming type names for class-based objects, see Define and reference object type names on page 45. For more information on the USING statement, see Reference an object type name without its package on page 48.

DYNAMIC-NEW *expression*

Where *expression* is a character expression that evaluates, at run time, to the fully-qualified type name of the class to instantiate; it cannot be an interface type name. For more information on forming type names for class-based objects, see Define and reference object type names on page 45. The value of *expression* is not affected by any USING statements because USING statements only operate with a compile-time type name, as you provide for the NEW statement. The DYNAMIC-NEW statement is useful for OERA-compliant applications, where you want to instantiate a class whose type (specified in *expression*) is passed as a parameter to a method. Such a method allows you to instantiate one of a set of classes without the need to perform an IF or CASE statement test for

the identity of the class type. This works especially well when the possible class types specified by *expression* all implement the same set of interfaces.

[*parameter* **[** *, parameter* **]...]**

The parameters, if any, passed to a PUBLIC, PACKAGE-PROTECTED, PACKAGE-PRIVATE, or PROTECTED constructor defined by the *class-type-name* or *expression* class. If the class defines more than one PUBLIC, PACKAGE-PROTECTED, PACKAGE-PRIVATE, or PROTECTED constructor, the constructors are overloaded and these parameters identify the constructor to use for instantiating the class. For DYNAMIC-NEW, the constructors for all possible class types specified by *expression* must all take the same number of parameters defined with compatible data types and modes. For any one class instantiated by DYNAMIC-NEW, its instance constructors can be overloaded by the number of parameters, by the parameter data types or by the modes.

Whenever you create an instance of a class, the specified constructor of the instantiated class, as well as the specified constructors of any super classes in its class hierarchy, are run. The instantiated object also gets its own copy of PUBLIC, PACKAGE-PROTECTED, PACKAGE-PRIVATE, and PROTECTED data members and properties defined by all classes in its class hierarchy. That is, all instances of a class share a single set of methods defined for that class, but each instance has its own data not shared with any of the others. Thus, just as with persistent procedures, each instance of a class is a separate entity with its own instance data. For more information on how ABL constructs a class-based object, see Construct an object on page 116.

This example, from the Main class described previously creates instances of two sample classes in its constructor:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  DEFINE PRIVATE VARIABLE rCustObj     AS CLASS CustObj     NO-UNDO.
  DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
  ...

  CONSTRUCTOR PUBLIC Main( ):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass ( )

      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj ( )
      cOutFile  = "Customers.out".

    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.
  ...
END CLASS.
```

You can also use the NEW function in any expression outside of a NEW statement, without assigning the object reference, as long as the expression is appropriate for referencing the instantiated class. An example is when accessing a property on the object reference returned by the NEW function that returns a value that is data-type compatible with the expression.

For example, if you have a class that takes a default constructor, Deposit, and this class defines a DECIMAL property, Maximum, you can both instantiate the class and display the property value directly on the object reference returned by the NEW function in the MESSAGE statement that follows:

```
MESSAGE "The maximum deposit is " (NEW Deposit( )):Maximum VIEW-AS ALERT-BOX.
```

This Maximum property reference represents a chained reference on the result of the NEW function. Note that in order to reference a member of a class by chaining on the instantiating NEW function, you must bracket the function reference with parentheses and follow the closing parenthesis with the member reference. Note also that when you do not assign the object reference from a class instantiation, as in this example, ABL automatically deletes the instantiated class at some point after it is no longer needed in the expression.

New( ) method

You can use the New( ) method of the Progress.Lang.Class class to dynamically create an instance of a class when the class name and any parameters passed to it are only known at run time. This method returns a Progress.Lang.Object, which provides an instance of a class.

The New( ) method is one of several reflection methods of the Progress.Lang.Class class that provide type information about a class or interface at run time. For more information about the reflection capabilities of the Progress.Lang.Class class, see Use the Progress.Lang.Class class on page 205.

Two overloaded versions of the New( ) method are supported. You can use the first version when the constructor of the class does not take any arguments. This is the syntax for the New( ) method when no parameters are passed to its constructor:

Syntax

```
New( )
```

In this example of the first overloaded version, the type name of the class to be instantiated is passed in as a CHARACTER string, pcDynTypeName. The GetClass( ) method of the Progress.Lang.Class class is used to get the fully-qualified object type name of the pcDynTypeName-specified class. The constructor for the pcDynTypeName-specified class does not take any parameters, so the New( ) method is called without parameters.

```
DEFINE INPUT PARAMETER pcDynTypeName AS CHARACTER                    NO-UNDO.
VAR class Progress.Lang.Class rObjClass.
VAR class Progress.Lang.Object rDynBusObj.

rObjClass  = Progress.Lang.Class:GetClass(pcDynTypeName).
rDynBusObj = rObjClass:New( ).
```

Note: Because the New( ) method returns an instance of the Progress.Lang.Object class, the result of the New( ) method must be assigned to a Progress.Lang.Object object. You can then cast the object to the appropriate subclass, as shown in the next example using the CAST function. The New( ) method must be coded as a standalone statement and not part of an ASSIGN statement.

You can use the second overloaded version when the constructor of the class takes zero or more parameters. This is the syntax for the New( ) method when the constructor of its class takes zero or more parameters:

```
New( parameterlist-object )
```

Element descriptions for this syntax diagram follow:

parameterlist-object

An instance of the Progress.Lang.ParameterList class.

Parameters are passed to the New( ) method as a Progress.Lang.ParameterList object. Since a Progress.Lang.ParameterList object can be built with zero or more parameters, this version of the New( ) method is used to instantiate any class, even those classes whose constructor does not take any arguments. For more information about the Progress.Lang.ParameterList object, see *ABL Reference* and Use the Progress.Lang.ParameterList class on page 209.

In this example of the second overloaded version of the New( ) method, the rParamList parameter list object is passed to the pcDynTypeName-specified class constructor when the New( ) method is invoked:

```
DEFINE INPUT PARAMETER pcDynTypeName AS CHARACTER NO-UNDO.

VAR class integer iValue = 10.
VAR class BusinessObject rDynBusObj.
VAR class Progress.Lang.Class rObjClass.
VAR class Progress.Lang.ParameterList rParamList.

ASSIGN
  rParamList =NEW Progress.Lang.ParameterList(1)
  rObjClass  = Progress.Lang.Class:GetClass(pcDynTypeName).

rParamList:SetParameter(1, "INTEGER", "INPUT", iValue).
rDynBusObj = CAST(rObjClass:New(rParamList), BusinessObject).
```

The class instance referenced by rDynBusObj is not a Progress.Lang.Object object, but rather a BusinessObject. Thus, a cast is necessary for the compiler to allow the assignment of the New( ) method result into the subclass referenced by rDynBusObj. For more information on casting object references, see Object reference assignment and casting on page 179.

Call class-based methods

How you call a class-based method depends on whether you are calling it from within or from outside the class hierarchy where it is defined. This is similar to internal procedure and user-defined function calls, which must be specified differently when called within the external procedure where they are defined compared to when called from outside the procedure where they are defined. Within the class hierarchy that defines the method, you only have to call the method by name, because ABL implicitly recognizes the method within the environment where it is defined. The same is true with internal procedures and user-defined functions called within the defining procedure.

Note: This section describes method calling mechanisms that apply in common to both instance and static methods and to instance methods in particular. For information on the unique features of calling static methods, see Access static members on page 186.

With respect to another object, within whose class hierarchy a given instance method is defined, you must reference the object along with the method name in order to tell ABL what environment (class instance) contains the method definition. Similarly, when calling an internal procedure or user-defined function from outside of the defining procedure, you use special syntax to reference the procedure where the called routine is defined. Thus, like internal procedures and user-defined functions, the syntax and requirements for calling methods vary depending on where you call them relative to where they are defined.

No matter where you call a method (whether inside or outside the class hierarchy), most other requirements are the same. However, you must invoke any VOID method (that does not return a value) as a statement by itself, where the syntax of *method-call* depends on where you call the method:

```
method-call [ NO-ERROR ] .
```

For any method that returns a value, you have the option of calling it like a VOID method, as a statement by itself (ignoring the return value), or you can include the method call within a run-time expression as part of another statement that relies on the method return value, exactly like a user-defined function. As with user-defined function calls, non-VOID method calls can appear anywhere that a run-time expression can appear, such as a procedure or method INPUT parameter or in the expression of an Assignment (=) statement. For more information on the syntax and the basic requirements for a *method-call*, see Call methods from inside a class hierarchy where they are defined on page 142 and Call instance methods from outside a class hierarchy where they are defined on page 144.

All method calls (VOID or with a return type) are always synchronous. That is, there is no way to call a method asynchronously, as with procedures.

Also, unlike procedures, but exactly like user-defined functions, any run-time arguments passed to a method must, with certain exceptions, have data types that exactly match the data types of the corresponding parameters defined for the method. These exceptions include:

- Built-in data types that have an appropriate widening relationship with each other. Otherwise, ABL raises a compiler error. For example, you can pass an INTEGER value as an INPUT parameter defined as DECIMAL, because a DECIMAL parameter can hold all INTEGER values. However, you cannot pass a DATETIME variable as an INPUT parameter defined as DATE without raising a compiler error, because a DATE variable cannot hold the time-value part of the date returned by a DATETIME parameter.
- Dynamic and static temp-tables where the run-time schema of the dynamic temp-table matches the static temp-table definition. Otherwise, the AVM raises a run-time error.
- Dynamic and static ProDataSets where the run-time schema of the dynamic ProDataSet matches the static ProDataSet definition. Otherwise, the AVM raises a run-time error.
- Class or interface types that can appropriately represent one another. Otherwise, ABL raises a compiler error. For more information, see Pass object reference parameters on page 159.

  Note: If the parameter is a class or interface type, the object instance is passed by reference as an object reference. The effect of passing an object reference parameter is identical to assigning one object reference variable to another. For more information, see Define an object reference parameter on page 158.

- A literal Unknown value (?) passed for any built-in data type.
- An expression whose data type cannot be known until run time, when the parameter data types are validated. If they do not match, the AVM raises a run-time error.

The compiler verifies that the parameters passed in the method invocation are consistent with the parameters defined for the method. The compiler verifies that the number and mode of these parameters match exactly, and that the data types match appropriately. There is no implicit conversion of any data types when passing method parameters, except for those with appropriate widening relationships. However, provided that the run-time schemas match, ABL does allow a dynamic temp-table or ProDataSet to be passed to static temp-table or ProDataSet parameter (respectively), and similarly for passing a static temp-table or ProDataSet to a corresponding dynamic temp-table or ProDataSet parameter.

In addition to the data types, the number and mode of the parameters passed to a method must match the method's definition. In other words, the method signature of the called method must match the signature of a method definition. In fact, because multiple methods can be defined in any class hierarchy with the same name (overloaded), at a minimum, ABL must be able to uniquely match method calls and method definitions only by their respective signatures (parameter lists). Depending on the difference in method signatures, the appropriate method can be identified at either compile time or run time. Otherwise, ABL raises a compiler or run-time error, as appropriate. For more information on the criteria for passing parameters to methods, see the "Parameter passing syntax" in *ABL Reference*.

The following sections describe the syntax and requirements for calling methods inside or outside the class hierarchy where they are defined. For more information on calling class methods, see the Class-based method call reference entry in *ABL Reference*. For more information on calling overloaded methods, see Call an overloaded method or constructor on page 150.

Call methods from inside a class hierarchy where they are defined

You can directly access methods from within the class hierarchy where they are defined by invoking the method name. This includes all methods implemented directly within the class definition and all `PUBLIC`, `PACKAGE-PROTECTED`, `PACKAGE-PRIVATE`, and `PROTECTED` methods implemented in any super class of the class hierarchy. Invoking an overridden method from within a class hierarchy executes the method defined in the most derived subclass of a given class hierarchy. This is true even if the method is called from a super class that contains an overridden implementation. Thus, for an instance method, the actual implementation of a method invoked from within its own class definition depends on where that class definition resides in the class hierarchy of a given class instance.

You can also use additional ABL elements to access instance methods within the hierarchy that have the following characteristics:

- The method is defined, and possibly overridden, in one or more super classes. By default (as noted above), invoking the method always invokes the most-derived method in the class hierarchy. However, you can, instead, invoke the closest implementation in a super class of the current class definition by using the `SUPER` system reference.
- The method has the same name as a reserved keyword. By default, ABL does not allow you to invoke a method with such a name directly within the class hierarchy where it is defined, but you can invoke the method using the `THIS-OBJECT` system reference. You can also use `THIS-OBJECT` to improve code readability for an internal hierarchy method call.

This is the syntax to invoke a method from inside a class:

Syntax

```
[ data-element = ][[ SUPER ]|[ THIS-OBJECT ] : ]method-name
  ( [parameter[ , parameter]...] )
  [ NO-ERROR ]
```

Element descriptions for this syntax diagram follow:

[SUPER]

A system reference that allows you to call the closest overridden implementation of an instance method specified by *method-name* and an equivalent signature in the current class hierarchy. For more information, see SUPER system reference on page 176.

[THIS-OBJECT]

A system reference that allows you to call an instance method defined in the current class hierarchy when *method-name* is a reserved keyword. This system reference can also provide an aid to coding. For more information, see THIS-OBJECT system reference on page 175.

data-element

If the method returns a value that is assigned, the name of a variable, property, or other data element defined to hold any return value.

method-name

The name of an accessible method in the class hierarchy.

[*parameter* [, *parameter*] . . .]

The parameters, if any, of the method. For more information on the syntax of *parameter*, see "Parameter passing syntax" in *ABL Reference*.

[NO-ERROR]

Optionally redirects error processing when the method is called as a statement.

Note: For more information on accessing static methods from inside the class where they are defined, see Access static members from inside the defining class on page 189.

The following is an example from the sample class, `acme.myObjs.CustObj`, where `MessageHandler( )` is an abstract method defined in and inherited from the abstract class, `acme.myObjs.Common.CommonObj`, and implemented in `CustObj`:

```
USING acme.myObjs.Common.*.

CLASS acme.myObjs.Common.CommonObj ABSTRACT:
  DEFINE PUBLIC VARIABLE timestamp AS DATETIME NO-UNDO.

  METHOD PUBLIC VOID updateTimestamp( ):
    timestamp = NOW.
  END METHOD.

  METHOD PROTECTED ABSTRACT CLASS MsgObj MessageHandler
    (INPUT piObjType AS CHARACTER).

END CLASS.
```

The constructor in CustObj invokes the PROTECTED method MessageHandler( ) within its class hierarchy by directly calling the method by its name:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj
                          IMPLEMENTS IBusObj:
  ...
  DEFINE PRIVATE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
  ...
  CONSTRUCTOR PUBLIC CustObj( ):
    ...
    rMsg = MessageHandler("acme.myObjs.CustObj").
  END CONSTRUCTOR.

  METHOD PROTECTED OVERRIDE CLASS MsgObj MessageHandler
       (INPUT piObjType AS CHARACTER):
    RETURN NEW MsgObj (piObjType).
  END METHOD.

  ...
END CLASS.
```

Call instance methods from outside a class hierarchy where they are defined

You can access PUBLIC instance methods from outside the class hierarchy of a given object by using an object reference to qualify the method name. When you invoke a PUBLIC method on an object that is overridden in the object's class hierarchy, the AVM invokes the method in the most derived subclass that defines the method. A PACKAGE-PRIVATE method can be called from within the class and any class within its package. A PACKAGE-PROTECTED method can be called from within the class, any class within its package, and from within any subclass that inherits the class. There is no access to PRIVATE or PROTECTED methods from outside the class hierarchy.

This is the syntax to invoke a method from outside a class instance:

Syntax

```
[ return-variable = ]
  object-reference:method-name ( [ parameter [ , parameter ] ... ] )
  [ NO-ERROR ]
```

Element descriptions for this syntax diagram follow:

return-variable

If the method returns a value that is assigned, the name of the variable to put the return value into.

object-reference

An object reference whose class or interface type defines the method you are calling.

method-name

The name of a PUBLIC method defined somewhere in the class hierarchy of *object-reference*.

[*parameter* **[** *, parameter* **]...]**

The parameters, if any, of the method. For more information on the syntax of *parameter*, see "Parameter passing syntax" in *ABL Reference*.

NO-ERROR

Optionally redirects error processing when the method is called as a statement.

The following example shows two sample classes, where a method in acme.myObjs.CustObj calls the public Alert( ) and InfoMsg( ) methods in an instance of acme.myObjs.Common.MsgObj that is created for the CustObj class:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj IMPLEMENTS IBusObj:
  ...
  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.
  ...

  CONSTRUCTOR PUBLIC CustObj( ):
    ...
    rMsg = MessageHandler("acme.myObjs.CustObj").
  END CONSTRUCTOR.
  ...

  METHOD PUBLIC VOID CheckCredit ( ):
    IF VALID-OBJECT(rCreditObj) THEN DO:
      FOR EACH ttCustomer:
        rCreditObj:SetCurrentCustomer (ttCustomer.CustNum).
        rCreditObj:CheckCustCredit ( ).
        /* Invokes the CustCreditLimit property GET accessor */
        rMsg:InfoMsg (ttCustomer.Name + " is in good standing." +
          "  Credit Limit has been increased to " +
          STRING(rCreditObj:CustCreditLimit)).

        CATCH e AS Progress.Lang.AppError:
          IF e:ReturnValue = "Over Limit" THEN
            /* Invokes the CustCreditLimit property GET accessor */
            rMsg:Alert(ttCustomer.Name + " is on Credit Hold." +
              "  Balance exceeds Credit Limit of " +
              STRING (rCreditObj:CustCreditLimit)).
          ELSE
            rMsg:Alert("Customer not found").
        END CATCH.
      END. /* FOR EACH ttCustomer */
    END. /* VALID-OBJECT(rCreditObj) */
    ELSE rMsg:Alert ("Unable to check credit").
  END METHOD.
  ...
END CLASS.
```

In this case, rMsg is a private object reference data member that is initialized to an instance of MsgObj that is instantiated and returned from the MessageHandler( ) method of the CommonObj super class. (See the listing for this super class in Call methods from inside a class hierarchy where they are defined on page 142).

This MsgObj instance holds error and general message information specifically for the CustObj class that can be accessed using the Alert( ) and InfoMsg( ) methods implemented as follows:

```
CLASS acme.myObjs.Common.MsgObj:

  DEFINE PRIVATE VARIABLE cObjType AS CHARACTER NO-UNDO.

  CONSTRUCTOR PUBLIC MsgObj (INPUT pcObjType AS CHARACTER):
    cObjType = pcObjType.
  END CONSTRUCTOR.

  METHOD PUBLIC VOID Alert (INPUT ErrorString AS CHARACTER):
    MESSAGE "Error in " cObjType "!" SKIP
       ErrorString VIEW-AS ALERT-BOX ERROR.
  END METHOD.

  METHOD PUBLIC VOID InfoMsg (INPUT MsgString AS CHARACTER):
    MESSAGE MsgString VIEW-AS ALERT-BOX.
  END METHOD.

END CLASS.
```

Dynamically invoke a method at run time

Traditional class-based programming in ABL assumes that classes and their members are strongly typed at compile time. The AVM verifies the type and validity of class-based objects and class-based object references when the class is compiled.

There might be points in your application where the name of the class and the parameters to be passed to it are only known at run time. You can use either the built-in DYNAMIC-INVOKE function or the Invoke( ) method of the Progress.Lang.Class class to dynamically invoke a class-based method at run time.

DYNAMIC-INVOKE is a built-in ABL function that invokes a class-based method whose name is specified by a run-time expression, but whose parameters are defined at compile time. DYNAMIC-INVOKE takes an instance of an object, and an expression for the method name to be invoked, followed by a fixed list of compile-time, bound parameters. Consistent with the functionality provided by the DYNAMIC-NEW statement, DYNAMIC-INVOKE fully supports method overloading.

The Invoke( ) method of the Progress.Lang.Class class provides similar functionality to DYNAMIC-INVOKE. The advantage to the latter is that it has a fixed, compile-time parameter list and does not require the creation of a Progress.Lang.ParameterList object at run time. For more information about the Progress.Lang.ParameterList class, see *ABL Reference* and Use the Progress.Lang.ParameterList class on page 209.

Note: DYNAMIC-INVOKE does not operate on built-in classes. This is analogous to the DYNAMIC-FUNCTION function and the DYNAMIC-NEW statement, which also do not operate on built-in classes.

DYNAMIC-INVOKE can invoke either void or non-void, class-based methods. When invoking a non-void method, the caller can capture the returned value by assigning DYNAMIC-INVOKE to an expression of an appropriate type. This assignment operation follows the existing rules for assigning the return value of a method to an expression. If the caller chooses, the program does not need to capture the returned value. In contrast, when invoking a void method, the caller must not attempt to assign the method return value to an expression—doing so will cause the AVM to generate a run-time error. When invoking a void method with DYNAMIC-INVOKE, it is only permissible to do so with code that does not expect a return value.

This is the syntax for the DYNAMIC-INVOKE function:

Syntax

```
[ data-element = ] DYNAMIC-INVOKE( { class-type-name | object-reference }
  , method-name [ , parameter [ , parameter ] ... ] )
```

The DYNAMIC-INVOKE function accepts either a class type name or an object reference. A class type name is specified when the method to be invoked is static. An object reference is used to identify the object on which the method invocation is to occur.

In this example, the FooDec( ) method in the class instance referenced by rObj expects a DECIMAL value. With data type widening support, an ABL DECIMAL, INTEGER, or INT64 value can be passed to this method:

```
VAR DECIMAL dValue.
VAR INT iVal32.
VAR INT64 iVal64.
VAR INT iReturn.

/* Invoking non-overloaded FooDec (INPUT AS DECIMAL) */
iReturn = DYNAMIC-INVOKE(rObj, "FooDec", INPUT 5).
iReturn = DYNAMIC-INVOKE(rObj, "FooDec", INPUT dValue).
iReturn = DYNAMIC-INVOKE(rObj, "FooDec", INPUT iVal32).
iReturn = DYNAMIC-INVOKE(rObj, "FooDec", INPUT iVal64).
```

You can use the Invoke( ) method of the Progress.Lang.Class class to invoke a method when the method name and any parameters are only known at run time. This method returns an optional data element which is assigned the return value from the invoked, non-void method. This data element can be any data type. The AVM checks the return value at run time for data type compatibility with what is actually returned by the method.

The Invoke( ) method is one of several reflection methods of the Progress.Lang.Class class that provide type information about a class or interface at run time. For more information about the reflection capabilities of the Progress.Lang.Class class, see Use the Progress.Lang.Class class on page 205.

The Invoke( ) method has four overloaded constructors. You can use the first version to invoke a class-based static or instance method that does not take any parameters. This is the syntax for using this version of the Invoke( ) method:

```
[ data-element = ] class-reference:Invoke ( object-reference , method-name )
```

You can use the second overloaded version to invoke a class-based method that takes zero or more parameters. The parameter list is passed as a Progress.Lang.ParameterList object. This is the syntax for using this version of the Invoke( ) method:

```
[ data-element = ] class-reference:Invoke
  ( object-reference , method-name , parameterlist-object )
```

You can use the third overloaded version to invoke a static method that does not take any parameters (see the). This is the syntax for using this version of the `Invoke( )` method:

```
[ data-element = ]class-reference:Invoke ( method-name )
```

You can use the fourth overloaded version to invoke a static method that takes zero or more parameters (see the). The parameter list is passed as a `Progress.Lang.ParameterList` object. This is the syntax for using this version of the `Invoke( )` method:

```
[data-element = ]class-reference:Invoke
( method-name , parameterlist-object )
```

Element descriptions for these syntax diagrams follow:

data-element

An optional ABL data element which is assigned the return value from the invoked, non-void method. This data element can be any ABL data type that is compatible with the return value data type.

class-reference

An object reference for the `Progress.Lang.Class` instance containing the type information.

object-reference

A reference to the object that contains the method you want to invoke, and whose type is compatible with the *class-reference* object. When invoking a static method, the `Unknown value (?)` is passed for this parameter.

method-name

A CHARACTER expression that evaluates at run time to the name of the method to be invoked.

parameterlist-object

An instance of the `Progress.Lang.ParameterList` class. Since a `Progress.Lang.ParameterList` object can be built with zero or more parameters, this version of the `Invoke( )` method can be used to invoke any method, even if it does not take any arguments. See Use the Progress.Lang.ParameterList class on page 209 for a list of common public properties and methods of the `Progress.Lang.ParameterList` class.

Note: The `DYNAMIC-INVOKE` function and the `Invoke( )` method of the `Progress.Lang.Class` class provide similar functionality. They differ in that the former requires a fixed, compile-time parameter list, while the latter requires a instantiated and populated run-time `Progress.Lang.ParameterList` object.

The following example shows the Invoke( ) method overloading to call an instance method without parameters. It gets the MyClass class type using the GetClass( ) method of the Progress.Lang.Class class. An instance of this class type is created using the New( ) method of Progress.Lang.Class and casting the Progress.Lang.Object result to a MyClass object reference, which is then assigned to rObj. Invoke( ) then calls the MyDynMethod( ) method, specified as a character string, in rObj without parameters:

```
VAR CLASS Progress.Lang.Class rClass.
VAR CLASS MyClass rObj.

rClass = Progress.Lang.Class:GetClass("MyClass").
rObj    = CAST(rClass:New( ), MyClass).

rClass:Invoke(rObj, "MyDynMethod").
```

Note: For information on using the New( ) method with casting, see Create a class instance on page 136.

The following example shows the Invoke( ) method overloading to call an instance method with parameters. It creates an instance of MyClass class assigned to rObj. The rParamList object, an instance of the Progress.Lang.ParameterList class, holds the parameter data. The parameter data is passed to the MyDynMethod( ) method when it is invoked in the rObj object:

```
VAR Progress.Lang.Class rClass.
VAR MyClass rObj.
VAR Progress.Lang.ParameterList rParamList.

rClass     = Progress.Lang.Class:GetClass("MyClass").
rObj       = CAST(rClass:New( ), MyClass).

ASSIGN
  rParamList = NEW Progress.Lang.ParameterList(1)
  rParamList:NumParameters = 1.

rParamList:SetParameter(1, "INTEGER", "INPUT", 5).
rClass:Invoke(rObj, "MyDynMethod", rParamList).
```

The following example shows the Invoke( ) method overloading to call a static method without parameters. In this case, Invoke( ) is called directly on the instance of Progress.Lang.Class that is returned from GetClass( ) for the MyClass type, which allows it to invoke the specified static MyStaticMethod( ) method on that type:

```
VAR CLASS Progress.Lang.Class rClass.

rClass = Progress.Lang.Class:GetClass("MyClass").

rClass:Invoke("MyStaticMethod").
```

The following example shows the Invoke( ) method overloading to call a static method with parameters. Again, Invoke( ) is called on a Progress.Lang.Class instance representing the MyClass type, which allows it to invoke the static MyStaticMethod( ) method, this time, with the parameter list specified by rParamList:

```
VAR CLASS Progress.Lang.Class rClass.
VAR CLASS Progress.Lang.ParameterList rParamList.
```

```
ASSIGN
  rClass     = Progress.Lang.Class:GetClass("MyClass")
  rParamList = NEW Progress.Lang.ParameterList(1)
  rParamList:NumParameters = 1.

rParamList:SetParameter(1, "INTEGER", "INPUT", 5).
rClass:Invoke("MyStaticMethod", rParamList).
```

Comparison with procedure-based programming user-defined functions

Methods never require a separately declared prototype in the class where they are invoked, as user-defined functions sometimes do in the procedure or class where they are invoked.

Call an overloaded method or constructor

Calling an overloaded method or constructor is generally the same as invoking non-overloaded methods and constructors, by calling the method or instantiating the class using the correct number of parameters with matching data types and modes.

Note: The scope of a method (indicated by use of the `STATIC` option of the `METHOD` statement) does not participate in method overloading. This means that you cannot define both an instance method and a static method of a class with identical signatures. Otherwise, the rules for calling overloaded methods apply the same to both types of methods. However, only instance constructors can be overloaded because a class can have only one static constructor. For more information on static methods and constructors, see Define static members on page 190.

The following fragment of the `acme.myObjs.CustObj` sample class defines two overloadings of the `printObj( )` method:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj IMPLEMENTS IBusObj:
  ...
  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD ...
  ...

  /* First version of printObj prints a single copy of a report */
  METHOD PUBLIC VOID printObj ( ):
    OUTPUT TO PRINTER.
    DISPLAY dtTimestamp.
    FOR EACH ttCustomer:
      DISPLAY ttCustomer.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to printer").
  END METHOD.

  /* Second version of printObj takes an integer parameter representing the
     number of copies to print. */
  METHOD PUBLIC VOID printObj (INPUT piCopies AS INTEGER):
    VAR INT iCnt.

    OUTPUT TO PRINTER.
```

```
    IF piCopies <> 0 THEN
    DO iCnt = 1 TO ABS(piCopies):
      DISPLAY dtTimestamp.
      FOR EACH ttCustomer:
        DISPLAY ttCustomer.
      END.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated(STRING(piCopies)
                           + " copies of report sent to printer").
  END METHOD.
  ...
END CLASS.
```

The following fragment is a part of the `Main` sample class modified to invoke these `printObj( )` methods depending on a condition in an `IF` statement:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  VAR CLASS CustObj rCustObj.
  ...

  CONSTRUCTOR PUBLIC Main ( ):
    ...
    /* Create an instance of the CustObj class */
    rCustObj = NEW CustObj ( ).    ...
  END CONSTRUCTOR.

  ...

  /* ObjectInfo processes information about the Customer object    */
  METHOD PUBLIC VOID ObjectInfo (piInfoCount AS INTEGER):
    ...

    /* OUTPUT: An interface is used to receive a class that implements that
       interface */
    rHelperClass:ReportOutput (OUTPUT rIBusObj).
    IF piInfoCount <> ? AND piInfoCount > 1 THEN
      rIBusObj:printObj(piInfoCount).
    ELSE
      rIBusObj:printObj( ).
    ...
  END METHOD.
  ...
END CLASS.
```

ABL matches each overload of the method according to its parameters, where the first version takes no parameters and the second takes an `INTEGER` input parameter, which in this case is passed the value 3.

The following fragment of the `Main` sample class defines two constructors:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  VAR PRIVATE CHAR cOutFile.
  VAR PRIVATE CLASS CustObj rCustObj.
  VAR PRIVATE CLASS HelperClass rHelperClass.
```

```
  ...
  /* First constructor instantiates a Customer object */
  CONSTRUCTOR PUBLIC Main ( ):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass( )
      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj( )
      cOutFile     = "Customers.out".
    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.

  /* Second constructor takes a character parameter representing an input file
     of email addresses to instantiate a New England Customer object */
  CONSTRUCTOR PUBLIC Main (INPUT EmailFile AS CHARACTER):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass( )

      /* Create an instance of the NECustObj class */
      rCustObj     = NEW NECustObj(EmailFile)
      cOutFile     = "NECustomers.out".

    /* Subscribe OutputGenerated event handler for NECustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_NECustObjHandler).
  END CONSTRUCTOR.
  ...
END CLASS.
```

This is the driver procedure for the sample classes, `Driver.p`, which instantiates `Main` using each constructor:

```
/** This procedure drives the class example **/
VAR Class Main rClassExample.

/* Run the example for all Customers */
rClassExample = NEW Main ( ).
rClassExample:ObjectInfo (0).

/* Run the example for New England Customers */
rClassExample = NEW Main ("email.txt").
rClassExample:ObjectInfo (2).
```

ABL matches each overload of the constructor according to its parameters, where the first version takes no parameters and the second takes a CHARACTER input parameter, which in this case is passed the filename of a text file containing E-mail addresses.

In many cases, ABL easily matches overloaded method and constructor calls to their definitions at compile time. However for some overloading scenarios, ABL can match the correct method or constructor to call only at run time, depending on the modes, data types, and values of the parameters involved. For other scenarios, ABL must weigh several valid matches against one another to determine the most appropriate method or constructor to call.

For example, ABL matches an overloaded method call at run time when a method is passed a dynamic temp-table (TABLE-HANDLE) parameter and all available overloads of that method are defined to take different static temp-table (TABLE) parameters. ABL cannot determine what method to call at compile time because the passed TABLE-HANDLE parameter has no schema associated with it to match any of the corresponding static temp-table parameters defined for the available method overloads. However, at run time, when the TABLE-HANDLE parameter is passed with an associated temp-table, ABL can examine the schema of this temp-table to determine which of the available method overloads takes a static temp-table parameter with a schema that matches the schema of the passed temp-table. If it finds a match, ABL then invokes the matching method. Otherwise, ABL raises a run-time error.

This kind of run-time matching of overloaded method or constructor calls provides a powerful mechanism, similar to polymorphism, that you can use to invoke an appropriate method or constructor that might otherwise require additional code to identify.

For more information on defining overloaded methods and constructors, see Define overloaded methods and constructors on page 113. For more information on finer points of method and constructor overloading and how ABL handles some of the more complex overloading scenarios, see Overloaded method and constructor call scenarios on page 253.

Access data members and properties

You can directly access data members and properties from within the class where they are defined by using the data member or property name anywhere you can use a variable. This includes all data members or properties defined directly within the class definition and all PUBLIC,PACKAGE-PROTECTED, PROTECTED, and PACKAGE-PRIVATE data members or properties defined in any super class of the class hierarchy. PACKAGE-PRIVATE data members and properties which are defined in any super class of the class hierarchy can be accessed directly if the defining class and the accessing class are in the same package. For properties, access also depends on whether they are defined as readable or writable, or both.

Note: This section describes data member and property access mechanisms that apply in common to both instance and static data elements and to instance data elements in particular. For information on the unique features of accessing static data members and properties, see Access static members on page 186.

You can access a PUBLIC instance data member or property from outside the class hierarchy of an object where it is defined by using an object reference to qualify the data member or property name. Also, note that only variable data members and properties can be PUBLIC.

While you can also define non-public properties, a primary benefit of properties is to encapsulate non-public data members. Thus, because properties can be defined for many variable data types, PUBLIC properties are well-suited for encapsulating variable data members.

There is no direct access to PRIVATE or PROTECTED data members or properties from outside the class hierarchy, although an instance can access a PRIVATE member of another instance if they are the same class, and an instance can access the PROTECTED member of a second instance that is at the same level or higher in the class hierarchy. The following code demonstrates how an instance can access the PRIVATE data member of another instance of the same class.

```
/* Defines a class with one private data member and one public method*/
CLASS PrivateNumber:

  VAR PRIVATE INT iMyNumber.

  CONSTRUCTOR PUBLIC PrivateNumber(INPUT piNum AS INTEGER):
    iMyNumber = piNum.
  END CONSTRUCTOR.

  METHOD PUBLIC VOID ShowNumbers(INPUT pOtherInstance AS CLASS PrivateNumber):
    MESSAGE "My number is" STRING(iMyNumber) SKIP
    "The other instance's number is" STRING(pOtherInstance:iMyNumber)
    VIEW-AS ALERT-BOX.
  END METHOD.

END CLASS.
```

```
/* Creates two instances of PrivateNumber and invokes ShowNumbers*/
VAR PrivateNumber InstanceOne = NEW PrivateNumber(1).
VAR PrivateNumber InstanceTwo = NEW PrivateNumber(2).

InstanceOne:ShowNumbers(InstanceTwo).
```

The constructor of PrivateNumber takes an integer as input, which it then assigns to the PRIVATE data member iMyNumber. The ShowNumbers method takes an object as input and then displays iMyNumber for both the current instance and the instance passed to it. Because instances of the same class can access each other's private data members, the invocation of ShowNumbers for InstanceOne with InstanceTwo as a parameter will display the private data members of both instances.

If you want to expose a non-variable (such as a buffer, temp-table, query, or ProDataSet) to other classes, you must do so using one of these mechanisms:

- Implement PUBLIC methods that pass the data object as a parameter.
- Define a PUBLIC HANDLE variable, data member, or property that allows access to the data object through its handle. Note that handle data members inherently undermine encapsulation because they provide direct access to the data objects they reference. To encapsulate access to all non-variable data members, including data objects, define non-PUBLIC and static versions of these data members and pass them as parameters (if possible) to PUBLIC methods of the class.

Note: Some data objects, such as query objects, can be defined as static data objects, but can only be passed as parameters using their handles.

Reference a public data member or property outside of the object that defines it

This is the syntax for referencing a PUBLIC instance data member or property from outside the object where it is defined:

Syntax

```
object-reference:data-member-or-property-name
```

Element descriptions for this syntax diagram follow:

object-reference

An object reference to a class instance whose class hierarchy defines the PUBLIC data member or property.

data-member-or-property-name

The name of a PUBLIC data member or property defined somewhere in the class hierarchy of *object-reference*.

For more information on accessing data members and properties, see the Class-based data member access and Class-based property access reference entries in *ABL Reference*.

The following fragment from the acme.myObjs.CustObj sample class shows how you can access a PUBLIC property from outside the class instance (acme.myObjs.CreditObj) where it is defined:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj
                          IMPLEMENTS IBusObj:
  ...
  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.

  CONSTRUCTOR PUBLIC CustObj ( ):
     rCreditObj = NEW CreditObj ( ).
    ...
    rMsg = MessageHandler("acme.myObjs.CustObj").
  END CONSTRUCTOR.

  ...

  METHOD PUBLIC VOID CheckCredit ( ):
    IF VALID-OBJECT(rCreditObj) THEN DO:
      FOR EACH ttCustomer:
        ...
        /* Invokes the CustCreditLimit property GET accessor */
        rMsg:InfoMsg(ttCustomer.Name + " is in good standing." +
          "  Credit Limit has been increased to " +
          STRING(rCreditObj:CustCreditLimit)).

        CATCH e AS Progress.Lang.AppError:
          IF e:ReturnValue = "Over Limit" THEN
            /* Invokes the CustCreditLimit property GET accessor */
            rMsg:Alert (ttCustomer.Name + " is on Credit Hold." +
              "  Balance exceeds Credit Limit of " +
              STRING (rCreditObj:CustCreditLimit)).
          ELSE
            rMsg:Alert ("Customer not found").
        END CATCH.
      END. /* FOR EACH */
    END.
    ELSE rMsg:Alert ("Unable to check credit").
  END METHOD.
```

```
  ...
END CLASS.
```

In the previous example, `CustCreditLimit` is a publicly readable property defined in the `CreditObj` class. However, its value is protected and can only be written from within its defining class hierarchy, because the property's `SET` accessor is defined as `PROTECTED`. For more information on property accessors, see Define properties within a class on page 62.

Dynamically access a property at runtime

You can access a class property dynamically using the `DYNAMIC-PROPERTY` function or statement by providing a character expression for the property name, along with an object reference, `THIS-OBJECT` system reference, or a class type. The compiler allows the object reference to be of any user-defined type or .NET type.

At runtime, the AVM performs the following:

- Determines the actual type of the object reference
- Resolves the property name expression
- Validates and performs the assignment.

The following syntax is for the `DYNAMIC-PROPERTY` function:

Syntax

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

The following syntax is for the `DYNAMIC-PROPERTY` statement. Through the `DYNAMIC-PROPERTY` statement you can set a property's value without knowing the type of the property's name at compile time:

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

`data-element`

Specifies an ABL data element which is assigned a value returned when you invoke the class property's `GET` accessor. The appropriate `CAST` or conversion function is applied to convert the property's value to the data type of *`data-element`*.

`object-reference`

Specifies a reference to an ABL or .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 an ABL or .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 only be class type names: interfaces are not valid.

property-name

Specifies a CHARACTER expression that evaluates to the property name at runtime.

index

Specifies an integer value for the index of the specified element. Use *index* to set or retrieve an individual array element. You can also operate on an entire array by removing the index similar to working with non-dynamic property access.

new-value

An expression of any data type. Its value is converted at run time to the property's data type.

ABL also provides the GetPropertyValue( ) and SetPropertyValue( ) methods with functionalities similar to the DYNAMIC PROPERTY function. Use the GetPropertyValue( ) and SetPropertyValue( ) methods to access and set a property's value at runtime, when you do not know the property's name or type at compile-time. There is a difference between using GetPropertyValue( ) or SetPropertyValue( ) and using 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 conversion function such as STRING or INTEGER used.

The following code snippets compare the dynamic and non-dynamic ways of accessing a property.The following ABL code snippet shows non-dynamic access of the property:

```
VAR Invoice objref = NEW Invoice().
objref:Amount = 654.32.   /* set the Amount property */
MESSAGE objref:Amount.    /* get the Amount property */
```

In the example above, 654.32 is set as the value for Amount property at compile-time.

The following code uses DYNAMIC-PROPERTY:

```
VAR Invoice objref = NEW Invoice().

/* set the Amount property */
DYNAMIC-PROPERTY(objref, "Amount") = 654.32.

/* get the Amount property */
MESSAGE DYNAMIC-PROPERTY(objref, "Amount").
```

In the example above, setting and getting values for the Amount property is resolved dynamically through the DYNAMIC-PROPERTY statement.

The following code snippet demonstrates the use of `Progress.Lang.Class`'s `SetPropertyValue( )` and `GetPropertyValue( )` methods:

```
VAR INVOICE objref = NEW Invoice().
VAR Progress.Lang.Class classref.
classref = objref:GetType().

/* set the Amount property */
classref:SetPropertyValue (objref, "Amount", 654.32).

/* get the Amount property */
MESSAGE classref:GetPropertyValue(objref, "Amount").
```

In the example above, setting and getting values for the `Amount` property is resolved at runtime through the `SetPropertyValue( )` and `GetPropertyValue( )` methods.

Define an object reference parameter

Procedures, user-defined functions, and class-based methods can all define object reference parameters (as class or interface types). When an object is passed locally, class instances are passed by reference using object references. Thus, only a reference to an object is passed, not the object itself. When an object is passed between an application server and an ABL client, the object is serialized by the sender, and the receiver uses this information to create a copy of the object.

Note: To pass an object reference parameter between an application server and an ABL client, the object's class must be defined on both the client and application server, and the class must be marked as serializable. For a full list of the restrictions on passable classes, see Pass object reference parameters on page 159.

This is the syntax to define an object reference parameter for a method in a class or a user-defined function in a procedure:

Syntax

```
[ INPUT | INPUT-OUTPUT | OUTPUT ]parameter-name
  AS [CLASS]object-type-name
```

This is the syntax to define an object reference parameter for a procedure:

```
DEFINE [ INPUT | INPUT-OUTPUT | OUTPUT ] PARAMETER parameter-name
  AS [CLASS]object-type-name ] .
```

Descriptions of the object reference parameter syntax elements (bolded) follow:

parameter-name

The name of the parameter representing the object reference.

CLASS

The CLASS keyword is required if *object-type-name* conflicts with an abbreviation for a built in ABL data type, such as INTE (INTEGER). Otherwise, it can optionally be used to clarify the readability of the statement.

object-type-name

The class or interface type name of an object reference to be passed. This can be the fully qualified object type name or the unqualified class or interface name, depending on the presence of an appropriate USING statement in the class or procedure file. For more information on object type names, see Define and reference object type names on page 45. For more information on the USING statement, see Reference an object type name without its package on page 48.

For more information on the full syntax for parameter definitions, see the "DEFINE PARAMETER" statement and "Parameter definition syntax" reference entries in *ABL Reference*.

Pass object reference parameters

When an application passes an object reference to a method, user-defined function, or procedure, the assignment rules can be summarized by designating the object that provides the reference as the source and the object that receives the reference as the target. The target object reference definition can be:

- The same type as the source (for example, to acme.myObjs.CustObj from acme.myObjs.CustObj)
- A super class of the source (for example, to acme.myObjs.Common.CommonObj from acme.myObjs.CustObj)
- An interface that the source implements (for example, to acme.myObjs.Interfaces.IBusObj from acme.myObjs.CustObj)

For more information on the rules for assigning object references, see Assign object references on page 177.

To pass an object between an application server and an ABL client, the class of the object must be defined on both the sending and receiving side, and the definitions of the class's data members, properties, events, and method signatures must match exactly. The class is serialized by the sender, and the receiver uses that information to create a copy of the object.

The following restrictions apply to objects being passed as parameters (or thrown as errors) between an application server and client:

- In the case of a user-defined class, the object's class and all of the classes in its hierarchy must be marked as SERIALIZABLE. For more information on marking a class as SERIALIZABLE, see Use the CLASS construct on page 54.
- The content of static data members is not serialized, and the state of queries, buffers, open files, streams, and event subscriptions, for example, are not maintained.
- All of the object's data members that are defined as class-based objects must be of a class type that is also marked SERIALIZABLE. (This restriction does not apply to static data members that are defined as objects, because static data members are not serialized.)
- Handle-based variables are serialized, but no information for reconstructing handle-based objects on the receiving side is serialized.
- MEMPTRs assigned by an ABL application are serialized, but MEMPTRs from an external source (such as a DLL or shared library) are not serialized.

- Statically defined temp-tables and ProDataSets in user-defined classes are serialized, except for REFERENCE-ONLY tables.
- The REJECTED, ERROR, ERROR-STRING, and DATA-SOURCE-MODIFIED attributes for temp-tables in an object are maintained as part of the deserialization process. Similarly, the REJECTED and ERROR attributes for ProDataSets are maintained during serialization.
- Not all built-in classes are serializable. See the CLASS statement entry in *ABL Reference* for a full list of serializable built-in classes.
- .NET and ABL-extended .NET objects cannot be passed as parameters.

The following examples demonstrate the passing of object reference parameters for INPUT, INPUT-OUTPUT, and OUTPUT. These examples highlight code from the Main class shown previously (see Define an object reference variable or property on page 134) and related sample classes that are described and fully listed in Sample classes on page 225. Refer to these samples for referenced class listings that are not shown in this section.

When passing an INPUT parameter, the caller is the source of the assignment and the invoked method is the target of the assignment. Therefore, the caller must pass an INPUT object reference that is the same class or interface type as the defined method parameter, that is a subclass of the class defined by method parameter, or that is a class which implements the interface defined by the method parameter.

If you pass a super class or interface object reference to the invoked method, the method can only use that object reference to call those methods and access data that are defined by the specified super class or interface, even if the referenced object actually represents a subclass or interface-implementing class that defines additional public methods and data. If the invoked method attempts to use this object reference to call these additional methods or access the additional data without first casting the object reference appropriately, the compiler generates an error. For more information on casting object references, see Object reference assignment and casting on page 179.

The following examples pass an INPUT parameter:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:

  VAR PRIVATE CLASS CustObj rCustObj.
  ...
  VAR PRIVATE CLASS HelperClass rHelperClass.
    ...

  METHOD PUBLIC VOID ObjectInfo( ):
    /* Demonstrates passing object references as parameters */
    /* INPUT: It is valid to pass a subclass to a method defined to take a
       super class */
    rHelperClass:InitializeDate (rCustObj).    ...
  END METHOD.

  ...

END CLASS.
```

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.Common.HelperClass:
```

```
  ...
  VAR PRIVATE CLASS MsgObj rMsg.
  ...

  METHOD PUBLIC VOID InitializeDate (INPUT prObject AS CLASS CommonObj):
    /* Timestamp this object */
    IF VALID-OBJECT(prObject) THEN
      prObject:updateTimestamp ( ).    ELSE
     rMsg:Alert ("Not a valid object").
  END METHOD.

  ...

END CLASS.
```

The previous example shows an `acme.myObjs.CustObj` instance passed as input to the `InitializeDate( )` method, which is defined to take an `acme.myObjs.Common.CommonObj` (super class of `CustObj`). Note that the `updateTimestamp( )` method invoked on the input object reference only exists in `CommonObj`. (For a listing of this class, see Call methods from inside a class hierarchy where they are defined on page 142.) This method can be invoked on any object that is a subclass of `CommonObj`. For more information on `acme.myObjs.Common.CommonObj`, see Sample classes on page 225.

When you pass an object reference data element as an `INPUT-OUTPUT` parameter, it must have the same class or interface type as the corresponding parameter definition. Because the object is being passed in both directions, its type must match the parameter definition exactly.

The following examples pass an `INPUT-OUTPUT` parameter:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:

  VAR PRIVATE CLASS CustObj rCustObj.
  VAR PRIVATE CLASS HelperClass rHelperClass.
  ...

  METHOD PUBLIC VOID ObjectInfo( ):
    /* Demonstrates passing object references as parameters */
    ...

    /* INPUT-OUTPUT: Must be an exact match, a class to a method defined to
       take that same class type */
    rHelperClass:ListNames(INPUT-OUTPUT rCustObj).    rCustObj:CheckCredit( ).    ...
  END METHOD.

  ...

END CLASS.
```

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.Common.HelperClass:

  VAR PRIVATE CLASS CustObj rCustObj.
  ...
```

```
  METHOD PUBLIC VOID ListNames (INPUT-OUTPUT prCustObj AS CLASS CustObj):
    VAR INT idx.

    DO idx = 1 to prCustObj:iNumCusts:
      CREATE ttNames.
      ttNames.CustName = prCustObj:GetCustomerName (idx).
    END.
    rCustObj = prCustObj.
  END METHOD.

END CLASS.
```

The previous example shows an `acme.myObjs.CustObj` instance passed as an `INPUT-OUTPUT` parameter to the `ListNames( )` method in `acme.myObjs.Common.HelperClass`. In this case, the type on both sides (the passed object reference and the parameter definition) have to be exactly the same, in this case `CustObj`. The `ListNames( )` method uses the input object reference and stores the value in its `PRIVATE` data member, `rCustObj`.

When passing an `OUTPUT` parameter, the caller is the target of the assignment and the invoked method is the source of the assignment. Therefore, the caller must pass a parameter that is the same class or interface type as the defined method parameter, that is a super class of the class defined by method parameter, or that is an interface which is implemented by the class defined by the method parameter.

Once again, if the caller passes a super class or interface object reference data element for `OUTPUT`, the caller can only use that object reference to invoke those methods and access data that are defined by the specified super class or interface, even if the referenced object actually represents a subclass or interface-implementing class that defines additional public methods and data. If the caller attempts to use this object reference to call these additional methods or access the additional data without first casting the object reference appropriately, the compiler generates an error. For more information on casting object references, see Use the CAST function on page 180.

The following examples pass an `OUTPUT` parameter:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:

  VAR PRIVATE CHAR cOutFile.
  VAR PRIVATE CLASS HelperClass rHelperClass.
  VAR PRIVATE CLASS IBusObj rIBusObj.
  ...

  METHOD PUBLIC VOID ObjectInfo( ):
    /* Demonstrates passing object references as parameters */
    ...

    /* OUTPUT: An interface is used to receive a class that implements that
       interface */
    rHelperClass:ReportOutput (OUTPUT rIBusObj).
    IF piInfoCount <> ? AND piInfoCount > 1 THEN
      rIBusObj:printObj(piInfoCount).
    ELSE
      rIBusObj:printObj( ).
      rIBusObj:logObj (outFile).
      ...
  END METHOD.
```

```
  ...
END CLASS.
```

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.Common.HelperClass:
  VAR PRIVATE CLASS CustObj rCustObj.
  VAR PRIVATE CLASS MsgObj rMsg.
  ...

  METHOD PUBLIC VOID ReportOutput (OUTPUT prInterface AS CLASS IBusObj):
    /* Send the PRIVATE CustObj instance back to be printed */
    IF VALID-OBJECT(rCustObj) THEN
      prInterface = rCustObj.
    ELSE
      rMsg:Alert("The object is not valid").
  END METHOD.
END CLASS.
```

In the previous example, `ReportOutput( )` defines an `OUTPUT` parameter defined as an object reference to the `acme.myObjs.Interfaces.IBusObj` interface. Through the `OUTPUT` parameter, this method in `acme.myObjs.Common.HelperClass` returns a `PRIVATE` object reference to an `acme.myObjs.CustObj`, which is initially set to a `CustObj` instance by the `ListNames( )` method of the class. This works because the `CustObj` class implements the `IBusObj` interface. Therefore, `ReportOutput( )` can return any object that implements `IBusObj`. Note that after `ReportOutput( )` returns in `ObjectInfo( )`, the `logObj( )` method is called on the `OUTPUT` object reference. This method is declared in the `IBusObj` interface and implemented in `CustObj`. It would be invalid to invoke a method on the `OUTPUT` object reference that is not declared in this interface, even though the original `CustObj` instance passed as an `IBusObj` defines additional `PUBLIC` methods.

Define an object reference return type

User-defined functions and class methods can define return types as object references (as class or interface types). When an object is passed locally, class instances are always returned by reference. Thus, only a reference to an object is returned, not the object itself. When an object is passed between an application server and an ABL client, the object is serialized by the sender, and the receiver uses this information to create a copy of the object. See Pass object reference parameters on page 159 for more information.

This is the syntax to define an object reference return type for a method:

Syntax

```
METHOD { PRIVATE | PACKAGE-PRIVATE | PROTECTED | PACKAGE-PROTECTED | PUBLIC }
[ STATIC | ABSTRACT ][ OVERRIDE ][ FINAL ][CLASS]
object-type-namemethod-name ( [parameter[ , parameter]...] ) :
  [method-body]
```

This is the syntax to define an object reference return type for a user-defined function:

```
FUNCTION function-name RETURNS [CLASS]object-type-name
  ( [parameter[ , parameter]...] ) :
  [function-body]
```

Descriptions of the object reference return type syntax elements (bolded) follow:

CLASS

The CLASS keyword is required if *object-type-name* conflicts with an abbreviation for a built in ABL data type, such as INTE (INTEGER). Otherwise, it can optionally be used to clarify the readability of the statement.

object-type-name

The class or interface type name of the object reference to be returned. This can be the fully qualified object type name or the unqualified class or interface name, depending on the presence of an appropriate USING statement in the class or procedure file. For more information on object type names, see Define and reference object type names on page 45. For more information on the USING statement, see Reference an object type name without its package on page 48.

For more information on the syntax of these statements, see the METHOD statement and FUNCTION statement reference entries in *ABL Reference*.

The following code fragment, from the sample class `acme.myObjs.CustObj`, illustrates a method returning a class return type, in this case, to initialize a message object (`acme.myObjs.Common.MsgObj`) for use by the current instance of `CustObj`:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj IMPLEMENTS IBusObj:
  DEFINE PRIVATE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
  ...

  CONSTRUCTOR PUBLIC CustObj( ):
    ...

    iNumCusts = 0.
    /* Fill temp table and get row count */
    FOR EACH Customer NO-LOCK WHERE Customer.CreditLimit > 50000:
      CREATE ttCustomer.
      ASSIGN
        iNumCusts          = iNumCusts + 1
        ttCustomer.RecNum  = iNumCusts
        ttCustomer.CustNum = Customer.CustNum
        ttCustomer.Name    = Customer.Name
        ttCustomer.State   = Customer.State.
    END.
    rMsg = MessageHandler ("acme.myObjs.CustObj").
  END CONSTRUCTOR.

  METHOD PROTECTED OVERRIDE CLASS MsgObj MessageHandler
    (INPUT iObjType AS CHARACTER):
    RETURN NEW MsgObj (iObjType).
  END METHOD.
  ...
END CLASS.
```

The `MessageHandler( )` method definition implements an abstract method defined in the immediate super class. This method instantiates the `MsgObj` class and returns the object to the caller.

ABL treats a return value from a class method the same as an assignment. That is, the compiler verifies that the object reference being returned by the method is consistent with the variable to which the object reference is being assigned. For more information on the compatibility rules for assigning object references, see Assign object references on page 177.

Thus, the rules for assigning an object reference return value from within a method or user-defined function are the same as for assigning an `OUTPUT` object reference parameter from within a method. (For more information on passing object references as `OUTPUT` parameters, see Pass object reference parameters on page 159.) The caller of the method must define a data element to receive the returned object reference that is the same class as the return object reference, a super class of the returned object reference, or an interface of the returned object reference. As with parameters, if the caller has as its target an object reference variable defined to reference a super class or interface, the caller can only invoke those methods and access data that are defined by the specified super class or interface. If the caller attempts to call a method or access data that is not defined in the super class or attempts to call a method not defined by the interface, the compiler generates an error.

Define an object reference field in a temp-table

You can define temp-table fields as the class type, `Progress.Lang.Object`, the built-in class that is the implicit super class (root class) of all other classes in ABL. This allows a temp-table field to hold object references of any class type. When object references are assigned to the field, they are implicitly cast to the root class, `Progress.Lang.Object`. When you want to use the object reference as a specific object type, you must cast the object reference to the specific type. For more information on casting, see Object reference assignment and casting on page 179.

This is the syntax to define a class field in a temp-table:

Syntax

```
FIELD field-name AS [ CLASS ] Progress.Lang.Object
```

Element description for this syntax diagram follow:

field-name

The name of a field defined as the ABL root class.

Note: You cannot define a class field in an OpenEdge database table.

The following example defines a temp-table field to hold a class instance, by defining the field as `Progress.Lang.Object`. In this case, the purpose is to store different `ShapeClass` instances in order to calculate an area on each `ShapeClass` object in sequence, according to its subclass type (`RectangleClass` or `CircleClass`), as shown:

```
USING Progress.Lang.*.

CLASS Main:
  VAR DECIMAL length = 5.0.
  VAR DECIMAL radius = 100.0.
  VAR DECIMAL width = 10.0.

  DEFINE TEMP-TABLE myTT NO-UNDO
    FIELD Shape AS CLASS Object
    FIELD Area  AS DECIMAL.

  CONSTRUCTOR PUBLIC Main ( ):
    CREATE myTT.
    myTT.Shape = NEW RectangleClass (width, length).

    CREATE myTT.
    myTT.Shape = NEW CircleClass (radius).

    FOR EACH myTT:
      /* Cast the field to the common shape super class, ShapeClass */
      displayArea(INPUTCAST(myTT.Shape, ShapeClass), OUTPUT myTT.Area).
    END.
  END CONSTRUCTOR.

  METHOD PUBLIC VOID displayArea
    (INPUT rShape AS CLASS ShapeClass, OUTPUT dArea AS DECIMAL):
    dArea = rShape:calculateArea( ).
    MESSAGE dArea VIEW-AS ALERT-BOX.
  END METHOD.
END CLASS.
```

This previous example is a modified version of the `Main` class used to demonstrate polymorphism (see Use polymorphism with classes on page 125). This version of the `Main` class shows how you might use temp-table fields to support polymorphic object references and use the `CAST` function to cast down to the common functionality of the polymorphic super class (`ShapeClass`).

Publish and subscribe to class events

As described previously (see Events of a class on page 52), class events allow you to dynamically register one or more event handlers that execute in response to run-time conditions. Thus, you publish the event to notify other parts of the application when a given condition is detected, and one or more event handlers execute in response to this notification. Depending on run-time conditions, you can change the number of event handlers that execute for any given event. In this way, different parts of the application can respond to the same event at different times whenever the event is published. For example, one common use of this mechanism is to help populate one or more data viewers from record buffers whose contents have been updated from a selection in a data browser.

The procedure for using a class event to provide such notifications follows a common pattern.

To set up and use a class event:

1. Define the event in a class definition using the `DEFINE EVENT` statement (see Define events within a class on page 77).
2. Subscribe one or more handlers to the event using the `Subscribe( )` event method (see Specify handler subscriptions for class events on page 168).
3. Publish the event from within the class that defines it for a given condition using the `Publish( )` event method, which runs all event handlers that are subscribed to the event, and processes any values returned from the `Publish( )` method parameters (see Publish class events on page 170).

Note: This section describes event subscription and publishing mechanisms that apply in common to both instance events and static events, but to instance events in particular. For information on the unique features of working with static events, see Access static members on page 186.

Specify handler subscriptions for class events

ABL allows you to subscribe an internal procedure or a class-based method as a handler for a given class event, and it also allows you to unsubscribe the handler. The effect of this process is similar to subscribing and unsubscribing internal procedures to named events using the `SUBSCRIBE` and `UNSUBSCRIBE` statements. However, instead of using these ABL statements for class events, you invoke the ABL built-in event methods `Subscribe( )` and `Unsubscribe( )`, using a syntax that is more natural for object-oriented programming and that supports strong typing when you subscribe methods as event handlers. This syntax also ensures that the event you are referring to is actually a member of the specified class-based object. The return type for these methods is void; so, you must invoke them as a statement (not in an expression).

This is the ABL syntax to manage class event handler subscriptions:

Syntax

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

Thus, the `Subscribe( )` method subscribes the specified method or internal procedure as a handler for the event specified by *`event-name`*, and the `Unsubscribe( )` method removes the specified method or internal procedure as a handler for the event specified by *`event-name`*.

The specified event (*`event-name`*) can be a member of any specified class instance or class type (*`publisher`*) that defines and publishes the event. If *`event-name`* is a static event, *`publisher`* must specify the type name of a class that defines the event. If *`event-name`* is an instance event, *`publisher`* must be an object reference to a class instance that defines the event. The definition of the event can be abstract or implemented as long as the specified class instance actually contains a `DEFINE EVENT` statement for *`event-name`*. The object reference can be of a class or interface type, as long as the referenced type defines the instance event.

If you do not specify a *`publisher`*, you must invoke the event subscription method in an ABL class definition, where *`event-name`* identifies an event (instance or static) defined in the current class hierarchy. For example, if the unqualified *`event-name`* identifies an instance event in the class hierarchy, you must call the event subscription method from within an instance method of the class.

You can specify a class event handler as a class-based instance or static method, where *handler-method* is the unquoted method name. This method can be either locally accessible to the current class definition or implemented in a specified class instance or class type (*subscriber*). You can also specify the handler as an internal procedure, where *handler-procedure* is a character expression that specifies the procedure name. This procedure can be either locally accessible to the current procedure context or implemented in another external procedure context (*subscriber-handle*). Note that for an internal procedure, you must specify its name using a character expression instead of an unquoted name as for a method name. Also, the kind of event handler you specify is independent of the kind of event. For example, you can specify an instance method, a static method, or an internal procedure as an event handler for either an instance event or a static event.

When you subscribe an event handler to an event, its name is added to a list of handlers for the specified event. Note that you can subscribe the same event handler to two different events as long as the signatures match. However, you can subscribe any given event handler only once to the same event.

Caution: Before subscribing more than one event handler to an event, you should carefully consider how any parameters for the event are defined and how they are processed by each event handler. For more information, see Publish class events on page 170.

Similarly, unsubscribing an event handler from a class event removes the handler's subscription from the list of event handlers that are subscribed to the event. When you unsubscribe an event handler, if there is no subscription for it in a given event handler list, no action takes place and no error is raised.

Use of the NO-ERROR option supports standard ABL error handling. For more information on error handling for classes, see Raise and handle error conditions on page 197.

For a complete description of each event subscription method, see the reference entry for the Subscribe( ) event method and the Unsubscribe( ) event method in *ABL Reference*.

For example, the following code fragment from the Main sample class subscribes separate handlers to the public OutputGenerated event, which is published in one of two different, but related classes, depending on the Main constructor used to instantiate the class:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  DEFINE PRIVATE VARIABLE cOutFile     AS CHARACTER         NO-UNDO.
  DEFINE PRIVATE VARIABLE rCommonObj   AS CLASS CommonObj   NO-UNDO.
  DEFINE PRIVATE VARIABLE rCustObj     AS CLASS CustObj     NO-UNDO.
  DEFINE PRIVATE VARIABLE rCustObj2    AS CLASS CustObj     NO-UNDO.
  DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
  DEFINE PRIVATE VARIABLE rIBusObj     AS CLASS IBusObj     NO-UNDO.

  /* First constructor instantiates a Customer object */
  CONSTRUCTOR PUBLIC Main ( ):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass ( )

      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj ( )
      cOutFile      = "Customers.out".

    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.

  /* Second constructor takes a character parameter representing an input
     file of email addresses to instantiate a New England Customer object */
  CONSTRUCTOR PUBLIC Main (INPUT EmailFile AS CHARACTER):
```

```
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass( )

      /* Create an instance of the NECustObj class */
      rCustObj      = NEW NECustObj (EmailFile)
      cOutFile      = "NECustomers.out".

    /* Subscribe OutputGenerated event handler for NECustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_NECustObjHandler).
  END CONSTRUCTOR.

  /* Event handlers for each Customer class instance */
  METHOD PRIVATE VOID OutputGenerated_CustObjHandler
      (pcOutputType AS CHARACTER):
    MESSAGE pcOutputType "for all customers." VIEW-AS ALERT-BOX.
  END METHOD.

  METHOD PRIVATE VOID OutputGenerated_NECustObjHandler
      (pcOutputType AS CHARACTER):
    MESSAGE pcOutputType "for New England customers." VIEW-AS ALERT-BOX.
  END METHOD.
  ...
END CLASS.
```

Publish class events

ABL allows you to publish a class event from within the class that defines it using the `Publish( )` event method. The effect is similar to publishing a named event using the `PUBLISH` statement, but using a syntax that is more natural for object-oriented programming, which supports strong-typing of the event and its handler signature. Because the method return type is void, you must invoke it as a statement (not in an expression).

This is the ABL syntax to publish a class event:

Syntax

```
[ THIS-OBJECT : |class-type-name : ]
 event-name : Publish ( [parameter[ , parameter]...] ) [ NO-ERROR ].
```

Thus, the `Publish( )` method publishes the event specified by *`event-name`*. The class definition where you invoke `Publish( )` must contain the `DEFINE EVENT` statement for *`event-name`*, and this statement cannot have the `ABSTRACT` option. If the class (abstract or non-abstract) inherits the event as abstract, you must override the event definition without the `ABSTRACT` option in order to publish it. Thus, you cannot invoke the `Publish( )` method to publish any inherited event or any abstract event, even if you define the event as abstract in the same class definition. You only need to qualify the event name (with `THIS-OBJECT` for an instance event or the publishing *`class-type-name`* for a static event) if *`event-name`* is an ABL reserved keyword.

Each *`parameter`* in the list of parameters must be run-time compatible with the corresponding method parameter defined for the event. Even if the event handler that `Publish( )` executes is an internal procedure, this compatibility conforms to passing parameters to methods. For more information, see the Parameter passing syntax reference entry in *ABL Reference*.

When you invoke `Publish( )`, all event handlers subscribed to *`event-name`* execute once, and the results from any `OUTPUT` or `INPUT-OUTPUT` parameters are returned from the last of these event handlers to execute. However, when you have more than one event handler subscribed, the AVM does not guarantee the order in which they execute. So, you cannot be certain which event handler is returning the results from `Publish( )`. Furthermore, for any `INPUT-OUTPUT` parameter, or any `INPUT` parameter that passes a reference value (object reference or handle), each event handler that executes returns its parameter results as input to the next event handler to execute. So, you might not be certain how the results of `INPUT-OUTPUT` parameters, or `INPUT` reference parameters, are changed from one event handler to the next.

Caution: If you subscribe multiple event handlers to an event, you should ensure that these handlers are defined in such a way that their order of execution does not matter to any parameter results that you expect `Publish( )` to return.

Use of the `NO-ERROR` option on `Publish( )` supports standard ABL error handling for any event handler that raises an error, just as if you called the event handler directly. Also, if more than one event handler is subscribed to *`event-name`*, when a given event handler raises an error, the AVM stops executing event handlers beyond those that have already executed. For more information on error handling for event handlers, see Raise errors within a class event handler on page 200.

For a complete description of the `Publish( )` method, see the reference entry for the `Publish( )` event method in *ABL Reference*.

For example, the following code fragment from the `acme.myObjs.NECustObj` sample class invokes a method (`PublishOutputGenerated( )`) provided by its super class to publish the `OutputGenerated` event, which has handlers subscribed to it in the sample `Main` class:

```
USING acme.myObjs.*.

CLASS acme.myObjs.NECustObj INHERITS CustObj:
  DEFINE PRIVATE TEMP-TABLE ttEmail NO-UNDO
    FIELD RecNum AS INTEGER
    FIELD Name   AS CHARACTER FORMAT "X(20)"
    FIELD Email  AS CHARACTER FORMAT "X(20)".
  ...

/* First override version of printObj for a single copy */
  METHOD PUBLIC OVERRIDE VOID printObj ( ):
    OUTPUT TO PRINTER.
    DISPLAY dtTimestamp.
    FOR EACH ttEmail:
      DISPLAY ttEmail.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to printer").
  END METHOD.

  /* Second override version of printObj for multiple copies */
  METHOD PUBLIC OVERRIDE VOID printObj
    (INPUT piCopies AS INTEGER):
    VAR INT iCnt.
    OUTPUT TO PRINTER.
    IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
      DISPLAY dtTimestamp.
      FOR EACH ttEmail:
        DISPLAY ttEmail.
      END.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated(STRING(piCopies)
                           + " copies of report sent to printer").
  END METHOD.
```

```
  /* Override method to log customer information with email */
  METHOD PUBLIC OVERRIDE VOID logObj (INPUT pcFilename AS CHARACTER):
    OUTPUT TO VALUE (pcFilename).
    DISPLAY dtTimestamp.
    FOR EACH ttEmail:
      DISPLAY ttEmail.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to "
                           + pcFilename + " file").
  END METHOD.
END CLASS.
```

`PublishOutputGenerated( )` is a protected method that is first defined as abstract, along with the public abstract event, `OutputGenerated`, in the abstract base class, `acme.myObjs.Common.CommonObj` (see Sample classes on page 225). The derived class, `acme.myObjs.CustObj`, then implements these members by overriding their abstract definitions, and `acme.myObjs.NECustObj` inherits these implementations from `CustObj`, as shown in the previous fragment. The following fragment shows the member implementations in `CustObj`:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj IMPLEMENTS IBusObj:
  ...
  DEFINE PUBLIC OVERRIDE EVENT OutputGenerated
    SIGNATURE VOID (pcOutputType AS CHARACTER).

  METHOD PROTECTED OVERRIDE VOID PublishOutputGenerated
    (INPUT pcOutputType AS CHARACTER):
    OutputGenerated:Publish(pcOutputType).
  END METHOD.
  ...
END CLASS.
```

Thus, the protected `PublishOutputGenerated( )` method invokes `Publish( )` on the event from within the class (`CustObj`) that implements it, allowing the derived class, `acme.myObjs.NECustObj`, to publish an event that it does not define.

Verify the type and validity of an object reference

Given an object reference, before you use it, you can:

- Test it to ensure that it references an object instance using the `VALID-OBJECT` built-in function
- Validate the type of the object represented by the object reference using the `TYPE-OF` built-in function

VALID-OBJECT function

`VALID-OBJECT` is a built-in `LOGICAL` function that verifies the validity of an object reference. If the object has been deleted or equals the `Unknown value (?)`, the reference is not valid.

This is the syntax for the `VALID-OBJECT` function:

Syntax

```
VALID-OBJECT( object-reference )
```

Element description for this syntax diagram follow:

object-reference

An object reference defined as any class or interface type.

If the object reference points to an object that is currently instantiated, VALID-OBJECT returns TRUE. Otherwise, it returns FALSE.

Note: VALID-OBJECT can also verify the validity of ABL handle-based objects. For more information, see *ABL Reference*.

The acme.myObjs.Common.HelperClass sample methods make use of the VALID-OBJECT function to verify object references that might not have been initialized, depending on the order that the methods are invoked. (In fact, the sample order of execution guarantees valid object references. For more information on these samples, see Compare constructs in classes and procedures on page 225. However, in a more complex application, the validity of object references at any given point might well be far from certain.)

For example, the following HelperClass fragment, the ReportOutput( ) method verifies that the object reference variable, rCustObj, has been set to reference a class instance before passing it as an OUTPUT parameter. In this application, rCustObj must be set by the ListNames( ) method before ReportOutput( ) can be run without error:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.Common.HelperClass:
  DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg     AS CLASS MsgObj  NO-UNDO.
  ...

  CONSTRUCTOR PUBLIC HelperClass ( ):
    rMsg = NEW MsgObj ("acme.myObjs.Common.HelperClass").
  END CONSTRUCTOR.
  ...

  METHOD PUBLIC VOID ListNames (INPUT-OUTPUT prCustObj AS CLASS CustObj):
    ...
    rCustObj = prCustObj.
  END METHOD.

  METHOD PUBLIC VOID ReportOutput (OUTPUT prInterface AS CLASS IBusObj):
    /* Send the PRIVATE CustObj instance back to be printed */
    IF VALID-OBJECT(rCustObj) THEN
      prInterface = rCustObj.
    ELSE
      rMsg:Alert("Not a valid object").
  END METHOD.
  ...
END CLASS.
```

TYPE-OF function

TYPE-OF is a built-in LOGICAL function that verifies that a specified object reference points to an object that is defined as a specified class, inherits from a specified class, or implements a specified interface.

This is the syntax for the TYPE-OF function:

Syntax

```
TYPE-OF (object-reference, object-type-name)
```

Element descriptions for this syntax diagram follow:

object-reference

An object reference to any class or interface type.

object-type-name

The type name of a class or interface. This can be the fully qualified object type name or the unqualified class or interface name, depending on the presence of an appropriate USING statement in the class or procedure file. For more information on object type names, see Define and reference object type names on page 45. For more information on the USING statement, see Reference an object type name without its package on page 48.

The function verifies that the object referenced by the specified *object-reference* satisfies one of the following conditions:

1. Matches the class or interface type specified by *object-type-name*
2. Is a subclass of the class type specified by *object-type-name*
3. Implements the interface type specified by *object-type-name*

If any of these conditions are valid, this function returns TRUE. If none are valid, this function returns FALSE. If *object-reference* does not point to a valid object (see the), the TYPE-OF function returns the Unknown value (?).

Use built-in system and object reference elements

ABL provides built-in system references and object reference elements to access instantiated class-based objects at run time. A *built-in system reference* (such as the THIS-OBJECT or SUPER system reference) returns a value set by ABL that represents a given class or class-based object depending on the class context where the reference occurs in an ABL session. A *built-in object reference* element (such as the FIRST-OBJECT attribute on the SESSION system handle) holds an object reference to a class-based object that is instantiated within the ABL session. The value of a built-in object reference element is automatically set by the AVM according to the function of the element.

The different built-in system references and object-reference elements are specified by keywords in ABL. The sections that follow describe these ABL elements.

THIS-OBJECT system reference

THIS-OBJECT is a system reference available for use within an instantiated class. At run time, it returns the currently running instance of the class as an object reference. It supports several features:

- Allows a method of the class to pass a reference to the currently instantiated class hierarchy as a parameter or to return a reference to itself as a method return value.
- Allows you to call an instance method from within the class hierarchy where the method is defined when the method name is a reserved ABL keyword. To make such a method call, you use THIS-OBJECT as an object reference to invoke the method. (This is the only situation where you can and **must** use the THIS-OBJECT system reference to call a method.)
- Allows you to access a variable data member or a class property from within the class hierarchy where the variable data member or class property is defined when the name is a reserved ABL keyword.
- In general, you can use THIS-OBJECT as an object reference to call any instance method defined and available in the current hierarchy of a class as an aid to readability. For example, you might, as a convention, omit THIS-OBJECT for calling methods implemented by the current class definition, and use it as a documentary object reference for calling methods inherited by the current class definition.

Note: You cannot use the THIS-OBJECT system reference to access a static method.

The following example shows a class, acme.myObjs.NewCustomer, that inherits from the sample class acme.myObjs.CustObj:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.NewCustomer INHERITS CustObj:
  DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.

  CONSTRUCTOR PUBLIC NewCustomer( ):
    /* Create an instance of the HelperClass class */
    rHelperClass = NEW HelperClass( ).
  END CONSTRUCTOR.

  METHOD PUBLIC VOID setNewCustObj( ):
    rHelperClass:InitializeDate(THIS-OBJECT).   END METHOD.
  ...
END CLASS.
```

This is not one of the installed sample classes, but is created here to illustrate a use of the THIS-OBJECT system reference. A call to its setNewCustObj( ) method passes an object reference to the current instance of itself (THIS-OBJECT) to the InitializeDate( ) method of the sample class, acme.myObjs.Common.HelperClass, which timestamps itself, as shown here:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.Common.HelperClass:
  ...
  DEFINE PRIVATE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
  ...

  METHOD PUBLIC VOID InitializeDate (INPUT prObject AS CLASS CommonObj):
```

```
    /* Timestamp this object */
    IF VALID-OBJECT(prObject) THEN
      prObject:updateTimestamp ( ).
    ELSE
      rMsg:Alert("Not a valid object").
  END METHOD.
  ...
END CLASS.
```

Because the `THIS-OBJECT` input parameter also represents a subclass of the sample class, `acme.myObjs.Common.CommonObj`, the `InitializeDate( )` method can invoke the `updateTimestamp( )` method on the object reference parameter (`prObject`) to timestamp the object, which `setNewCustObj( )` can reference when `InitializeDate( )` returns in the `acme.myObjs.NewCustomer` class shown previously.

In the following example, a method of the `Reserved` class, `DisplayToday( )`, calls the `Now( )` method, also defined in the class:

```
CLASS Reserved:
  METHOD PUBLIC DATETIME-TZ Now( ):
    RETURN NOW.
  END METHOD.

  METHOD PUBLIC VOID DisplayToday( ):
    MESSAGE "Current date and time: " THIS-OBJECT:Now( ) VIEW-AS ALERT-BOX.
  END METHOD.
END CLASS.
```

The name of the method is an ABL reserved keyword, which is the name of the `NOW` function that also happens to be invoked by the method in this example. If `THIS-OBJECT` is not used to prefix the `Now( )` function call, ABL returns a compiler error indicating that it cannot recognize the method.

SUPER system reference

SUPER is a system reference available for use within an instantiated class hierarchy that, when executed from a subclass, invokes an instance method implemented in a super class. A common use for this system reference is to invoke a super class method that the subclass overrides. The SUPER system reference provides the only way a super class's implementation of an overridden method can be invoked, and it must be invoked from a method of a subclass. Also, the SUPER system reference can **only** be used within a subclass to invoke a super class method in the class hierarchy of that subclass. There is no way to directly invoke a super class version of an overridden method from outside of the class hierarchy. For more information on the syntax and use of this system reference, see Construct an object on page 116.

Note: Unlike `THIS-OBJECT`, the SUPER system reference is not an object reference, and it can only be used to invoke a super class method from a subclass in the same class hierarchy. You cannot use SUPER, by itself for example, to pass an object reference to the current instance's super class as a parameter.

The SUPER system reference is not the same as the SUPER statement. For more information on the SUPER statement, see Construct an object on page 116.

You cannot use the SUPER system reference to access a static method.

ABL session object reference attributes

The FIRST-OBJECT and LAST-OBJECT attributes on the SESSION system handle provide access to the list of currently instantiated objects, commonly referred to as the SESSION object chain. FIRST-OBJECT and LAST-OBJECT both hold references to Progress.Lang.Object objects. Once you get the first object reference in the list using FIRST-OBJECT or the last object reference in the list using LAST-OBJECT, you can use the NEXT-SIBLING and PREV-SIBLING properties on Progress.Lang.Object to walk the list of currently instantiated objects.

Because these are object references to Progress.Lang.Object, if you need to use the object as its instantiated type (the type referenced by the NEW function), you need to CAST the object to the required type. For more information on casting, see Object reference assignment and casting on page 179.

For example, you can display the list of class type names for all classes currently instantiated in the ABL session with the following code:

```
var class Progress.Lang.Object myObj.

myObj = SESSION:FIRST-OBJECT.REPEAT:
  IF myObj EQ ? THEN LEAVE.

  MESSAGE myObj:GetClass( ):TypeName VIEW-AS ALERT-BOX.
  myObj = myObj:NEXT-SIBLING.
END.
```

Note that object references on the SESSION object chain (such as the FIRST-OBJECT attribute and the NEXT-SIBLING object property) are not included in the reference count for garbage collection. When the AVM determines that an object's reference count is zero, it is garbage collected, even though the SESSION object chain still references it. Once it is garbage collected, the object is also removed from the SESSION object chain.

Assign object references

You can freely assign one object reference data element to another if they are defined with compatible data types. The object reference elements in an assignment have compatible data types if one of the following is true:

- Both reference elements are defined as the same class or interface type (acme.myObjs.CustObj = acme.myObjs.CustObj or acme.myObjs.Interfaces.IBusObj = acme.myObjs.Interfaces.IBusObj).
- The left-hand reference element is defined as a super class of the right-hand reference element (acme.myObjs.Common.CommonObj = acme.myObjs.CustObj).
- The left-hand reference element is defined as an interface implemented by the class referenced by the right-hand element (acme.myObjs.Interfaces.IBusObj = acme.myObjs.CustObj).

This assignment always sets the left-hand object reference data element to a copy of the right-hand object reference that points to the same object, not a reference to a new copy of the object.

When you assign a subclass object reference to a super class object reference element, the new super class object reference still points to the original subclass instance, but it provides access to the object according to the definition of the super class type. Thus, if you use the super class object reference, you can only access the methods defined in that super class.

You can also assign a class object reference to an interface object reference element, where the class implements the specified interface type at any level in its class hierarchy. If you use the new interface object reference, you can only access the methods defined in the interface.

So an object reference can be copied to an object reference data element of the same type, a super type, or an interface that the referenced object implements. The converse of this rule is **not** true. That is, an object reference to a super class cannot be copied to a subclass object reference element unless you use the CAST function or the DYNAMIC-CAST function, and the super class reference actually points to an object originally instantiated as the specified subclass. Thus, the compiler rejects such an assignment and only allows it if an appropriate compile-time validated CAST function, or run-time validated DYNAMIC-CAST, function is provided on the right-hand side.

The following variation on the sample Main class demonstrates compatible assignment between different types of object reference variables using the sample classes. These assignments are indicated in this Main class code by the following numbered lines:

1. Assignment between identical class types (acme.myObjs.CustObj to acme.myObjs.CustObj)
2. Assignment to a super class type from one of its subclass types (acme.myObjs.CustObj to acme.myObjs.Common.CommonObj)
3. Assignment to an interface type from a class type that implements the interface (acme.myObjs.CustObj to acme.myObjs.Interfaces.IBusObj)

This is the Main class fragment with the numbered lines:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  DEFINE PRIVATE VARIABLE cOutFile     AS CHARACTER         NO-UNDO.
  DEFINE PRIVATE VARIABLE rCommonObj   AS CLASS CommonObj   NO-UNDO.
  DEFINE PRIVATE VARIABLE rCustObj     AS CLASS CustObj     NO-UNDO.
  DEFINE PRIVATE VARIABLE rCustObj2    AS CLASS CustObj     NO-UNDO.
  DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
  DEFINE PRIVATE VARIABLE rIBusObj     AS CLASS IBusObj     NO-UNDO.

  /* First constructor instantiates a Customer object */
  CONSTRUCTOR PUBLIC Main ( ):
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass ( )

      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj ( )
      cOutFile      = "Customers.out".
      ...
  END CONSTRUCTOR.
  ...

  /* ObjectInfo processes information about the Customer object    */
  METHOD PUBLIC VOID ObjectInfo ( ):
    /* Demonstrates passing object references as parameters      */

    rCommonObj = rCustObj. /* #2 */

    IF rCustObj:GetClass( ):TypeName = "acme.myObjs.NECustObj" THEN
      MESSAGE "Initializing reports for New England customers at"
        STRING(rCommonObj:updateTimestamp( ))
        VIEW-AS ALERT-BOX.
    ELSE
      MESSAGE "Initializing reports for all customers at"
        STRING(rCommonObj:updateTimestamp( ))
        VIEW-AS ALERT-BOX.
```

```
    ...

    /* INPUT-OUTPUT: Must be an exact match, a class to a method defined to
       take that same class type */
    rHelperClass:ListNames (INPUT-OUTPUT rCustObj).
    rCustObj:CheckCredit ( ).
    rCustObj2 = rCustObj.  /* #1 */

    /* OUTPUT: An interface is used to receive a class that implements that
       interface */
    rHelperClass:ReportOutput (OUTPUT rIBusObj).
    IF piInfoCount <> ? AND piInfoCount > 1 THEN
      rIBusObj:printObj(piInfoCount).
    ELSE
      rIBusObj:printObj( ).
    rIBusObj:logObj (cOutFile).
    rIBusObj = rCustObj.   /* #3 */
  END METHOD.
END CLASS.
```

Note that all of these object references, in fact, point to the same instance of an object. When the default destructor executes, garbage collection deletes the `CustObj` instance that all four object references point to, and all four variables then hold invalid object references.

Object reference assignment and casting

As described in the previous section, you can assign a source object reference to a target object reference data element if the target data element is defined as the same class, a super class, or an interface of the source object. When the target data element is defined for a super class or interface, an implicit cast is done by the AVM at run time. The word *cast*, as used in object-oriented terminology, can be compared to the cast of a play, in which each actor is assigned a role. When you program with classes, each reference to an object must identify the role that the object is required to play, as identified by its data type. The compiler verifies that no object reference is allowed to act contrary to its assigned role and access a data member, property, method, or event not defined by its data type.

Using a super class object reference, rather than an object reference to a specific subclass, allows you to take advantage of any polymorphic behavior implemented by available subclasses. Thus, you can assign a subclass instance to an object reference for a given super class. When you invoke any method on this super class object reference, the subclass implementation of that method executes, depending on the particular subclass instance that you have assigned. By implicitly casting the assignment of the subclass to the super class object reference, ABL enables transparent access to this polymorphic behavior. (For more information on polymorphism, see Use polymorphism with classes on page 125.)

Similarly, using an interface object reference, rather than an object reference to a specific class that implements that interface, allows you to take advantage of any polymorphic behavior provided by all classes that implement the same interface. Thus, you can assign a class object reference to a data element defined as a given interface type. When you invoke any method on the assigned interface object reference, the implementation of that method for the actual class instance executes, depending on the particular class instance that you have assigned. By implicitly casting the assignment of the class to the interface object reference, ABL enables transparent access to this polymorphic behavior.

Again, ABL permits all of these assignments from a subclass reference to a super class-type data element, or from an implementing-class reference to an interface-type data element, because the very nature of subclass inheritance and class interface implementation guarantees that `PUBLIC` and `PROTECTED` members provided by the respective super class and interface object references must work.

For example, refer to the following class hierarchy:

```
Progress.Lang.Object           <--- Top of hierarchy
  acme.myObjs.Common.CommonObj
    acme.myObjs.CustObj
      acme.myObjs.NECustObj    <--- Bottom of hierarchy
```

You can directly assign an `acme.myObjs.NECustObj` object reference to either an `acme.myObjs.CustObj`, `acme.myObjs.Common.CommonObj` or `Progress.Lang.Object` data element. You can also directly assign an `acme.myObjs.CustObj` object reference to an `acme.myObjs.Common.CommonObj` or `Progress.Lang.Object` data element.

Sometimes it is necessary to cast downward—that is, to take an object reference for a super class or interface in a class hierarchy and assign it to a data element defined as a subclass or interface-implementing class further down the class hierarchy. As noted, the compiler will not do this implicitly when you perform an assignment or pass a parameter, because it cannot verify that references to the object will all be valid.

To explicitly cast an object reference downward, use either the `CAST` function or the `DYNAMIC-CAST` function. You can use either of these two functions to implement a cast. However, a cast using the `CAST` function is initially validated at compile time, while the same cast using the `DYNAMIC-CAST` function is validated only at run time.

For the cast to work:

- An initial super class object reference must actually point to an instance of the subclass to which you are casting
- An initial interface object reference must actually point to the class instance to which you are casting, and that class instance must implement the interface type of the initial object reference

Thus, you can only cast an object reference downward when the object was originally instantiated as the target class or one of its subclasses. This means that you must know that a super class object reference, at run time, is really an object reference to a subclass instance or that an interface object reference is really an object reference to an interface-implementing class instance.

Note: While you must use the `CAST` or `DYNAMIC-CAST` function to cast downward, you can also use these functions to cast upward. However, this is typically not necessary because, as already noted, ABL automatically casts object reference assignments upward.

Casting tells ABL to trust your judgment and allow the downward cast within the specified class hierarchy. However, the AVM checks at run time to ensure that a downward cast is valid. For example, if a super class object reference is not pointing to an instance of the specified subclass, the AVM returns a run-time error. This means that when you use the `CAST` (or `DYNAMIC-CAST`) function, you are bypassing some (or all for `DYNAMIC-CAST`) of the compiler's checks for validity of object references. This increases the flexibility of how you can assemble different classes and let them interoperate, but with increased responsibility on your part to make sure that all types are valid at run time.

In addition to validation, the `CAST` and `DYNAMIC-CAST` functions each have some different usage requirements.

Use the CAST function

You can thus use the `CAST` function wherever an object reference is permitted:

- To assign an object reference to an object reference data element, especially when assigning a super class or interface reference to a related subclass or interface-implementing class element, respectively.

- To pass a super class object reference as an argument to a routine `INPUT` parameter defined as a subclass type, or to pass an interface object reference as an argument to a routine `INPUT` parameter defined as a class that implements the passed interface type.
- To access a method, property, or data member on a subclass of a specified class-type object reference, or on a class that implements a specified interface-type object reference. You can do this by implicitly using the return value of the `CAST` function as the new object reference on which to access the method, property, or data member.

Syntax descriptions for each of these uses of the `CAST` function follow.

This is the syntax for the `CAST` function when used to assign an object reference to another object reference data element:

Syntax

```
target-object-reference = CAST ( object-reference , target-type-name )
```

This is the syntax for the `CAST` function when used to pass an object reference to an `INPUT` parameter:

```
routine-name ( [ INPUT ] CAST ( object-reference , target-type-name ) )
```

This is the syntax for the `CAST` function when used to cast an object reference to access a method, property, or data member:

```
CAST ( object-reference , target-type-name ) : member-name[ ( ... ) ]
```

Element descriptions for these syntax diagrams follow:

target-object-reference

A data element defined as the object reference type specified by *target-type-name*, typically defined to reference one of the following types:

- A subclass of the class whose type defines the *object-reference*
- A class that implements the interface whose type defines the *object-reference*

object-reference

An object reference source, whose type is typically defined as a super class of *target-type-name*, or an interface implemented by a class type specified by *target-type-name*. This can be a variable or it can be a temp-table field defined as `Progress.Lang.Object`. At run time, the AVM verifies that *object-reference* in fact points to an instance of the specified *target-type-name*.

target-type-name

The type name of the target class or interface type for the cast. Any class type name must specify one of the following types:

- The same class whose type defines *object-reference*
- A super class of the class whose type defines *object-reference*

- A subclass of the class whose type defines *object-reference*
- A class that implements the interface whose type defines *object-reference*

Any interface type name must specify one of the following types:

- The same interface whose type defines *object-reference*
- An interface implemented by the class type that defines *object-reference*

The specified type name can be a fully qualified object type name or an unqualified class or interface name, depending on the presence of an appropriate USING statement in the class or procedure file. For more information on object type names, see Define and reference object type names on page 45. For more information on the USING statement, see Reference an object type name without its package on page 48.

routine-name

The name of any procedure, user-defined function, or method that takes an object reference argument as INPUT. The corresponding INPUT parameter defined by *routine-name* must have a data type that is compatible with *target-type-name*.

member-name **[** (**...**) **]**

This can be one of the following:

- The name of a PUBLIC method (and its parameter list) defined by the class or interface type specified by *target-type-name*. The compiler verifies that *member-name* specifies a valid method defined by *target-type-name* and that the parameters ((**...**)) are valid.
- The name of a PUBLIC data member defined by the class type specified by *target-type-name*. The compiler verifies that *member-name* specifies a valid data member defined by *target-type-name*.
- The name of an accessible PUBLIC property defined by the class type specified by *target-type-name*. The compiler verifies that *member-name* specifies a valid property defined by *target-type-name*, and that the specified property is accessible (readable or writable) in the coded context.

Casting an object reference assignment

For an example of assigning an object reference using the CAST function, see Use the CAST function on page 180.

Casting an object reference parameter

The following example classes are used by the following examples to demonstrate the use of the CAST function to pass INPUT object reference parameters. The first example class is a super class that defines nothing:

```
CLASS SuperClass:
END CLASS.
```

The next example class is a subclass that defines its own method, `subClassMethod( ):`

```
CLASS SubClass INHERITS SuperClass:
  METHOD PUBLIC VOID subclassMethod ( ):
  END METHOD.
END CLASS.
```

ABL does not, by default, allow an object reference of `SuperClass` to be used as an instance of `SubClass`. For example, if you try to execute `subClassMethod( )` using the object reference to `SuperClass`, the method cannot be found. However, if you know that an object reference to `SuperClass` in reality points to an instance of `SubClass`, you can cast the reference as `SubClass` to use it.

The following example demonstrates the use of the CAST function to pass an `INPUT` object reference parameter. The `Main` container class defines an object reference (`mySuper`) for the super class and a method (`subParmMethod( )`) that expects an object parameter of type `SubClass`, as shown:

```
CLASS Main:
  DEFINE PRIVATE VARIABLE mySuper AS CLASS SuperClass NO-UNDO.

  CONSTRUCTOR PUBLIC Main ( ):
    mySuper = NEW SubClass ( ).
    subParmMethod (CAST(mySuper, SubClass)).
  END CONSTRUCTOR.

  METHOD PRIVATE VOID subParmMethod
    (INPUT myLocalSubClass AS CLASS SubClass):
    myLocalSubClass:subClassMethod( ).
  END METHOD.
END CLASS.
```

In the previous example, the constructor instantiates a `SubClass` object, but assigns it to the `SuperClass` object reference. Using the CAST function, the constructor then invokes `subParmMethod( )`, passing the object reference as the `SubClass` parameter the method expects and that it actually is. The `subParmMethod( )` method then invokes the `subClassMethod( )` method on the passed object reference.

Casting an object reference to invoke a method

The following example shows an alternative `Main` class used to access the sample classes `SuperClass` and `SubClass` defined in the previous example, this time to demonstrate the use of the CAST function to cast an object reference for a method call:

```
CLASS Main:
  DEFINE PRIVATE VARIABLE mySuper AS CLASS SuperClass NO-UNDO.

  CONSTRUCTOR PUBLIC Main ( ):
    mySuper = NEW SubClass ( ).
    CAST(mySuper, SubClass):subClassMethod ( ).
  END CONSTRUCTOR.
END CLASS.
```

In the previous example, the constructor sets the `SuperClass` object reference to a new `SubClass` instance, then invokes `subClassMethod( )` on the `SubClass` instance by casting the `SuperClass` object reference to `SubClass`.

Use the DYNAMIC-CAST function

You can thus use the DYNAMIC-CAST function in some of the same ways as a CAST function:

- To assign an object reference to an object reference data element, especially when assigning a super class or interface reference to a related subclass or interface-implementing class element, respectively
- To pass a super class object reference as an argument to a routine INPUT parameter defined as a subclass type, or to pass an interface object reference as an argument to a routine INPUT parameter defined as a class that implements the passed interface type

However unlike the CAST function, you cannot use the DYNAMIC-CAST function to directly access a method, property, or data member on the function return value without first assigning that value to an appropriate object-reference data element for accessing the class member.

Syntax descriptions for each of the permitted uses of the DYNAMIC-CAST function follow.

This is the syntax for the DYNAMIC-CAST function when used to assign an object reference to another object reference data element:

Syntax

```
target-object-reference = DYNAMIC-CAST
  ( object-reference , target-type-expression )
```

This is the syntax for the DYNAMIC-CAST function when used to pass an object reference to an INPUT parameter:

```
routine-name ( [ INPUT ] DYNAMIC-CAST
  ( object-reference , target-type-expression ) )
```

Element descriptions for these syntax diagrams follow:

target-object-reference

A data element defined as the object reference type specified by *target-type-expression*, typically defined to reference one of the following types:

- A subclass of the class whose type defines the *object-reference*
- A class that implements the interface whose type defines the *object-reference*

object-reference

An object reference source, whose type is typically defined as a super class of the class type specified by *target-type-expression*, or an interface implemented by a class type specified by *target-type-expression*. This can be a variable or it can be a temp-table field defined as Progress.Lang.Object. At run time, the AVM verifies that *object-reference* in fact points to an instance of the object type specified by *target-type-expression*.

target-type-expression

A character expression that evaluates to the type name of the target class or interface type for the cast. Any class type name must specify one of the following types:

- The same class whose type defines *object-reference*

- A super class of the class whose type defines *object-reference*
- A subclass of the class whose type defines *object-reference*
- A class that implements the interface whose type defines *object-reference*

Any interface type name must specify one of the following types:

- The same interface whose type defines *object-reference*
- An interface implemented by the class type that defines *object-reference*

The specified type name must be a fully qualified object type name. Because *target-type-expression* is evaluated at run time, any USING statements have no affect on its evaluation. For more information on object type names, see Define and reference object type names on page 45.

routine-name

The name of any procedure, user-defined function, or method that takes an object reference argument as INPUT. The corresponding INPUT parameter defined by *routine-name* must have a data type that is compatible with object type specified by *target-type-expression*.

Compare objects

You can compare two object references for equality using the equals operator (=), which checks if two object references are actually referencing the same object. The object references can be equal even if the object references have been defined for different classes in the class hierarchy. If the two object references are defined for different classes, they are also equal if they are both set to the Unknown value (?).

Note: You can also perform the same object reference comparison using the Equals( ) method on Progress.Lang.Object. For more information, see Use the root class: Progress.Lang.Object on page 91.

The following example demonstrates the use of the equal operator using the two object references stored in rSuper and rSubClass:

```
VAR SubClass rSubClass = NEW SubClass( ).
VAR SuperClass rSuper.

rSuper = rSubClass.

IF rSuper = rSubClass THEN
  MESSAGE "they are the same".
```

The MESSAGE statement will indeed be executed, demonstrating that rSuper and rSubClass are equal even though they reference different class types in the class hierarchy.

Use static members of a class

As previously described, you can define certain members of a class as static (see Use the CLASS construct on page 54). Defining a member as static scopes it to the defining class type and makes it available for the duration of the ABL session, rather than scoping it to a class instance for the duration of the object, as with instance members. You do not need to instantiate a class in order to initialize and access its static members; for all static members defined with appropriate access modes, you only need to have access to the class type that defines them in order to access them. Thus, static members of a class automatically become initialized after your first reference to that class type, and you can access these static members directly as components of that class, independent of any instances of the class you might also have created or deleted.

The following sections describe additional basic features of static members and a common use case:

- Static member scoping on page 186
- Access static members on page 186
- Define static members on page 190
- Initialize and delete static members on page 192
- Common use case for static members on page 192
- Static type-name syntax and naming conflicts on page 194

Static member scoping

From the moment they are initialized, static members are available for the defining class type and for the duration of an ABL session. As such, static members are globally available resources for an ABL session, and values of static data persist throughout the session. For example, whether you access it from inside the class hierarchy of a class instance or (if `PUBLIC`) from outside of the class hierarchy where it is defined, the value of a given static property or data member persists from one access to the next for the duration of the session until it is changed by one of these accesses.

Note: If you BIND a client to an ABL session and use QUIT to release the session, the application server agent begins a fresh ABL session with each new client connection. Thus, each client connection to the application server re-initializes the static members of each class that is referenced.

Access static members

In general, you can access static members from other static member definitions, from instance member definitions, and from external procedures. As with instance members, the requirements for accessing static members differ when accessing them from outside or inside the class where they are defined. The available options for accessing static members also depend on the kinds of static members you are accessing. Finally, the requirements for accessing overridden static methods are different than for accessing overridden instance methods.

The following sections describe these access options and requirements:

- Options for referencing static members on page 187
- Use static type-name syntax on page 187

- Access static members from outside the defining class on page 188
- Access static members from inside the defining class on page 189
- Call static methods from inside the defining class on page 189
- Call overridden and super-class static methods on page 189
- Call overloaded static methods on page 190
- Access static and instance events and event handlers on page 190

Options for referencing static members

In most cases, you can reference a static member by name, with or without a type-name prefix, depending on the access modes that it supports. For each kind of static member that supports the `PUBLIC` access mode, you can, with one exception, reference the member by prefixing the member name with the object type name of the class that defines it. For all other static members, you can only reference them by name, according to their kind. You can use this type-name prefix to reference the member name as specified for static type-name syntax. For more information, see Use static type-name syntax on page 187.

Therefore, you **can** use static type-name syntax to reference the following kinds of static members, all of which support the `PUBLIC` access mode:

- Variable data members
- Properties
- Named methods (not constructors or destructors)
- Class events

If it has been named with a reserved keyword, accessing a static variable data member, property, method, or class event within the class in which it is defined requires static type-name syntax as a qualifier.

You **cannot** use static type-name syntax to reference the following kinds of static members, all of which **do not** support the `PUBLIC` access mode:

- Buffers
- Data-sources
- ProDataSets
- Queries
- Temp-tables

For these kinds of static members, you can only reference them by name, exactly like instance members, as defined for the type of member. For example, to reference the default buffer of a statically defined temp-table, you precede the temp-table name with the `BUFFER` keyword.

Use static type-name syntax

You can call a static method or access a static variable data member, property, or class event using static type-name syntax, as follows:

Syntax

```
class-type-name : {method-name ( [parameters] ) |data-element |event}
```

Element descriptions for this syntax diagram follow:

class-type-name

The type name of the class where the static member you are referencing is defined. This can be the fully qualified class type or the class name with the presence of an appropriate USING statement. For more information on class type names, see Define and reference object type names on page 45.

method-name ([*parameters*])

The name of a static method (*method-name*) that is accessible and defined by *class-type-name*, followed by any required parameter list (*parameters*). In all other respects, you call a static method exactly like an instance method. For more information on calling methods and specifying parameter lists, see the class-based method call entry in *ABL Reference*.

data-element

The name of a static variable data member or static property that is accessible and defined by *class-type-name*. In all other respects, you access a static variable data member or property exactly like an instance data member or property. For more information, see the and the class-based data member access and property access entries in *ABL Reference*.

event

The name of a static class event that is accessible and defined by *class-type-name*. In all other respects, you access a static class event exactly like an instance event, in order to publish it or subscribe event handlers to it. For more information, see the and the entries for the Publish( ), Subscribe( ), and Unsubscribe( ) event methods in *ABL Reference*.

Use of this syntax for referencing these static members is required or optional, depending on the context where you access them and on other features of static members.

Access static members from outside the defining class

From outside its defining class, you can access any supported static member that is defined as PUBLIC, and you must access that member using static type-name syntax.

For example, if the updateTimestamp( ) method from the acme.myObjs.Common.CommonObj sample class was defined as static, you could call it as in the following procedure fragment:

```
USING acme.myObjs.Common.*.

CommonObj:updateTimestamp( ).
```

Access static members from inside the defining class

In most cases, you can access a static member from inside the class hierarchy where it is defined in exactly the same way as an instance member, by referencing the member directly by name. However, Progress Software Corporation recommends that you always use static type-name syntax to reference static members that support its use, even from within the class where they are defined. In part, this results in cleaner code that clearly distinguishes references to static members from references to instance members. It also helps to avoid some potential problems, including conflicts with references to other ABL elements, such as table fields, for example, that might share the same name as a static variable. For more information, see Static type-name syntax and naming conflicts on page 194.

The following section describes special circumstances for calling static methods from inside their defining class hierarchy. Otherwise, calling static methods or accessing static data members or properties from inside their defining class hierarchy is similar to calling and accessing instance methods, data members, and properties. For more information on calling methods within the class hierarchy where they are defined, see Call methods from inside a class hierarchy where they are defined on page 142. For more information on accessing data members or properties from within the class hierarchy where they are defined, see Access data members and properties on page 153.

Call static methods from inside the defining class

ABL imposes two specific requirements for calling static methods within a class hierarchy:

- You cannot use the SUPER system reference to call a static method in a super class. This system reference only works with an instance hierarchy. Instead, you can use static type-name syntax to call any static method in a super class. For more information on calling static methods in a super class, see Call overridden and super-class static methods on page 189.
- If the method name is a reserved keyword, you cannot use the `THIS-OBJECT` system reference to call the method, as with instance methods. Instead, you must use static type-name syntax to call a method named with a reserved keyword from within its defining class.

For example, the ACCUMULATE keyword is reserved in ABL. If you have an accessible static `Accumulate( )` method defined within the current class hierarchy, you **cannot** call it directly by name, as follows:

```
Accumulate( ). /* Compiler error */
```

However, if the method is defined in the `acme.myObjs.Common.CommonObj` class, you can call it within the defining class hierarchy as follows:

```
acme.myObjs.Common.CommonObj:Accumulate( ). /* Valid call */
```

Call overridden and super-class static methods

When an instance method is overridden in a class hierarchy, calling the method will, by default, always polymorphically invoke the overridden method implementation in the most derived subclass, unless you invoke the method using the SUPER system reference. Using SUPER, you can call any overridden method implementation in the immediate super class. All other implementations of the overridden instance method within the same class hierarchy are effectively hidden from access (see Override methods within a class hierarchy on page 107.

However, when a static method is overridden in a class hierarchy, this does not define a polymorphism, and you can call any implementation of the overridden method in the hierarchy with reference to its implementing class type. Unlike for instance methods, each override of a static method represents a re-definition that is available for execution at any point in your code using static type-name syntax.

If you call an overridden static method (whether from inside or outside the class hierarchy where it is defined), ABL executes the static method implementation at the closest point in the hierarchy of the specified class type. Thus, if the method is implemented in the specified class definition, that method is called; however, if the method is defined in a super class of the specified class definition, the method in the closest super class to the specified class definition is called.

Note that an unqualified reference to a static method can never execute an override in a subclass that derives from the calling class, because there is no class instance that defines and contains the further derived subclass as part of its class hierarchy. The only way to access a further derived override of a static method that is defined in the current class hierarchy is to call the override directly with reference to the overriding class type. In addition, the static method override that you call must be public, like any static method that you call on a class type outside of the current class hierarchy.

Call overloaded static methods

Aside from the requirement to use static type-name syntax in certain cases, the requirements for calling overloaded static methods are exactly the same as for calling overloaded instance methods, except that, with only a single static constructor in a class, overloading is not supported for the static constructor of a class. For more information on calling overloaded methods, see Call an overloaded method or constructor on page 150.

Access static and instance events and event handlers

There is no restriction on how static and instance events and event handlers are used with one another. You can subscribe a static method as a handler for an instance event or subscribe an instance method as a handler for a static event. However, as with accessing other instance members from a static method defined in the same class hierarchy, you cannot publish an instance event or subscribe any handler for an instance event from within a static method, constructor, or property accessor that is defined in the same class or class hierarchy where the instance event is referenced. You can only make reference to a public instance event from within a static method using an accessible object reference to a class instance that defines the event. For more information on publishing and subscribing handlers for class events, see Publish and subscribe to class events on page 167.

Define static members

You can define many kinds of class members as static, including, a single optional constructor, with an empty parameter list. In addition, you can define any number of the following static class members that support the `PRIVATE`, `PACKAGE-PRIVATE` , `PROTECTED`, `PACKAGE-PROTECTED` and `PUBLIC` access modes:

- Variable data members
- Methods
- Properties
- Events

You can define any number of the following static data members that support only the `PRIVATE` and `PROTECTED` access modes:

- Buffers

- ProDataSets
- Data-sources
- Queries
- Temp-tables

You can, and typically do, define both static and instance members in the same class. In other words, ABL does not support the concept of a static class where all of its members must be static. Syntactically, you can define a static member of a class exactly the same as an instance member, except for the addition of the `STATIC` option in the member definition. For more information on the syntax for defining static members, see Use the CLASS construct on page 54.

Basic requirements for defining static members include the following:

- As part of its definition, a static member can access only other static members and other data that is defined locally as part of the static member definition itself. A static member cannot access an instance member from within the same class hierarchy as the static member, itself. As a result, you cannot use the `SUPER` or `THIS-OBJECT` system references from within a static method, property accessor, or constructor. However, you can access public instance members on an object reference to any class instance that is not equal to `THIS-OBJECT`.

Note: When accessed from a class, ABL views the default buffer of a database table as an instance member of the class. Note also, that, unlike a static member, an instance member can include access, in its definition, to both other instance members and to static members.

- A complex static data member, such as a ProDataSet, can only include another static data member in its definition (in this case, a buffer) that has the same or a less restrictive access mode as the access mode defined for the complex data member itself.

For example, in the sample `acme.myObjs.Common.CommonObj` class, you might change the definitions of the `timestamp` instance variable and `updateTimestamp( )` instance method to a static property and method, respectively, as follows:

```
USING acme.myObjs.Common.*.

CLASS acme.myObjs.Common.CommonObj ABSTRACT:
  DEFINE PROTECTED STATIC PROPERTY dtTimestamp AS DATETIME NO-UNDO
    GET .
    PRIVATE SET .

  METHOD PUBLIC STATIC VOID updateTimestamp ( ):
    CommonObj:dtTimestamp = NOW.
    RETURN CommonObj:dtTimestamp.
  END METHOD.
  ...
END CLASS.
```

Of course, you would likely revise some of the related sample classes accordingly. For more information on the sample classes see Compare constructs in classes and procedures on page 225.

For more examples of static definitions for all supported class members, see the examples provided with the `CLASS` statement reference entry in *ABL Reference*.

Define a static constructor

If a class has static members, ABL creates a default static constructor for the class, if you do not define one, that initializes any static data members and properties of the class with their initial values. If you do define a static constructor, it can take no parameters, and like any static method, it cannot access instance members of the class. As a result, it also cannot execute the SUPER or THIS-OBJECT statements, which invoke only instance constructors. A typical reason to define a static constructor is to complete the initialization of static data members, such as ProDataSets, that require executable statements to complete their initialization. For more information on static constructor execution, see Initialize and delete static members on page 192.

Define static method overrides and overloadings

When overriding a super-class method with a static method, the overridden method in the super class must also be defined as static. You cannot override an instance method with a static method, and you cannot do the reverse.

When overloading methods, note that instance and static methods overload each other. In other words, the scope (presence or absence of the STATIC keyword) is not counted in distinguishing one overloaded method from another. For this reason, you cannot define an instance method and a static method in the same class hierarchy that both have the same signature. Such a combination causes ABL to raise a compile-time error.

Initialize and delete static members

ABL calls the static constructor of a class exactly once in a session to initialize its static data members and properties some time after you first reference the class type in the session. ABL calls this static constructor before any other action on a class type that has static members. So, if you instantiate the class as the first reference to its type, its static constructor executes before any of its instance constructors. If the static constructor raises ERROR, ABL fails to load its class type. This makes the class type unavailable for access during the remainder of the session. For more information on error handling for static constructors, see Raise errors within a static constructor on page 205.

If the class definition has super classes with their own static members, ABL initializes all of these static members by calling the static constructor for each class in the hierarchy from top to bottom. If the class is being instantiated with this reference, it then initiates the call to all the instance constructors of the class hierarchy. For more information on initialization during class instantiation, see Class hierarchies and inheritance on page 102.

Note: ABL provides no guarantee exactly when a static constructor will run, only that it will run before any instance constructor of the class type runs.

As noted previously, static members of a class, once initialized, are scoped to the class type and remain available for the duration of the ABL session. Thus, you cannot delete them during the session. The only way to remove static members from an application is by removing them from the source code and running the recompiled application in a new ABL session.

Common use case for static members

As a short example, a common use case for static members is to implement the singleton programming pattern. A *singleton* is a class for which you can instantiate only one instance that is used everywhere. Thus, a singleton implements a class that can provide instance members to an application globally in a manner analogous to how static members provide resources globally to an application.

You can create a singleton by defining a class that instantiates itself and assigns its object reference to a static read-only property when that same property is first referenced. After that point, every reference to this static property always returns an object reference to the same class instance, allowing you to access its instance members.

Thus, the following class definition shows how you can create a singleton class, `SingletonProp`:

```
CLASS SingletonProp:
  DEFINE PUBLIC STATIC PROPERTY Instance AS CLASS SingletonProp NO-UNDO
    GET.
    PRIVATE SET.

  CONSTRUCTOR PRIVATE SingletonProp( ):
    /* Make the constructor private so it cannot be called from outside of
       the class  */
  END CONSTRUCTOR.

  CONSTRUCTOR STATIC SingletonProp( ):
    SingletonProp:Instance = NEW SingletonProp( ).
    /* Create the one single instance when the static constructor is
       called */
  END CONSTRUCTOR.

  /* Continue with additional code for the class */

  DEFINE PUBLIC PROPERTY MaxTemperature AS DECIMAL INITIAL 0.0 NO-UNDO
    GET.
    SET.
END CLASS.
```

This example class defines a `PRIVATE` instance constructor, and it defines a static constructor that instantiates the class using the `NEW` statement, which invokes the `PRIVATE` constructor and assigns the object reference to the static property, `Instance`. Note that this `PUBLIC` property has a `PRIVATE SET` accessor to ensure that only a static method or constructor of the class or another instance of the class can set the property. This example also defines a `PUBLIC` instance property, `MaxTemperature`, that can be accessed on the instance.

The following example procedure indirectly instantiates `SingletonProp` by referencing its static `Instance` property in order to assign the resulting object reference to its own object reference variable, `rSingle`:

```
VAR CLASS SingletonProp rSingle.

rSingle = SingletonProp:Instance.

/* Displays 0 the first time this is run only */
MESSAGE "Max temp is" rSingle:MaxTemperature VIEW-AS ALERT-BOX.

rSingle:MaxTemperature = 98.6.

/* Displays 98.6 */
MESSAGE "Max temp is" rSingle:MaxTemperature VIEW-AS ALERT-BOX.
```

It then reads and sets the value of the instance property, `MaxTemperature`. Note that if you re-run this procedure in the same application session, both `MESSAGE` statements display `"Max temp is 98.6"` because `rSingle:MaxTemperature` is already initialized from the first execution and retains its previous value.

Static type-name syntax and naming conflicts

The following situations involving static type-name syntax can result in name conflicts that cause unexpected behavior or compiler errors:

- **Naming a variable the same as a class** — ABL allows you to have both a class and a variable defined with the same name. This is true even for an object-reference variable that is defined as the given class. However, because variable references take precedence over static type-name references, this can result in hiding static references with unexpected effects. For example, if you have a class, `clsName`, and define a variable, `clsName`, any attempt to reference public members of the `clsName` class raises a compile-time error, because ABL identifies all references to `clsName` as a variable. Therefore, Progress Software Corporation recommends that you do not name any locally scoped data elements (be they variables, temp-tables, buffers, etc.) the same as a class name.
- **Type-names used with qualified table field names** — In addition to hiding static type name references, variable references hide table field references, and temp-table buffer references hide database buffer references. Because the syntax for many of these references are the same, having identical names for these different references can cause naming conflicts. For example, the following references will result in compile-time errors if you are connected to the `Sports2000` database and you try to reference the static `MaxValue` property on a class type with same syntactic naming as a reference to the:
  - `Customer.Name` field as a static `Customer.Name:MaxValue` reference
  - `Sports2000.Customer` buffer as a static `Sports2000.Customer:MaxValue` reference
  - `Sports2000.Customer.Name` field as a static `Sports2000.Customer.Name:MaxValue` reference

  Note that a class type name with three or more dots (`.`), such as `Sports2000.Customer.Name.Security`, can never be ambiguous with a database reference, because no database reference can contain more than two dots. Therefore, Progress Software Corporation recommends that you do not use package and class names that might result in type-name references that are syntactically identical to the possible qualified temp-table or database buffer and field references. You can also avoid such ambiguities by ensuring that type names contain a minimum of three dots (`.`) and that you always use fully qualified type names for static type-name syntax references.
- **Type-names for classes named as reserved keywords** — ABL allows a class to have the name of a reserved keyword. However, ABL does not fully support the use of static type-name syntax for classes whose names are reserved keywords. You can work around this limitation by using a package name to qualify any class whose name is a reserved keyword.

Note: Progress Software Corporation recommends that you always define a class in a package and always qualify the class name with the package name whenever you reference the class type.

Define and use widgets in classes

A class definition file can use all static widgets (for example, `BROWSE`, `BUTTON`, `FRAME`, `IMAGE`, `MENU`, `RECTANGLE`, `SUB-MENU`), as well as certain other static handle-based objects, including `STREAM` and `WORK-TABLE` objects. However, you can only define these handle-based objects as data members that are local to a class definition file, that is, as `PRIVATE` data members.

Also, because a class definition file does not support executable code in the main block, neither the WAIT-FOR statement nor conditional code is supported within the main block of a class definition file. The main block can only contain the CLASS statement, DEFINE statements, and ON statements. Typical graphical user interface (GUI) applications have a WAIT-FOR statement or include ON statements within conditional execution pathways.

Therefore, when using GUI objects you must follow these restrictions:

1. An ON statement cannot have any conditional code surrounding its definition.
2. The WAIT-FOR statement can only be specified in a method or constructor.
3. All GUI objects must be implicitly or explicitly defined as PRIVATE.

Given these rules, the following class definition file is valid:

```
CLASS Driver:
  DEFINE PRIVATE BUTTON msg.
  DEFINE PRIVATE BUTTON done.
  DEFINE FRAME f msg done.

  ON 'choose':U OF msg IN FRAME f DO:
    MESSAGE "click" VIEW-AS ALERT-BOX.
  END.

  CONSTRUCTOR PUBLIC Driver ( ):
    ModalDisplay ( ).
  END CONSTRUCTOR.

  METHOD PRIVATE VOID ModalDisplay ( ):
    ENABLE ALL WITH FRAME f.
    WAIT-FOR CHOOSE OF done.
  END METHOD.
END CLASS.
```

However, the following class definition file is **invalid**, because the lines marked with bold-faced comments (**#1** and **#2**) contain executable code:

```
CLASS Driver:
  DEFINE VARIABLE bGraphical AS LOGICAL NO-UNDO.
  DEFINE PRIVATE BUTTON msg.
  DEFINE PRIVATE BUTTON done.
  DEFINE FRAME f msg done.

/* #1 */
  IF bGraphical THEN DO:
    ON 'choose':U OF msg IN FRAME f DO:
      MESSAGE "click" VIEW-AS ALERT-BOX.
    END.
  END.
  ELSE DO:
    ON 'choose':U OF msg IN FRAME f DO:
      PUT "click" TO STREAM outstream.
    END.
  END.

  CONSTRUCTOR PUBLIC Driver ( ):
    ModalDisplay ( ).
  END CONSTRUCTOR.

  METHOD PRIVATE VOID ModalDisplay ( ):
    ENABLE ALL WITH FRAME f.
  END METHOD.

/* #2 */
```

```
  WAIT-FOR CHOOSE OF done.
END CLASS.
```

Use preprocessor features in a class

ABL supports preprocessor features in class definition files, with some restrictions. The following sections describe this support.

- Use compile-time arguments on page 196
- Use preprocessor names and directives on page 196

Use compile-time arguments

Compile-time arguments are a mechanism to pass compile-time values to an external procedure file or an include file. This facility is not supported for classes but is supported by include files referenced by a class, as shown:

```
CLASS Test:
  CONSTRUCTOR PUBLIC Test (INPUT in AS INTEGER):
    {build-info.i &build-num=num &build-date=today}  END CONSTRUCTOR.
END CLASS.
```

This is the contents of `build-info.i`:

```
MESSAGE {&build-num}
        {&build-date}
        VIEW-AS ALERT-BOX.
```

Use preprocessor names and directives

All built-in preprocessor names are supported in classes. These names include `BATCH-MODE`, `FILE-NAME`, `LINE-NUMBER`, `OPSYS`, `PROCESS-ARCHITECTURE`, `SEQUENCE` and `WINDOW-SYSTEM`. The following SpeedScript built-in preprocessor names are also supported: `DISPLAY`, `OUT`, `OUT-FMT`, `OUT-LONG` and `WEBSTREAM`.

All preprocessor directives, such as `&IF`, `&GLOBAL-DEFINE`, and `&SCOPED-DEFINE`, can be used in classes and behave as they do in procedures. `&SCOPED-DEFINE` defines a compile-time constant or preprocessor name that is in scope from the line that defines it, until the end of the source file that defines it (which may be a class, procedure, or include file). `&GLOBAL-DEFINE` defines a compile-time constant that is in scope from the line that defines it through to the end of the compilation unit. In the case of a class, this means until the end of the class definition, including any referenced include files.

There is no preprocessor directive that defines a compile-time constant that is in scope for the entire class hierarchy.

Raise and handle error conditions

Object-oriented error handling evolved from the *traditional error handling model*. This model primarily supports the management of OpenEdge system errors, with limited support for recording and raising application errors using `RETURN ERROR` in various ABL contexts. With this model, both system and application errors can raise a single `ERROR` condition. You can then trap this `ERROR` condition at different levels of an application and with varying degrees of control. Once trapped, you can further examine a system error using attributes and methods of the `ERROR-STATUS` system handle, or you can examine an optional setting of the `RETURN-VALUE` function for more information on an application error. In addition, if you do not handle an `ERROR` condition, the AVM displays an error message and standard ABL `UNDO` handling occurs, which can roll back transactions.

However, that model provides little or no context information about a given error and does so in an inconsistent manner. Depending on the context, you might have to use different mechanisms, other than raising the `ERROR` condition, for returning and handling errors that have no automatic effect on transactions. For example, built-in handle methods do not raise `ERROR` when called. So, to check for an error on a built-in handle method call, you must always check the `ERROR-STATUS` handle for the presence of a message after the call. You then might raise `ERROR` based on the message contents to affect a transaction.

You can continue to use the traditional error handling model because it is efficient and works well in many applications. However, ABL also supports an additional model for handling errors—the *structured error handling model*. This model provides a consistent mechanism for handling both application and system errors using standardized class-based error objects. It allows you to customize application error objects with application-specific data, and the model is fully compatible with the `ERROR` condition mechanisms of traditional error handling. You can thus use structured error handling alone or together with traditional error handling in both procedure-based and class-based applications.

The following sections provide a brief introduction to ABL structured error handling and its relationship with traditional error handling, and include specific information on raising and handling errors in a class-based application environment using either error handling model:

- Structured and traditional error handling on page 197
- Raise errors within a method on page 199
- Raise errors within a class event handler on page 200
- Raise errors within a property on page 201
- Raise errors within an instance constructor on page 203
- Raise errors within a static constructor on page 205
- Raise errors within a destructor on page 205

Structured and traditional error handling

As a general feature of computer languages, *structured error handling* is an error management model where an error is encapsulated by a class instance whose type depends on the type of error. Once created, this *error object* can be passed (*thrown*) to various parts of a computer program, where it might be examined (*caught*), modified, and optionally thrown to other parts of the program, all using the same standard mechanisms.

ABL supports structured error handling using the following basic elements:

- **Error classes** — A set of built-in and user-defined classes that inherit directly or indirectly from the `Progress.Lang.ProError` class. These classes are used to instantiate error objects that encapsulate various types of system and application errors. When any system error occurs, the AVM instantiates and

throws an instance of the built-in class Progress.Lang.SysError (subclass of Progress.Lang.ProError). You can also instantiate and throw an application error object, a Progress.Lang.AppError (subclass of Progress.Lang.ProError) using a RETURN ERROR. You can extend Progress.Lang.AppError with additional members to define new classes that encapsulate different types of application errors. You can then throw error objects instantiated from these classes like all the built-in error classes.

- **Error throwing mechanism** — An UNDO, THROW option of several ABL elements that raises ERROR and throws a specified error object to the *associated block* (the block where a particular ERROR condition is raised), which can handle the ERROR condition in a variety of well-defined ways. Using this option, you can throw a current system or application error object, or throw a new application error object that you create. Note that UNDO, THROW, by raising ERROR, also initiates the same UNDO handling as the traditional error handling model, when raising ERROR with a RETURN ERROR.
- **Error catching mechanism** — A CATCH statement that defines a special block that can catch and process an error object of any specified type that is thrown to the associated block. It is this *CATCH block*, which is available for use in all UNDO blocks, where you can decide whether to access the caught error object, throw the same or a new error object, or do nothing further with the error. Note that the CATCH statement is analogous to using the NO-ERROR option on a statement or the ON ERROR phrase (or default ON ERROR setting) on a block, both of which provide options for responding to ERROR conditions using traditional error handling. However, unlike the ON ERROR phrase, the CATCH block available with the CATCH statement is available for use in all UNDO blocks, including procedure, user-defined, function, and method blocks, and it allows, in one mechanism, far greater application control over many more kinds of error conditions than is possible with the NO-ERROR option and ON ERROR phrase.

Thus, structured error handling in ABL provides a more flexible and robust error handling model than traditional error handling, and it is often easier to use. You can, in fact, use both structured and traditional error handling in the same application. The two error models are fully compatible, even though each model handles errors differently. That is, using UNDO, THROW to throw an error object always raises the ERROR condition, and using RETURN ERROR to raise ERROR always throws an error object of one type or another; likewise, a system error both raises ERROR and throws a Progress.Lang.SysError object. Similarly, an error raised by any of these mechanisms can be handled by the error handling mechanisms of either model, including a CATCH block, the NO-ERROR option, or the ON ERROR phrase (or its default) setting.

In order to fully integrate the two models, ABL recognizes an order of precedence in how the mechanisms of traditional and structured error handling work together. Thus, when ERROR is raised in any way, ABL applies one of the possible mechanisms for handling that error in the associated block, depending on the mechanisms specified and available in a given context, and it chooses the mechanism according to the following order of precedence:

1. An appropriate NO-ERROR option
2. An appropriate CATCH statement
3. An explicit setting of the ON ERROR phrase for the block
4. The default setting of the ON ERROR phrase for the block, which can be changed on a routine-level block using the ROUTINE-LEVEL ON ERROR UNDO, THROW statement

For more information on error handling, including how to use both ABL structured and traditional error handling, see *ABL Error Handling*. The remainder of this section describes aspects of working with both error handling models in classes.

Raise errors within a method

When a method raises an error from a RETURN ERROR, an UNDO, THROW, or an unhandled system error the effects depend on the context of the method call. In general, the results of raising ERROR from within a non-VOID or a VOID method (or non-VOID method executed as a statement, ignoring the return value) are similar, and correspond to the results of raising ERROR from within an expression or a procedure, respectively.

So, if a non-VOID method raises ERROR in an expression, the ERROR is raised on the statement containing the expression, and the existing value of any receiving data element for the expression remains unchanged. In expressions that contain multiple method calls, methods execute in order from left to right. In the case of an error condition, all methods in the expression execute, in order, prior to the one that raises ERROR, and the existing value for any receiving data element for the expression, again, remains unchanged.

Method error handling example

The following class fragments from the sample classes show an example of method error handling:

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CreditObj:
  ...
  METHOD PUBLIC VOID SetCurrentCustomer (INPUT piCustNum AS INTEGER):
    /* Verify that this object has the current    */
    /* Customer before the property is referenced */
    FIND FIRST Customer WHERE Customer.CustNum = piCustNum NO-ERROR.
  END METHOD.

  METHOD PUBLIC VOID CheckCustCredit ( ):
    /* invokes the CustCreditLimit property SET accessor */
    IF AVAILABLE (Customer) THEN
      CustCreditLimit = Customer.Creditlimit.
    ELSE
      UNDO, THROW NEW Progress.Lang.AppError("No Customer").
  END METHOD.
END CLASS.
```

The CheckCustCredit( ) method of acme.myObjs.CreditObj throws an error object for two different conditions, which are distinguished by the error return string returned in the ReturnValue property of the Progress.Lang.AppError object. Note that the object returning the "No Customer" string is created using the NEW function, and another error object containing the "Over Limit" can be thrown from an error condition raised in the SET accessor of the CustCreditLimit property (see the code for this property in the). The ROUTINE-LEVEL ON ERROR UNDO, THROW statement that begins the class definition ensures that any error objects not caught and consumed by methods of the class are thrown to a method's caller.

Also note that to make a Customer record available, the SetCurrentCustomer( ) method must be successfully invoked prior to invoking CheckCustCredit( ). SetCurrentCustomer( ) relies on the NO-ERROR option of a FIND statement to complete with an error raised because no record is found with the specified criteria. So, instead of using structured error handling for this situation in both methods, CheckCustCredit( ) checks for record availability to handle that particular error.

The CheckCredit( ) method of acme.myObjs.CustObj then invokes these methods in the correct sequence and, if ERROR is raised, checks the value of the ReturnValue property on the error object to handle the error condition appropriately, as shown:

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.CustObj INHERITS CommonObj
                          IMPLEMENTS IBusObj:

  DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.
  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO
    FIELD RecNum  AS INTEGER
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name    LIKE Customer.Name
    FIELD State   AS CHARACTER.
  DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.
  DEFINE PRIVATE VARIABLE rMsg       AS CLASS MsgObj    NO-UNDO.

  ...

  METHOD PUBLIC VOID CheckCredit ( ):
    IF VALID-OBJECT (rCreditObj) THEN DO:
      FOR EACH ttCustomer:
        rCreditObj:SetCurrentCustomer (ttCustomer.CustNum).
        rCreditObj:CheckCustCredit ( ).

        /* invokes the CustCreditLimit property GET accessor */
        rMsg:InfoMsg(ttCustomer.Name + " is in good standing." +
          "  Credit Limit has been increased to " +
          STRING(rCreditObj:CustCreditLimit)).

        CATCH e AS Progress.Lang.AppError:
          IF e:ReturnValue = "Over Limit" THEN DO:
            /* invokes the CustCreditLimit property GET accessor */
            rMsg:Alert(ttCustomer.Name + " is on Credit Hold." +
              "  Balance exceeds Credit Limit of " +
              STRING (rCreditObj:CustCreditLimit)).
          END.
          ELSE
            rMsg:Alert ("Customer not found").
        END CATCH.
      END. /* FOR EACH */
    END.
    ELSE rMsg:Alert ("Unable to check credit").
  END METHOD.

  ...

END CLASS.
```

Raise errors within a class event handler

Error handling for ABL class event handlers is a special case of error handling for methods in general. If a class event handler raises ERROR (in any way), it does so within the context of a statement that calls the Publish( ) event method on the event it is handling. Thus, any unhandled error condition from an event handler that includes the ROUTINE-LEVEL ON ERROR UNDO, THROW statement is raised on the statement that published the event. In addition, once an error condition is raised, any handlers for the event that have not yet executed do not execute.

Raise errors within a property

Error handling for GET and SET accessors works similar to the way it does for methods. In this respect, a GET accessor corresponds to a method (non-VOID) that returns a value and a SET accessor corresponds to a method (VOID) that does not return a value. So, when you read or write a property in a statement, if the property accessor returns ERROR from a RETURN ERROR, an UNDO, THROW, or an unhandled system error, ABL raises the ERROR condition on this statement, which you can handle appropriately in the associated block.

Error handling within property accessors works exactly like it does for raising and handling errors within methods. For example, you can include blocks within property accessors to handle ERROR conditions, using the options of the ON ERROR phrase and you can include CATCH blocks in any UNDO block, including the accessor block.

Note: While you can include most any kind of processing, including UNDO processing, within a property accessor, Progress Software Corporation recommends that you employ the simplest processing possible within property accessors. Avoid any processing that involves long execution times, database processing, or any other processing that might result in unexpected side-effects.

Raise errors in a GET accessor

A GET accessor runs when the property appears in an expression or is passed as an INPUT parameter. You can therefore handle any ERROR that it raises similar to an ERROR raised from an expression.

Raise errors in a SET accessor

A SET accessor runs when you set the value of the property, for example, on the left side of an Assignment (=) statement or when you pass the property as an OUTPUT parameter. When a SET accessor raises ERROR, it can produce a different result from a property or method raising ERROR on the right side of the assignment.

Thus, after executing the following assignment statement, the iVal variable either changes to the value returned by *expression* or ERROR is raised on the statement from some problem in *expression*, in which case iVal remains unchanged at 5:

```
VAR INT iVal = 5.
iVal = expression.
```

However, in the following assignment statement, if the SET accessor for the iPropVal property does a RETURN ERROR or an UNDO, THROW, the value of iPropVal will retain its latest value prior to raising ERROR, which can well be the value returned by *expression*:

```
CLASS TestReturn:
  DEFINE PROPERTY iPropVal AS INTEGER INITIAL 5 NO-UNDO
    SET ... .
    iPropVal = expression.
END CLASS.
```

The actual value retained by iPropVal if its SET accessor raises ERROR depends on how you code the SET accessor.

Note: If a property appears on the left side of an assignment and the right side of the assignment raises ERROR, the SET accessor of the left-side property never runs.

When a property appears on the left side of an assignment and the SET accessor raises ERROR, the property can retain its latest value prior to raising ERROR depending on NO-UNDO settings, the property default memory, and any other data that the SET accessor changes. This is no different than any method that raises ERROR, except that the method (SET accessor) is executing on the left side of the assignment, independent of any right-side result.

In a similar fashion, when you pass a property as a routine OUTPUT parameter, if the property SET accessor raises ERROR, and depending on NO-UNDO settings, the SET accessor can leave all data that it affects at their latest values prior to raising ERROR.

Property error handling example

The following code fragment from the sample class, acme.myObjs.CreditObj, shows an example of property error handling:

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.
USING acme.myObjs.Common.*.

CLASS acme.myObjs.CreditObj:
  DEFINE PUBLIC PROPERTY CustCreditLimit AS DECIMAL INITIAL ? NO-UNDO
    /* GET: Returns the credit limit of the current Customer.       */
    /* If there is no current Customer, it returns Unknown (?).     */
    GET.
    /* SET: Raises the credit limit for Customers in good standing. */
    /* Current increase is $1,000.                                  */
    PROTECTED SET (INPUT piCL AS DECIMAL):
      IF Customer.Balance > piCL THEN DO:
        CustCreditLimit = Customer.Creditlimit.
        UNDO, THROW NEW Progress.Lang.AppError("Over Limit").
      END.
      ELSE
       ASSIGN
         Customer.Creditlimit = piCL + 1000.
         CustCreditLimit      = Customer.Creditlimit.
    END SET.
   ...

  METHOD PUBLIC VOID CheckCustCredit ( ):
    /* Invokes the CustCreditLimit property SET accessor */
    IF AVAILABLE (Customer) THEN
      CustCreditLimit = Customer.Creditlimit.
    ELSE
      UNDO, THROW NEW Progress.Lang.AppError( "No Customer" ).
  END METHOD.
END CLASS.
```

This class defines the CustCreditLimit property to set the credit limit for the current Customer record, depending on the customer's credit standing. The SET accessor raises ERROR by throwing an application error object if the customer's balance is over their current credit limit, by first setting the property value to that current limit, then creating the error object with a constructor that specifies an error return string ("Over Limit") to indicate this condition. For this error condition, note also that the SET accessor retains its setting because the property is defined as NO-UNDO, and it automatically throws the error object out of the accessor block to (raises ERROR on) the statement that sets the property value because the class is defined with the ROUTINE-LEVEL ON ERROR UNDO, THROW statement. However, if the customer's credit standing is good, the SET accessor immediately raises the customer's credit limit and sets the property with that new value.

So, as defined in the same class, the `CheckCustCredit( )` method sets the `CustCreditLimit` property with the customer's current credit limit and automatically re-throws any error object to the caller that is thrown from setting the property value. This error object is, in turn, caught and checked by the `CheckCredit( )` method in the sample class, `acme.myObjs.CreditObj`.

Raise errors within an instance constructor

There are two types of errors that can occur during the creation of an object:

- **Instantiation error** — Caused by any number of system errors, such as when the class file or a class file in the class hierarchy (source code or r-code) is not found on the `PROPATH` or there is an incorrectly coded signature for one of the constructors in the class hierarchy. For those instantiation errors that you do not or cannot handle within a constructor, the AVM automatically raises the `ERROR` condition and throws a `Progress.Lang.SysError` object on the constructor.

  Note: An improper signature can appear in a previously compiled class hierarchy if out-dated r-code for one of the classes is found and executed on the `PROPATH`. Thus, you must ensure that all classes in a class hierarchy are properly compiled. For more information, see Compile class definition files on page 248.

- **Application error** — Your application logic encounters an abnormal condition that it handles in one of the constructors in the class hierarchy using one of the available error handling mechanisms. For any such handled error, you can use a `RETURN ERROR` or an `UNDO, THROW` to raise the `ERROR` condition on the constructor.

Handle instance constructor errors

Whenever you or the AVM raise the `ERROR` condition on a constructor, the AVM raises that `ERROR` condition directly in the statement that instantiated the object using the `NEW` function, `NEW` statement, or `DYNAMIC-NEW` statement. Regardless of where in the class hierarchy the constructor is running, an `ERROR` raised on the constructor terminates object instantiation immediately. In addition, instead of returning an object reference to an instantiated object, the `ERROR` condition causes the `NEW` function to return no object reference, leaving any data element set to receive a value in any `NEW` or `DYNAMIC-NEW` statement unchanged.

Because `ERROR` raised on a constructor immediately terminates object instantiation, you cannot handle the `ERROR` condition on a corresponding `SUPER` or `THIS-OBJECT` statement that invokes the constructor from within another constructor. (You cannot specify `NO-ERROR` for either of these statements, nor can you use a `CATCH` block to handle errors raised on them.) You can only handle constructor errors on the statement that invokes a `NEW` or `DYNAMIC-NEW` to create the object.

Also, when a constructor raises the `ERROR` condition, along with terminating any further object instantiation, the AVM invokes the destructor, in reverse order of instantiation, for each super class whose constructor has already completed while instantiating the class hierarchy.

Note: Within a constructor, you can also use `DELETE OBJECT THIS-OBJECT` to abort class instantiation, which sets the returned object reference to the `Unknown value (?)`. However, Progress Software Corporation recommends that you use `RETURN ERROR` or `UNDO, THROW` to return `ERROR` from the constructor, instead, as this both aborts class instantiation and raises an `ERROR` condition that you can handle in the instantiating context.

Instance constructor error handling example

In the following contrived examples, one class (Derived) inherits behavior from another class (Base), each of which can halt object instantiation for different reasons that are each handled by the instantiating procedure (instantiater2.p):

```
CLASS Base:
  CONSTRUCTOR PUBLIC Base ( ):
    FIND FIRST Customer NO-ERROR.
    IF NOT AVAILABLE(Customer) THEN
      RETURN ERROR "No records found".
    /* Do any further Base class initialization */
  END CONSTRUCTOR.
END CLASS.
```

```
CLASS Derived INHERITS Base:
  CONSTRUCTOR PUBLIC Derived(INPUT iName AS CHARACTER, OUTPUT id AS INTEGER):
    SUPER( ). /* Unnecessary, but called for illustration */
    FIND Customer WHERE Customer.NAME BEGINS iName NO-ERROR.
    IF NOT AVAILABLE(Customer) THEN
      RETURN ERROR "Specified record not found".
    ELSE id = customer.cust-num.
    /* Do any further Derived class initialization. */
  END CONSTRUCTOR.
END CLASS.
```

```
/* instantiator2.p */

VAR INT id.
VAR CLASS Derived rDerived.

rDerived = NEW Derived ("fred", OUTPUT id).

/* Do normal application processing */ . . .

CATCH e AS Progress.Lang.AppError:
  IF e:ReturnValue = "Specified record not found" . . .
  ELSE IF e:ReturnValue = "No records found" . . .
END CATCH.
```

The instantiater2.p, procedure knows if the Derived class is not instantiated properly by catching any application error objects thrown for an ERROR condition raised on the NEW statement that instantiates the object. If there are no records in the Customer table, object instantiation fails when its super class constructor tries to find the first record in the table, which the constructor traps and raises ERROR using RETURN ERROR with an appropriate error return string. If the application-specified record is not found in the Customer table, object instantiation fails when the Derived class constructor traps the error and raises ERROR using RETURN ERROR with another appropriate error return string. In the latter case, the AVM also runs the default destructor for the Base class to undo its instantiation. The instantiating procedure, instantiater2.p, can then catch any thrown AppError object and check its ReturnValue property for the type of error that caused the instantiation failure. (RETURN ERROR automatically creates and throws a Progress.Lang.AppError object as part of raising ERROR.)

Thus, ABL error handling (structured or traditional) provides a common mechanism that you can use to handle errors at any level of an instantiating class hierarchy.

Raise errors within a static constructor

As described in a previous section (see the), when the static members of a class type are initialized, ABL executes the static constructor for every class in the class type hierarchy that has static members, starting in order from the root class (top of the hierarchy) to the most derived class of the class type (bottom of the hierarchy). If any static constructor in the hierarchy raises ERROR, ABL considers the load of the entire class type to have failed. If a class type fails to load, ABL does not attempt to load the class again, and all further references to that class type raise a run-time error. This is also true of all subclasses of the super class in the class type hierarchy whose static constructor first raises ERROR. That is, each class type below the class with the failed static constructor also fails to load. However, all super classes above the class whose static constructor raises ERROR do load successfully and these class types are all available to the ABL session. The only way to recover classes that have failed to load because of a static constructor fault is to restart the ABL session and re-run the application after the errors in the failed constructor have been fixed.

A static constructor can raise ERROR in the same way as in an instance constructor—from a RETURN ERROR, an UNDO, THROW, or an unhandled system error that throws an error object out of the constructor block. Any ERROR returned from a static constructor is raised on the statement that caused the static constructor to execute, whether it is the statement that first references a static member or the statement (such as the NEW statement) that first instantiates the class.

Raise errors within a destructor

ABL does not allow an error from a RETURN ERROR or an UNDO, THROW, or an unhandled system error, to be raised in a destructor beyond the level of the destructor block. This means that during the object destruction process, if the destructor is unable to perform its function, there is no mechanism for it to report a failure to the caller. The object is thus destroyed with no indication that an error occurred during the process.

Reflection

Reflection is the ability to get information about a class, interface, or enumeration type at run time. ABL provides reflection support using the built-in ABL class Progress.Lang.Class class, several built-in classes (e.g., Progress.Reflect.Method) designed to provide information about specific class members, and several built-in enumerations (e.g., Progress.Reflect.Flags) used to pass information to and from the reflection methods. This support allows you to get information about the type and hierarchy of a class-based object, as well as information about its members.

Use the Progress.Lang.Class class

This section describes how to get an instance of the Progress.Lang.Class class and how to use its public properties and methods.

Get a Progress.Lang.Class instance

You can get an instance of a Progress.Lang.Class containing the type information for a specified object type by invoking one of these two built-in ABL class methods:

- GetClass( ) instance method of Progress.Lang.Object invoked on a valid object reference of the specified object type

- `GetClass( )` static method of `Progress.Lang.Class` invoked on that class type with the object type specified as a character parameter

As an alternative to the static `GetClass( )`, you can use the `GET-CLASS` function, which takes an object type name instead of an object type specified as a character string.

This is the syntax to get a `Progress.Lang.Class` instance containing the specified type information:

Syntax

```
class-reference = { object-reference:GetClass( ) |
                    Progress.Lang.Class:GetClass( expression ) } .
```

Element descriptions for this syntax diagram follow:

class-reference

A data element defined as a `Progress.Lang.Class`.

object-reference

A reference to an instantiated class-based object whose type information you want returned in the `Progress.Lang.Class` instance.

expression

A `CHARACTER` expression that evaluates to the fully qualified object type name of the type whose information you want returned in the `Progress.Lang.Class` instance.

For example, after the calls to `GetClass( )` in the following procedure fragment, `rType1` and `rType2` each reference the same `Progress.Lang.Class` instance:

```
USING Progress.Lang.*.
USING acme.myObjs.*.

VAR CustObj rCustObj = NEW CustObj( ).
VAR Class rType1.
VAR Class rType2.

rType1 = rCustObj:GetClass( ).
rType2 = Progress.Lang.Class:GetClass("acme.myObjs.CustObj").
```

`rType1` is obtained using the instance version of `GetClass( )`, whereas `rType2` is obtained using the static version.

Use Progress.Lang.Class properties and methods

`Progress.Lang.Class` is a built-in ABL class that provides a set of properties and methods that return type information about a class, interface, or enumeration type. `Progress.Lang.Class` is `FINAL` and therefore it can not be inherited. The lifetime of these objects is controlled by the ABL session; therefore, you cannot delete them. In other words, using `DELETE OBJECT` on a `Progress.Lang.Class` instance will be ignored. However, if all ABL references and all references internally maintained by the AVM are gone, the object will be garbage collected.

The following table describes the public properties on `Progress.Lang.Class`.

Table 5: Progress.Lang.Class public properties

Property	Description
`TypeName`	A read-only property that contains the fully qualified class, interface, or enum type name. For example, `"acme.myObjs.CustObj"` (the sample class defined in Sample classes on page 225).
`Package`	A read-only property that contains the package (relative directory) of the class, interface, or enum type name. For example, `"acme.myObjs"` (the sample package used in Sample classes on page 225).
`SuperClass`	A read-only property that contains an object reference. This object reference represents the type of the super class if the current data type represents a subclass.

This table describes a sampling of the public methods available on `Progress.Lang.Class`. (Full descriptions of each of these methods as well as entries for additional methods not included here can be found in *ABL Reference*.)

Table 6: Progress.Lang.Class public methods

Method	Description
`GetClass( INPUT expression AS CHARACTER)`	Returns a `Progress.Lang.Class` instance containing information on the object type specified by *`expression`*. This method is static.

`GetConstructor ( INPUT params AS Progress.Lang.ParameterList )`
 	
`GetConstructor ( INPUT flags AS Progress.Reflect.Flags, INPUT params AS Progress.Lang.ParameterList )`
	Returns a `Progress.Reflect.Constructor` instance containing information on the constructor described by the input parameters.

`GetConstructors ( )`
 	
`GetConstructors ( INPUT flags AS Progress.Reflect.Flags )`
	Returns an array of `Progress.Reflect.Constructor` instances containing information on the constructors described by the input parameters.

`GetEvent ( INPUT event-name AS CHARACTER )`
 	
`GetEvent ( INPUT event-name AS CHARACTER, INPUT flags AS Progress.Reflect.Flags )`
	Returns a `Progress.Reflect.Event` instance containing information on

Method	Description
	the event described by the input parameters.
`GetEvents ( )` `GetEvents ( INPUT` *`flags`* `AS Progress.Reflect.Flags )`	Returns an array of `Progress.Reflect.Event` instances containing information on the events described by the input parameters.
`GetInterfaces ( )`	Returns an array of `Progress.Lang.Class` instances containing information on the interfaces that the class implements.

`GetMethod ( INPUT` *`method-name`* `AS CHARACTER,`
`INPUT` *`params`* `AS Progress.Lang.ParameterList )`
`GetMethod ( INPUT` *`method-name`* `AS CHARACTER,`
`INPUT` *`flags`* `AS Progress.Reflect.Flags,`
`INPUT` *`params`* `AS Progress.Lang.ParameterList )`
	Returns a `Progress.Reflect.Method` instance containing information on the method described by the input parameters.

`GetMethods ( )` `GetMethods ( INPUT` *`flags`* `AS Progress.Reflect.Flags )`	Returns an array of `Progress.Reflect.Method` instances containing information on the methods described by the input parameters.
`GetProperties ( )` `GetProperties ( INPUT` *`flags`* `AS Progress.Reflect.Flags )`	Returns an array of `Progress.Reflect.Property` instances containing information on the properties described by the input parameters.

`GetProperty ( INPUT` *`property-name`* `AS CHARACTER )`
`GetProperty ( INPUT` *`property-name`* `AS CHARACTER,`
`INPUT` *`flags`* `AS Progress.Reflect.Flags )`
	Returns a `Progress.Reflect.Property` instance containing information on the property described by the input parameters.

`GetVariable ( INPUT` *`variable-name`* `AS CHARACTER )`
`GetVariable ( INPUT` *`variable-name`* `AS CHARACTER,`
`INPUT` *`flags`* `AS Progress.Reflect.Flags )`
	Returns a `Progress.Reflect.Variable` instance containing information on the variable described by the input parameters.

`GetVariables ( )` `GetVariables ( INPUT` *`flags`* `AS Progress.Reflect.Flags )`	Returns an array of `Progress.Reflect.Variable` instances containing information on

Method	Description
	the variables described by the input parameters.
`HasStatics ( )`	Returns TRUE if a given class (or any of its super classes) has any static members.

`Invoke ( INPUT object-reference AS Project.Lang.Object,`
`  INPUT method-name AS CHARACTER )`
 	
`Invoke ( INPUT object-reference AS Project.Lang.Object,`
`  INPUT method-name AS CHARACTER,`
`  INPUT parameterlist-object`
`  AS Progress.Lang.ParameterList )`
 	
`Invoke ( INPUT method-name AS CHARACTER )`
 	
`Invoke ( INPUT method-name AS CHARACTER,`
`  INPUT parameterlist-object`
`  AS Progress.Lang.ParameterList )`
	Invokes a method when the method name and any parameters are only known at run time. There are four overloaded versions. For more information on using the `Invoke( )` method, see Dynamically invoke a method at run time on page 146.

`IsAbstract ( )`	Returns TRUE if a given class is abstract.
`IsFinal ( )`	Returns TRUE if the type is defined as `FINAL`.

`New ( )`
`New ( INPUT parameterlist-object`
`  AS Progress.Lang.ParameterList )`
	Instantiates a class when the class name and any parameters passed to its constructor are only known at run time. For more information on using the `New( )` method, see New( ) method on page 139.

A number of the methods of `Progress.Lang.Class` use classes in the `Progress.Reflect` package both as parameters and as a way of returning information. These are discussed in more detail in Identify classes in the Progress.Reflect package on page 211.

Use the Progress.Lang.ParameterList class

`Progress.Lang.ParameterList` is a built-in ABL class that provides a set of properties and methods in support of specific overloaded versions of the `Invoke( )` method and the `New( )` method of the `Progress.Lang.Class` class. A `Progress.Lang.ParameterList` object stores the parameter values for a specified method or constructor call and is used to pass all parameters. For example, if your method or class constructor takes five parameters, your application creates a `Progress.Lang.ParameterList` object defined to hold five parameters. The `Progress.Lang.ParameterList:SetParameter( )` method provides data for each parameter element. You can also use a `Progress.Lang.ParameterList` instance for invoking a method or instantiating a class via the `Progress.Reflect.Method` and `Progress.Reflect.Constructor` objects.

This section describes how to get an instance of the Progress.Lang.ParameterList class and how to use its public properties and methods.

Get and use a Progress.Lang.ParameterList instance

You can get an instance of a Progress.Lang.ParameterList class using the NEW statement. This is the syntax for creating a Progress.Lang.ParameterList instance.

Syntax

```
parameterlist-object = NEW Progress.Lang.ParameterList(n).
```

Element descriptions for this syntax diagram follow:

parameterlist-object

An instance of the Progress.Lang.ParameterList object.

n

An INTEGER expression indicating the number of parameters that the object contains. The argument must be greater than or equal to zero. A value of zero indicates that the method or constructor to which the Progress.Lang.ParameterList object is to be passed does not take any parameters. If a negative value is specified, the AVM raises a run-time error.

In this example, the rDynObj object is an instance of the class specified by pcDynObjTypeName, which in this case might have been CustomerObj, or OrderObj, or any other derived class of BusinessObject:

```
DEFINE INPUT PARAMETER pcDynObjTypeName AS CHARACTER NO-UNDO.

VAR INT iConstParam = 1000.
VAR CLASS BusinessObject rDynObj.
VAR CLASS Progress.Lang.Class rObjClass.
VAR CLASS Progress.Lang.ParameterList rParamList.

ASSIGN
  rObjClass  = Progress.Lang.Class:GetClass(pcDynObjTypeName)
  rParamList = NEW Progress.Lang.ParameterList(1).

rParamList:SetParameter(1, "INTEGER", "INPUT", iConstParam).
rDynObj = CAST(rObjClass:New(rParamList), BusinessObject).
```

Use Progress.Lang.ParameterList methods and properties

The Progress.Lang.ParameterList class contains several public properties and methods that allow you to manage parameter list objects. The following table describes the common methods and properties on the Progress.Lang.ParameterList class.

Table 7: Progress.Lang.ParameterList public properties and methods

Property or method	Description
`PUBLIC NumParameters AS INTEGER`	A readable/writable property that identifies the number of parameters in a `Progress.Lang.ParameterList` object
`METHOD PUBLIC LOGICAL Clear( )`	Releases all information in the `Progress.Lang.ParameterList` object and sets the number of parameters to zero
`METHOD PUBLIC LOGICAL SetParameter ( position, data-type, iomode, parameter-value )`	Populates elements in the `Progress.Lang.ParameterList` object

For more information about the `Progress.Lang.ParameterList` class and its public properties and methods, see *ABL Reference*. For more information on using the `Invoke( )` method, see Dynamically invoke a method at run time on page 146. For more information on using the `New( )` method, see New( ) method on page 139.

Identify classes in the Progress.Reflect package

A number of the `PUBLIC` methods of `Progress.Lang.Class` use classes in the `Progress.Reflect` package both as parameters and as a way of returning information. The "Get" reflection methods return instances of classes to describe class members. For example, if you call `GetMethod( )` to get information about a particular method of a class, `GetMethod( )` returns an instance of `Progress.Reflect.Method` representing the method. The following table describe some of the public properties of `Progress.Reflect.Method`.

Table 8: Progress.Reflect.Method public properties

Property	Description
`AccessMode` AS `Progress.Reflect.AccessMode`	Returns an instance of the built-in enumeration `Progress.Reflect.AccessMode` indicating the access mode of the method.
`IsOverride` AS `LOGICAL`	Returns `TRUE` if the method is an override.
`Name` AS `CHARACTER`	Returns the method name.
`OriginatingClass` AS `Progress.Lang.Class`	Returns an instance of `Progress.Lang.Class` representing the class the method was retrieved from.

As mentioned in the table, `Progress.Reflect.AccessMode` is a built-in enumeration type. For more information about working with ABL enumeration types, see the `ENUM` statement entry in *ABL Reference*.

`Progress.Reflect.Method` also provides methods, including an `Invoke( )` method (with overrides for both static and instance methods) that allows you to call the method that the `Progress.Reflect.Method` instance represents. This is the syntax for the instance method override:

```
Invoke ( INPUT instance AS Progress.Lang.Object,
         INPUT params AS Progress.Lang.ParameterList )
```

This override takes an instance of the class that implements the method and a parameter list to hold the parameters that are passed to the method.

The following code uses reflection methods to add two pieces of data to a JSON array and then write that array to a file, following these steps:

1. `GetClass( )` is invoked statically to create `plcjArray`, a `Progress.Lang.Class` instance representing a `Progress.Json.ObjectModel.JsonArray` object. The `GetConstructor( )` method of `Progress.Lang.Class` is invoked to create `jArray`, an instance of `JsonArray`.
2. `GetMethod( )` is invoked on `plcjArray` to create `myMethod`, an instance of `Progress.Reflect.Method`, which in this case represents the `Add( )` method of `JsonArray`. Then `Invoke( )` is called on `myMethod`, which invokes `Add( )` to add the character data to `jArray`.

3. After getting the method again using a revised parameter list, `Invoke( )` is called on `myMethod` to add the date to `jArray`.
4. `GetMethod( )` is invoked again on `plcjArray`, this time to create an instance of `Progress.Reflect.Method` that represents the `WriteFile( )` method of `JsonArray`. `Invoke( )` is called on `myMethod` to write `jArray` to file.

```
/* Use GetMethod() and Invoke() to add data to a JsonArray */
USING Progress.Json.ObjectModel.* FROM PROPATH.
USING Progress.Reflect.*.

VAR Progress.Lang.Class plcjArray.
VAR Constructor myConstructor.
VAR Method myMethod.
VAR CLASS JsonArray jArray.
VAR Progress.Lang.Parameterlist myPL.

VAR CHAR char1 = "Mr. Smith".
VAR DATE date1 = 11/16/2010.

/* 1. Create a JsonArray object. */
plcjArray =
   Progress.Lang.Class:GetClass("Progress.Json.ObjectModel.JsonArray").
myPL = NEW Progress.Lang.Parameterlist(0).
myConstructor = plcjArray:GetConstructor(myPL).
jArray = CAST(myConstructor:Invoke(myPL), Progress.Json.ObjectModel.JsonArray).

/* 2. Get and invoke Add( ) to add the name. */
myPL = NEW Progress.Lang.Parameterlist(1).
myPL:SetParameter(1, "CHARACTER", "INPUT", char1).
myMethod = plcjArray:GetMethod("Add", myPL).
myMethod:Invoke(jArray, myPL).

/* 3. Get and invoke Add( ) again to add the date */
myPL:SetParameter(1, "DATE", "INPUT", date1).
myMethod = plcjArray:GetMethod("Add", myPL).
myMethod:Invoke(jArray, myPL).

/* 4. Get and invoke WriteFile( ). */
myPL = NEW Progress.Lang.Parameterlist(2).
myPL:SetParameter(1, "CHARACTER", "INPUT", "filename.json").
myPL:SetParameter(2, "LOGICAL", "INPUT", TRUE).
myMethod = plcjArray:GetMethod("WriteFile", myPL).
myMethod:Invoke(jArray, myPL).
```

For full descriptions of the properties and methods of `Progress.Reflect.Constructor`, `Progress.Reflect.Method`, `Progress.Reflect.Event`, `Progress.Reflect.Property`, and `Progress.Reflect.Variable`, see their entries in *ABL Reference*.

Serialization and deserialization

Serialization is a way of saving a class-based object so the object can be recreated later or in a different context. The serialization process stores the values of an object's instance data members, properties, temp-tables, and ProDataSets. Deserialization is the process of then recreating the object using those stored values.

You can take advantage of the serialization/deserialization process for several use cases. For example, you can :

- pass an object between an application server and a remote ABL client
- save the objects that make up the context of an application server session, so that context is available no matter which agent happens to handle subsequent requests from the client
- save the objects that make up the context of an ABL client so that a given task can be resumed at a later time.

In the case of passing objects between an application server and ABL client, the AVM handles the actual serialization/deserialization process, but for the other cases, you must manage the process, using the built-in file stream classes and the serialization/deserializaton classes.

Serialize an instance to file

Before an instance can be serialized to file, some basic conditions must be met:

- The class of the instance you want to serialize, and all of that class's super classes, must be defined using the `SERIALIZABLE` keyword, as described in Use the CLASS construct on page 54.
- If there are any class members that are themselves class-based objects, those classes must also be defined as `SERIALIZABLE`. (Alternatively, the class members themselves can be defined as `NON-SERIALIZABLE`, and the AVM will skip those during serialization. See the entries for each `DEFINE` statement to learn more.)
- Any class members you want to include in the serialization that are not included by default must be defined using the `SERIALIZABLE` keyword.

The ABL offers a choice of two formats for a serialized instance: binary and JSON. Aside from the format itself, these two formats differ in default behavior. For example, binary serialization includes all data members, properties, temp-tables, and ProDataSets by default, regardless of their scope, whereas JSON serialization only includes those defined as public. The steps given below for serializing an instance use binary format, but the same general process can be used for JSON.

To serialize an instance:

1. Instantiate a variable of type `Progress.IO.BinarySerializer` (or `Progress.IO.JsonSerializer`).
2. Instantiate a variable of type `Progress.IO.FileOutputStream` with your target filename as a parameter. This class provides functionality for writing to the file.
3. Invoke the `Serialize( )` method of your `Progress.IO.BinarySerializer` instance and pass it the object to be serialized and the `Progress.IO.FileOutputStream` object.
4. Invoke the `Close( )` method of the `Progress.IO.FileOutputStream` object.

The following code example shows `myObj`, an instance of `Acme.MyClass`, being serialized to a binary file `MyClass.bin`. The corresponding steps are numbered with comments.

```
VAR Acme.MyClass myObj = NEW Acme.MyClass().

/* 1 */
VAR Progress.IO.BinarySerializer mySerializer = NEW Progress.IO.BinarySerializer().

/* 2 */
VAR Progress.IO.FileOutputStream myFileOutStream = NEW
Progress.IO.FileOutputStream("MyClass.bin").

/* 3 */
mySerializer:Serialize(myObj, myFileOutStream).

/* 4 */
myFileOutStream:Close().
```

Beyond the special cases mentioned above, there are additional restrictions and special behaviors for serializing some types of data. Refer to the full list of these in the Pass object reference parameters on page 159 section.

Deserialize from a file

To deserialize an instance:

1. Instantiate a variable of type `Progress.IO.BinarySerializer` (or `Progress.IO.JsonSerializer`).
2. Instantiate a variable of type `Progress.IO.FileInputStream` with your target filename as a parameter. This class provides functionality for reading from the file.
3. Invoke the `Deerialize( )` method of your `Progress.IO.BinarySerializer` instance and pass it the filename from which the object will be deserialized. The the `Deserialize( )` method returns an instance of type `Progress.Lang.Object`, so you must use the CAST function to set the deserialized object to the correct type.
4. Invoke the `Close( )` method of the `Progress.IO.FileInputStream` object.

The following code example shows `myObj`, an instance of `Acme.MyClass`, being deserialized from a binary file `MyClass.bin`. The corresponding steps are numbered with comments.

```
VAR Acme.MyClass myObj = NEW Acme.MyClass().

/* 1 */
VAR Progress.IO.BinarySerializer mySerializer = NEW Progress.IO.BinarySerializer().

/* 2 */
VAR Progress.IO.FileInputStream myFileInStream = NEW
Progress.IO.FileInputStream("MyClass.bin").

/* 3 */
myObj = CAST(mySerializer:Deserialize(myFileInStream), Acme.MyClass).

/* 4 */
myFileInStream:Close().
```

For full descriptions of all the classes and methods used in serializing and deserializing objects, see their entries in *ABL Reference*.

5

Program with class-based and procedure objects

The CLASS and INTERFACE statements provide a clear distinction in ABL between a class (or interface) and a procedure. Any source file (or larger compilation unit, if you consider the use of include files) that contains the CLASS statement defines a class; any file that contains an INTERFACE statement defines an interface, and any file that does not contain a CLASS or INTERFACE statement defines an external procedure.

As you have learned, there are several key differences in the language statements you use in procedures and the statements you use in classes. However, always keep in mind that the vast majority of ABL statements are usable in both procedures and classes, and in general have the same behavior, with the exceptions noted as part of this and other topics.

For details, see the following topics:

- Class-based and procedure object compatibility
- Compare handles and object references
- Use handle-based object events in classes
- Use widget pools
- Reference routines on the call stack
- Compare constructs in classes and procedures
- Sample classes
- Comparative procedures
- Summary comparison of classes and procedures

Class-based and procedure object compatibility

ABL allows class-based objects and procedures to co-exist and work together to form a complete application. This section describes the features and limitations of mixing class-based objects and persistent procedures.

Each class defines a unique data type (object type). You must therefore use the appropriate syntax to instantiate a class-based object of a specified type and execute its methods. You can use this syntax to instantiate a class within a procedure as well as within another class. In other words, a procedure can use the `NEW` function to create an instance of class, work with it, and delete it when it is no longer needed. The procedure can pass a reference to the object as a parameter in any procedure or user-defined function call. In addition, a user-defined function can define its return value to be an object reference. Classes and class-based objects are just another development tool that you have at your disposal.

From the opposite perspective, a method in a class can run a procedure using the same procedure-based syntax for running internal procedures within external procedures. A class can run a procedure persistently and save a handle to it just as another procedure can do. The class can run internal procedures and user-defined functions using that handle. It can delete the procedure when it is no longer needed. A method in the class can pass a procedure handle as a parameter or use it as a return value.

In summary, the restrictions that apply to classes and procedures apply to a limited set of definitional statements that are used in one or the other, not to the way in which they can interact with each other.

Compatibility rules

Since an object reference and a procedure handle represent two very different constructs in ABL, the following rules apply when working with both:

- You cannot execute r-code for a class using the `RUN` statement. You must use the `NEW` function to create a class instance.
- You can only use the `NEW` function to create an instance of a class. You cannot use the `NEW` function to run a persistent procedure.
- An instance of a class never has a procedure object handle associated with it. Thus, a class does not use system handles, such as `THIS-PROCEDURE` or `TARGET-PROCEDURE`, that return procedure object handles for referencing the r-code of a referencing procedure. Using any of these keywords in a class results in a compile-time error.
- A procedure has no notion of a class or object reference. Thus a procedure cannot use system references, such as `THIS-OBJECT`, that return references to the r-code of a referencing class or class hierarchy. Using any of these keywords in a procedure results in a compile-time error.
- You cannot assign a procedure handle to an object reference. You cannot assign an object reference to a procedure handle. Thus, while you can cast object reference types from one to another, you cannot cast an object reference type to a procedure handle or in any way convert one to the other.
- You cannot pass an object reference to a routine expecting a procedure handle. Likewise, you cannot pass a procedure handle to a routine expecting an object reference. As noted, there is no way to cast a procedure handle to an object reference or an object reference to a procedure handle.
- You cannot define methods in a procedure. Procedures define their internal entry points as internal procedures or user-defined functions only. You can define methods only in classes. Likewise, you cannot define internal procedure or user-defined functions in procedures, but only in external procedures.
- You cannot use the `object-reference:method-name` syntax to run an internal procedure or a function, but only a method. You can use this syntax, however, in both classes and procedures.

- You cannot define a constructor or destructor for a procedure. You can define these special methods only for classes.
- You cannot specify a PRIVATE, PACKAGE-PRIVATE, PROTECTED, PACKAGE-PROTECTED or PUBLIC access mode on a variable or other data definition except in the main block of a class.
- You cannot use the ABL built-in ADD-SUPER-PROCEDURE( ) method on the THIS-OBJECT system reference or the THIS-PROCEDURE system handle within a class or otherwise use the super procedure mechanism to create a hierarchy of object references. Thus, you cannot add a class-based object as a super procedure. However within a class, you can use the SESSION:ADD-SUPER-PROCEDURE( ) method to extend the super procedure chain of the SESSION handle, and you can use the *procedure-handle*:ADD-SUPER-PROCEDURE( ) method, where *procedure-handle* is a procedure object handle set from running a persistent procedure, in order to extend the super procedure chain for other, procedure objects.

Invalid ABL within a user-defined class

Several ABL statements have been identified as deprecated (obsolete) language features. The life-cycle of products and features are described in the OpenEdge 12 Platform and Product Availability Guide on the Progress Information Hub.

The product direction states that applications using these languages elements will continue to function but it is strongly recommended that applications should stop using these features. In addition, the guide identifies that these features will not get bug fixes or any enhancements.

From these ABL deprecated features, classes do not support the following language elements at all. If the compiler encounters any one of these statements in a class, it generates an error:

- Support for SQL within ABL:
  - ALTER TABLE statement
  - CLOSE statement
  - CREATE INDEX statement
  - CREATE SCHEMA statement
  - CREATE TABLE statement
  - CREATE VIEW statement
  - DECLARE CURSOR statement
  - DELETE FROM statement
  - DROP INDEX statement
  - DROP TABLE statement
  - DROP VIEW statement
  - FETCH statement
  - GRANT statement
  - INSERT INTO statement
  - OPEN statement
  - REVOKE statement

  - SELECT statement
  - UNION statement
  - UPDATE statement

- CHOOSE statement
- EDITING phrase in UPDATE, SET, PROMPT-FOR statements
- GATEWAYS function
- GO-PENDING function
- IS-ATTR-SPACE function
- PUT SCREEN statement
- SCROLL statement

Verify the source for an r-code file at run time

Because ABL generates a common r-code (.r) file type for both classes and procedures, you cannot necessarily tell from the filename if a given r-code file is generated from a class or a procedure. If your application uses both procedures and classes, you might need to verify whether the source for a given r-code file is a class definition (.cls) or procedure (.p or .w) file. This can be useful, for example, if your application is or provides a tool for managing ABL classes and procedures in some way.

To distinguish r-code files that are generated for classes and procedures, ABL supports a read-only LOGICAL attribute, IS-CLASS, on the RCODE-INFO system handle. Thus, if the IS-CLASS attribute is TRUE for a specified r-code file, the source is a class definition file that defines a class or interface. If the attribute is FALSE, the source is a procedure source file.

Compare handles and object references

A procedure object handle always references a running procedure instance, and is stored in a variable or field of type HANDLE. An object reference always points to an instance of a class and is stored in a variable that is defined as that class, a super class, or related interface type, or in a temp-table field of type Progress.Lang.Object. These two kinds of references—procedure object handles and object references—are not interchangeable.

Use handles

HANDLE variables and HANDLE temp-table fields are weakly-typed. A handle can be used as a reference to virtually any kind of handle-based object, such as a procedure, a visual object (widget), or a data object such as a buffer or query. Because handle references are set at run time, the AVM does not (and indeed cannot) verify much of anything about how a handle is used at compile time. As a result, you can easily use a handle improperly, for example, invoking a buffer object method on a procedure handle object—an error that the AVM can only catch at run time. In addition, you must explicitly delete all handle-based objects that you create using a DELETE OBJECT statement or another appropriate DELETE*keyword* statement, or you risk a memory leak.

Use object references

Because classes in ABL are strongly-typed, ABL can check at compile-time if an object reference is valid. However, you can only reference an object of a compatible type. You can allow a single variable or a temp-table field to hold a reference to a variety of types using an object reference of an appropriate super class type. If you define a variable or a temp-table field as the class type `Progress.Lang.Object`, you can use it to store an object reference for any class type. If you want to access public data members, properties, methods, or events defined for the object as a subclass of the specified class type, you can cast the object reference to the type of that subclass (or lower in the class hierarchy). Otherwise, ABL catches any attempt to reference this subclass functionality as a compiler error. For more information on casting, see Object reference assignment and casting on page 179. In addition, ABL automatically garbage collects any class-based objects that you create; it is not necessary to explicitly delete them.

If you need to keep track of a collection of object references, for example for all the class instances running in a session, you can create a temp-table with a field of type `Progress.Lang.Object`. You can then store any object reference in that field. For more information on defining and using temp-table fields to reference class-based objects, see Define an object reference field in a temp-table on page 166.

You can then use the temp-table to retrieve a `Progress.Lang.Object` reference and cast it to the appropriate class or you can use built-in `Progress.Lang.Object` methods to return information about the object. For example, to display the class type name for each object reference saved in a temp-table you might write the following code fragment:

```
DEFINE TEMP-TABLE ttObjHolder NO-UNDO
  FIELD objRef AS CLASS Progress.Lang.Object.
  ...

FOR EACH ttObjHolder:
  MESSAGE objRef:GetClass( ):TypeName.
END.
```

Use handle-based object events in classes

ABL supports events with several different constructs. Classes support widget and built-in object events using the `ON` statement, and they support ProDataSet, query, buffer, socket object and server socket object callbacks using the `SET-CALLBACK( )` method. Classes also allow you to publish named events associated with external procedures using the `PUBLISH` statement's `FROM` *`publisher-handle`* option and they allow you to manage subscriptions of internal procedures as event handlers for named events using the `SUBSCRIBE` and `UNSUBSCRIBE` statements' `PROCEDURE` *`subscriber-handle`* option. However, you cannot subscribe class methods to named events as event handlers or otherwise provide behavior to a class using named events. Classes support class events as members that provide a similar form of event behavior for a class. For more information, see Define events within a class on page 77.

ON statement

Class definition files can define static visual handle-based objects (widgets) as PRIVATE data members. These built-in widgets expose a well-defined set of events that an application can respond to, by using the ON statement. The ON statement is supported in the main block of a class definition file. As noted previously, the main block cannot contain executable code. Therefore a class definition file cannot specify an ON statement within executable conditional logic, nor can it contain a WAIT-FOR statement in the main block. However, the main block can contain ON statements that respond to a specific event that occurs on a static visual object.

The following contrived code demonstrates this capability:

```
CLASS Disp:

  /* These data members are PRIVATE by default. */
  DEFINE BUTTON msg.
  DEFINE BUTTON done.
  DEFINE FRAME f msg done.

  /* Unconditional ON block valid in the main block. */
  ON 'choose':U OF msg
    MESSAGE "click" VIEW-AS ALERT-BOX.

  CONSTRUCTOR PUBLIC Disp ( ):
    modal_display ( ).
  END CONSTRUCTOR.

  /* WAIT-FOR is done in a method. */
  METHOD PRIVATE VOID modal_display ( ):
    ENABLE ALL WITH FRAME f.
    WAIT-FOR CHOOSE OF done.
  END METHOD.

END CLASS.
```

SET-CALLBACK( ) method

The SET-CALLBACK( ) method is used to associate a method and object reference with ABL-defined events on the following objects:

- ProDataSet object
- Buffer object
- Query object
- Socket object
- Server Socket object

Note: You can also use SET-CALLBACK( ) to associate an internal procedure with a named ABL event.

This is the syntax for using SET-CALLBACK( ) to associate a method and object reference (or internal procedure and procedure handle) with a named ABL event:

Syntax

```
SET-CALLBACK ( callback-name, method-or-proc-name
  [ , method-or-proc-context ] )
```

Element descriptions for this syntax diagram follow:

callback-name

A quoted string or character expression representing the name of a callback (named ABL event). The callback name is not case-sensitive. For information on the supported callbacks, see the SET-CALLBACK( ) method reference entry in *ABL Reference*.

method-or-proc-name

A quoted string or character expression representing the name of an internal procedure or method that resides within *method-or-proc-context*.

method-or-proc-context

A reference to a class instance that contains the method specified by *method-or-proc-name* or a handle to a procedure that contains the internal procedure specified by *method-or-proc-name*. If not specified, THIS-PROCEDURE is used if SET-CALLBACK( ) is executed within a procedure, and THIS-OBJECT is used if SET-CALLBACK( ) is executed within a class.

Note:

- If *method-or-proc-name* represents a class method that is associated with an ABL event as a callback, and the method is overridden in a derived class, the overridden method is run when the callback is invoked.
- Only public methods and public procedures may be used as callback handlers. Using private procedures, private methods, and protected methods will generate a runtime error.

Use widget pools

Classes instantiated using the NEW function cannot be created in a widget pool. You can assign the object reference to each instantiated class-based object to an object reference data element.

Within a procedure file, many dynamic handle-based objects can be created using one of three memory allocation strategies. A dynamic handle-based object can be created in an unnamed widget pool, a named widget pool, or the system's unnamed widget pool. The IN WIDGET-POOL phrase of the CREATE*handle-based-object* statement (for example, CREATE BROWSE or CREATE TEMP-TABLE) controls which of these three memory pools the resources for the dynamic handle-based object go into.

Within a class file, the existing rules continue to apply for how widget pools behave and from which widget pool memory is allocated for dynamic handle-based objects. Thus, you can create zero or more of the following kinds of widget pools:

- Named widget pools within the methods of a class
- Unnamed widget pools within the methods of a class
- A single unnamed widget pool scoped to the entire class

If you create a named widget pool in a method, dynamic handle-based objects will only be created in the pool if they explicitly reference that widget pool by name.

If you create unnamed widget pools within a class, the existing rules for how dynamic handle-based objects get created in an unnamed widget pool apply. These rules state that a dynamic handle-based object is created in the most locally scoped unnamed widget pool, if one has been created, and in the system unnamed widget pool, if no unnamed widget pool has been created. Unnamed widget pools created in a method are scoped to the execution lifetime of that method. Thus, unnamed widget pools created within a method can be explicitly deleted within the method using the `DELETE WIDGET-POOL` statement, or they are implicitly deleted by the ABL session when method execution ends.

In addition for classes, you can create two additional unnamed widget pools by specifying the `USE-WIDGET-POOL` option on the `CLASS` statement. (For more information, see Use the CLASS construct on page 54.) This creates a single, unnamed widget pool for all handle-based objects created using:

- Instance members that are scoped to the class instance. This unnamed widget pool is implicitly deleted by the ABL session when the class instance to which it is scoped is deleted.
- Static members that are scoped to the class type. This unnamed widget pool is implicitly deleted only when the ABL session in which the widget pool is created terminates.

If a class that does not specify the `USE-WIDGET-POOL` option inherits (either directly or indirectly) from a class that does specify the option, the subclass inherits the `USE-WIDGET-POOL` option, also. Furthermore, if a class does specify the `USE-WIDGET-POOL` option, the option applies to any classes that it is derived from when they are running as part of an instance of the class. In other words, at run time, an object has an unnamed widget pool scoped to it if any class in its hierarchy is defined with the `USE-WIDGET-POOL` option. As already noted, an unnamed widget pool created in one of an object's methods takes precedence over an object's unnamed widget pool during the lifetime of the method's widget pool.

Reference routines on the call stack

The `PROGRAM-NAME` built-in function returns the name of a routine on the call stack. It takes an integer argument that indicates the level of the call stack to be return. If the argument is 1, the name of the current routine is returned. If it is 2, the name of the calling routine is returned. If there is no calling routine, the application is at the top of the call stack and the AVM returns the `Unknown value (?)`.

If a position on the call stack contains a method reference, `PROGRAM-NAME` returns a string in the following syntax:

```
"method-nameclass-file-name"
```

Element descriptions for this syntax diagram follow:

`method-name`

The calling method.

`class-file-name`

The class file where the calling method is implemented.

Compare constructs in classes and procedures

The examples in the following sections compare the use of classes and inheritance with the use of procedures and super procedures. Almost the same application is implemented using a set of sample classes and procedures, which checks, adjusts, and reports on customer credit limits. A summary comparison between the class-based and procedure-based versions follows the listing of all the classes and procedures. The commented numbers in the code match code-numbered comments in the summary comparison.

Note that this is not a real-world application. It is designed solely to demonstrate object-oriented programming concepts, and it includes one possible implementation using procedure-based programming as a point of comparison with the class-based implementation. As a result, some constructs are artificially coded for point-by-point illustration and comparison. Little of this code is intended to demonstrate best practices for designing and implementing ABL applications. Indeed for brevity, some best practices might be intentionally avoided to illustrate a concept. For information on best practices, see the resources available on PSDN: http://communities.progress.com/pcom/community/psdn/openedge.

These sample classes and procedures are also available for compiling and running in your OpenEdge development environment.

Both versions of this sample application rely on the `Customer` table of the `sports2020` database installed with OpenEdge. The samples are coded to filter out all but a few test records in the database. Before running these samples, you can also run the `IncreaseBalance.p` procedure that is duplicated for convenience in both the `classes` and `procedures` directories. This procedure updates some of the filtered test records to ensure that they have `Balance` fields with values greater than their corresponding `CreditLimit` fields. Also duplicated in these directories is a text file of sample E-mail addresses (`email.txt`) that is referenced by the samples. Be sure to move this file to the working directory that is appropriate for your environment.

Sample classes

The class-based sample application consists of several class definition files and a procedure file. These files define several classes, an interface, and a procedure that drives the class-based application. The following is a description of the relationships among the classes, interface, and procedure, using unqualified names for the application classes and interface for readability:

```
Abstract Class CommonObj INHERITS Class Progress.Lang.Object
    instantiates and uses Class MsgObj

  Class CustObj INHERITS Abstract Class CommonObj
            and IMPLEMENTS Interface IBusObj
    instantiates and uses Class CreditObj
    instantiates and uses Class MsgObj

    Class NECustObj INHERITS Class CustObj

Class MsgObj INHERITS Class Progress.Lang.Object

Class CreditObj INHERITS Class Progress.Lang.Object

Class HelperClass INHERITS Class Progress.Lang.Object
  instantiates and uses Class MsgObj
  uses Abstract Class CommonObj
  uses Class CustObj
  uses Interface IBusObj

Class Main INHERITS Class Progress.Lang.Object
```

```
  instantiates and uses Class HelperClass
  instantiates and uses Class CustObj
  instantiates and uses Class NECustObj
  uses Abstract Class CommonObj
  uses Interface IBusObj

Procedure Driver.p
  instantiates and uses Class Main
```

The descriptions and code listings for these files follow.

This is the top-level user-defined abstract super class that provides a common method and variable for storing time-tracking information. It also provides abstract definitions for a class event and event publishing method, and for a message handler method, all to be implemented by classes that inherit from it:

CommonObj.cls

```
USING acme.myObjs.Common.*.

CLASS acme.myObjs.Common.CommonObj ABSTRACT:

  VAR PROTECTED DATETIME dtTimestamp.                              /* 5 */

  METHOD PUBLIC DATETIME updateTimestamp ( ):
    dtTimestamp = NOW.
    RETURN dtTimestamp.
  END METHOD.

  DEFINE PUBLIC ABSTRACT EVENT OutputGenerated                    /* 16 */
    SIGNATURE VOID (pcOutputType AS CHARACTER).

  METHOD PROTECTED ABSTRACT VOID PublishOutputGenerated           /* 16 */
    (INPUT pcOutputType AS CHARACTER).

  METHOD PROTECTED ABSTRACT CLASS MsgObj MessageHandler           /* 3 */
    (INPUT piObjType AS CHARACTER).

END CLASS.
```

This interface is implemented by the following class, `acme.myObjs.CustObj`:

IBusObj.cls

```
INTERFACE acme.myObjs.Interfaces.IBusObj:

  METHOD PUBLIC VOID printObj ( ).                                /* 11 */
  METHOD PUBLIC VOID printObj (INPUT pcCopies AS CHARACTER).
  METHOD PUBLIC VOID logObj (INPUT pcFilename AS CHARACTER).

END INTERFACE.
```

This class extends `acme.myObjs.Common.CommonObj` to provide general functionality for handling customers and is a super class for the `acme.myObjs.NECustObj` class, which handles New England customers:

CustObj.cls

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.
```

```
CLASS acme.myObjs.CustObj INHERITS CommonObj                          /* 1 */
                          IMPLEMENTS IBusObj:

  VAR PUBLIC INT iNumCusts.

  DEFINE PROTECTED TEMP-TABLE ttCustomer NO-UNDO                      /* 14 */
    FIELD RecNum  AS INTEGER
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name    LIKE Customer.Name
    FIELD State   AS CHARACTER.
  VAR PRIVATE CLASS MsgObj rMsg.
  VAR PRIVATE CLASS CreditObj rCreditObj.

  DEFINE PUBLIC OVERRIDE EVENT OutputGenerated                        /* 16 */
    SIGNATURE VOID (pcOutputType AS CHARACTER).

  METHOD PROTECTED OVERRIDE VOID PublishOutputGenerated               /* 16 */
    (INPUT pcOutputType AS CHARACTER):
    OutputGenerated:Publish(pcOutputType).
  END METHOD.

  CONSTRUCTOR PUBLIC CustObj ( ):                                     /* 2 */
    rCreditObj = NEW CreditObj ( ).
    iNumCusts = 0.
    /* Fill temp table and get row count */
    FOR EACH Customer NO-LOCK WHERE Customer.CreditLimit > 50000:
      CREATE ttCustomer.
      ASSIGN
        iNumCusts          = iNumCusts + 1
        ttCustomer.RecNum  = iNumCusts
        ttCustomer.CustNum = Customer.CustNum
        ttCustomer.Name    = Customer.Name
        ttCustomer.State   = Customer.State.
    END.
    rMsg = MessageHandler ("acme.myObjs.CustObj").                    /* 3 */
  END CONSTRUCTOR.

METHOD PROTECTED OVERRIDE CLASS MsgObj MessageHandler                 /* 3 */
    (INPUT iObjType AS CHARACTER):
    RETURN NEW MsgObj (iObjType).
  END METHOD.

  METHOD PUBLIC CHARACTER GetCustomerName (INPUT piRecNum AS INTEGER):
    FIND ttCustomer WHERE ttCustomer.RecNum = piRecNum NO-ERROR.      /* 15 */
    IF AVAILABLE ttCustomer THEN
      RETURN ttCustomer.Name.
    ELSE DO:
      rMsg:Alert ("Customer number" + STRING(ttCustomer.RecNum)
                  + "does not exist").
      RETURN ?.
    END.
  END METHOD.

  METHOD PUBLIC VOID CheckCredit ( ):
    IF VALID-OBJECT (rCreditObj) THEN DO:
      FOR EACH ttCustomer:
        rCreditObj:SetCurrentCustomer (ttCustomer.CustNum).
        rCreditObj:CheckCustCredit ( ).

        /* Invokes the CustCreditLimit property GET accessor */
        rMsg:InfoMsg (ttCustomer.Name + " is in good standing."
          + "  Credit Limit has been increased to "
          + STRING(rCreditObj:CustCreditLimit)).                      /* 12 */
        CATCH e AS Progress.Lang.AppError:                            /* 13 */
          IF e:ReturnValue = "Over Limit" THEN DO:
            /* Invokes the CustCreditLimit property GET accessor */
            rMsg:Alert (ttCustomer.Name + " is on Credit Hold."
              + "  Balance exceeds Credit Limit of "
```

```
              + STRING(rCreditObj:CustCreditLimit)).              /* 12 */
          END.
          ELSE
            rMsg:Alert ("Customer not found").
        END CATCH.
      END. /* FOR EACH */
    END.
    ELSE rMsg:Alert ("Unable to check credit").
  END METHOD.

/* Must implement methods defined in the IBusObj interface. Timestamp is
     a PROTECTED variable inherited from CommonObj */

  /* First version of printObj prints a single copy of a report */
  METHOD PUBLIC VOID printObj ( ):                                 /* 11 */
    OUTPUT TO PRINTER.
    DISPLAY dtTimestamp.                                           /* 5 */
    FOR EACH ttCustomer:
      DISPLAY ttCustomer.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to printer").  /* 16 */
  END METHOD.

  /* Second version of printObj takes an integer parameter representing the
     number of copies to print. */
  METHOD PUBLIC VOID printObj (INPUT piCopies AS INTEGER):         /* 11 */
    VAR INT iCnt.
    OUTPUT TO PRINTER.
    IF piCopies <> 0 THEN
    DO iCnt = 1 TO ABS(piCopies):
      DISPLAY dtTimestamp.                                         /* 5 */
      FOR EACH ttCustomer:
        DISPLAY ttCustomer.
      END.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated(STRING(piCopies)                        /* 16 */
                           + " copies of report sent to printer").
  END METHOD.

  /* Method to log customer information */
  METHOD PUBLIC VOID logObj (INPUT pcFilename AS CHARACTER):
    OUTPUT TO VALUE(pcFilename).
    DISPLAY dtTimestamp.                                           /* 5 */
    FOR EACH ttCustomer:
      DISPLAY ttCustomer.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to "           /* 16 */
                           + pcFilename + " file").
  END METHOD.

  DESTRUCTOR PUBLIC CustObj ( ):                                   /* 6 */
    EMPTY TEMP-TABLE ttCustomer.
  END DESTRUCTOR.

END CLASS.
```

This class extends `acme.myObjs.CustObj` to handle New England customers by overriding `GetCustomerName( )` to return the customer's E-mail address along with their name:

NECustObj.cls

```
USING acme.myObjs.*.

CLASS acme.myObjs.NECustObj INHERITS CustObj:                      /* 7 */

  DEFINE PRIVATE TEMP-TABLE ttEmail NO-UNDO
    FIELD RecNum AS INTEGER
    FIELD Name   AS CHARACTER FORMAT "X(20)"
    FIELD Email  AS CHARACTER FORMAT "X(20)".

  CONSTRUCTOR PUBLIC NECustObj (INPUT EmailFile AS CHARACTER):
    /* Because there are no parameters to the super class's constructor, this
       constructor call is optional */
    SUPER ( ).                                                     /* 8 */

    /* Code to initialize ttEmail. The supplied file lists the email addresses
       in the correct order for the customers being processed. */
    INPUT FROM VALUE (EmailFile).
    FOR EACH ttCustomer
      WHERE ttCustomer.State = "MA" OR                             /* 14 */
            ttCustomer.State = "VT" OR
            ttCustomer.State = "NH" OR
            ttCustomer.State = "CT" OR
            ttCustomer.State = "RI" OR
            ttCustomer.State = "ME":
      CREATE ttEmail.
      ASSIGN
        ttEmail.RecNum = ttCustomer.RecNum
        ttEmail.Name   = ttCustomer.Name.
      IMPORT ttEmail.Email.
    END.
  END CONSTRUCTOR.

  /* Override method to always get customer name and email */
  METHOD PUBLIC OVERRIDE CHARACTER GetCustomerName
    (INPUT piCustNum AS INTEGER):                                  /* 9 */

    VAR CHAR EmailName = SUPER:GetCustomerName (piCustNum).
    FIND FIRST ttEmail WHERE ttEmail.Name = EmailName NO-ERROR.    /* 15 */
    IF AVAILABLE (ttEmail) THEN
      RETURN EmailName + ";" + ttEmail.Email.
    ELSE
      RETURN EmailName.
  END METHOD.

/* First override version of printObj for a single copy */
  METHOD PUBLIC OVERRIDE VOID printObj ( ):                        /* 11 */
    OUTPUT TO PRINTER.
    DISPLAY dtTimestamp.                                           /* 5 */
    FOR EACH ttEmail:
      DISPLAY ttEmail.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to printer").  /* 16 */
  END METHOD.

  /* Second override version of printObj for multiple copies */
  METHOD PUBLIC OVERRIDE VOID printObj
    (INPUT piCopies AS INTEGER):                                   /* 11 */
    VAR INT iCnt.
    OUTPUT TO PRINTER.
    IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
      DISPLAY dtTimestamp.                                         /* 5 */
      FOR EACH ttEmail:
        DISPLAY ttEmail.
      END.
    END.
```

```
    OUTPUT CLOSE.
    PublishOutputGenerated(STRING(piCopies)                        /* 16 */
                           + " copies of report sent to printer").
  END METHOD.

  /* Override method to log customer information with email */
  METHOD PUBLIC OVERRIDE VOID logObj (INPUT pcFilename AS CHARACTER):
    OUTPUT TO VALUE (pcFilename).
    DISPLAY dtTimestamp.                                           /* 5 */
    FOR EACH ttEmail:
      DISPLAY ttEmail.
    END.
    OUTPUT CLOSE.
    PublishOutputGenerated("One copy of report sent to "           /* 16 */
                           + pcFilename + " file").
  END METHOD.

END CLASS.
```

This class provides various support methods for other classes:

HelperClass.cls

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS acme.myObjs.Common.HelperClass:

  VAR PRIVATE CLASS CustObj rCustObj.
  VAR PRIVATE CLASS MsgObj rMsg.

  DEFINE PRIVATE TEMP-TABLE ttNames NO-UNDO
    FIELD CustName AS CHARACTER.

  CONSTRUCTOR PUBLIC HelperClass ( ):
    rMsg = NEW MsgObj ("acme.myObjs.Common.HelperClass").
  END CONSTRUCTOR.

  METHOD PUBLIC VOID InitializeDate (INPUT prObject AS CLASS CommonObj):
    /* Timestamp this object */
    IF VALID-OBJECT(prObject) THEN
      prObject:updateTimestamp ( ).                                 /* 4 */
    ELSE
      rMsg:Alert ("Not a valid object").
  END METHOD.

  METHOD PUBLIC VOID ListNames (INPUT-OUTPUT prCustObj AS CLASS CustObj):
    VAR INT idx.
    DO idx = 1 to prCustObj:iNumCusts:
      CREATE ttNames.
      ttNames.CustName = prCustObj:GetCustomerName (idx).
    END.
    rCustObj = prCustObj.
  END METHOD.

  METHOD PUBLIC VOID ReportOutput (OUTPUT prInterface AS CLASS IBusObj):
    /* Send the PRIVATE CustObj instance back to be printed */
    IF VALID-OBJECT(rCustObj) THEN
      prInterface = rCustObj.
    ELSE
      rMsg:Alert ("Not a valid object").
  END METHOD.

END CLASS.
```

This class provides a method for notification of customers who have exceeded their credit limit:

CreditObj.cls

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.                                  /* 13 */

CLASS acme.myObjs.CreditObj:

  DEFINE PUBLIC PROPERTY CustCreditLimit AS DECIMAL INITIAL ? NO-UNDO /* 12 */
    /* GET: Returns the credit limit of the current Customer. If there is no
       current Customer, it returns Unknown (?).*/
    GET .
    /* SET: Raises the credit limit for Customers in good standing. Current
       increase is $1,000. */
    PROTECTED SET (INPUT piCL AS DECIMAL):
      IF Customer.Balance > piCL THEN DO:
        CustCreditLimit = Customer.Creditlimit.                      /* 13 */
        UNDO, THROW NEW Progress.Lang.AppError( "Over Limit" ).
      END.
      ELSE
        ASSIGN
          Customer.Creditlimit = piCL + 1000
          CustCreditLimit      = Customer.Creditlimit.
    END SET.

  METHOD PUBLIC VOID SetCurrentCustomer (INPUT piCustNum AS INTEGER):
    /* Verify that this object has the current Customer before the property
       is referenced. */                                             /* 15 */
    FIND FIRST Customer WHERE Customer.CustNum = piCustNum NO-ERROR.
  END METHOD.

  METHOD PUBLIC VOID CheckCustCredit ( ):
    /* invokes the property SET */
    IF AVAILABLE (Customer) THEN
      CustCreditLimit = Customer.Creditlimit.                        /* 12 */
    ELSE
    UNDO, THROW NEW Progress.Lang.AppError( "No Customer" ).         /* 13 */
  END METHOD.

END CLASS.
```

This class provides a common mechanism together with the `MessageHandler( )` method in `acme.myObjs.Common.CommonObj` for other classes to store and report error information:

MsgObj.cls

```
CLASS acme.myObjs.Common.MsgObj:

  VAR PRIVATE CHAR cObjType.

  CONSTRUCTOR PUBLIC MsgObj (INPUT pcObjType AS CHARACTER):
    cObjType = pcObjType.
  END CONSTRUCTOR.

  METHOD PUBLIC VOID Alert (INPUT ErrorString AS CHARACTER):
    MESSAGE "Error in " cObjType "!" SKIP
      ErrorString VIEW-AS ALERT-BOX ERROR.
  END METHOD.

  METHOD PUBLIC VOID InfoMsg (INPUT MsgString AS CHARACTER):
    MESSAGE MsgString VIEW-AS ALERT-BOX.
  END METHOD.

END CLASS.
```

This is the class that initializes the environment for running all the other sample classes:

Main.cls

```
USING acme.myObjs.*.
USING acme.myObjs.Common.*.
USING acme.myObjs.Interfaces.*.

CLASS Main:
  VAR PRIVATE CHAR cOutFile.
  VAR PRIVATE CLASS CommonObj rCommonObj.
  VAR PRIVATE CLASS CustObj rCustObj.
  VAR PRIVATE CLASS CustObj rCustObj2.
  VAR PRIVATE CLASS HelperClass rHelperClass.
  VAR PRIVATE CLASS IBusObj rIBusObj.

  /* First constructor instantiates a Customer object */
  CONSTRUCTOR PUBLIC Main ( ):                                    /* 10 */
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass ( )

      /* Create an instance of the CustObj class */
      rCustObj     = NEW CustObj ( )
      cOutFile      = "Customers.out".
                                                                  /* 16 */
    /* Subscribe OutputGenerated event handler for CustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_CustObjHandler).
  END CONSTRUCTOR.

  /* Second constructor takes a character parameter representing an input
     file of email addresses to instantiate a New England Customer object */
  CONSTRUCTOR PUBLIC Main (INPUT EmailFile AS CHARACTER):          /* 10 */
    ASSIGN
      /* Create an instance of the HelperClass class */
      rHelperClass = NEW HelperClass( )

      /* Create an instance of the NECustObj class */
      rCustObj     = NEW NECustObj (EmailFile)
      cOutFile      = "NECustomers.out".
                                                                  /* 16 */
    /* Subscribe OutputGenerated event handler for NECustObj */
    rCustObj:OutputGenerated:Subscribe(OutputGenerated_NECustObjHandler).
  END CONSTRUCTOR.

  /* Event handlers for each Customer class instance */
  METHOD PRIVATE VOID OutputGenerated_CustObjHandler               /* 16 */
      (pcOutputType AS CHARACTER):
    MESSAGE pcOutputType "for all customers." VIEW-AS ALERT-BOX.
  END METHOD.

  METHOD PRIVATE VOID OutputGenerated_NECustObjHandler             /* 16 */
      (pcOutputType AS CHARACTER):
    MESSAGE pcOutputType "for New England customers." VIEW-AS ALERT-BOX.
  END METHOD.

/* ObjectInfo processes information about the Customer object */
  METHOD PUBLIC VOID ObjectInfo (piInfoCount AS INTEGER):
    /* Demonstrates passing object references as parameters */

    rCommonObj = rCustObj.
                                                                  /* 10 */
    IF rCustObj:GetClass( ):TypeName = "acme.myObjs.NECustObj" THEN
      MESSAGE "Initializing reports for New England customers at"
        STRING(rCommonObj:updateTimestamp( ))                     /* 4 */
        VIEW-AS ALERT-BOX.
    ELSE
      MESSAGE "Initializing reports for all customers at"
        STRING(rCommonObj:updateTimestamp( ))                     /* 4 */
```

```
        VIEW-AS ALERT-BOX.

    /* INPUT: It is valid to pass a subclass to a method defined to take a
       super class */
    rHelperClass:InitializeDate (rCustObj).                        /* 4 */

    /* INPUT-OUTPUT: Must be an exact match, a class to a method defined to
       take that same class type */
    rHelperClass:ListNames (INPUT-OUTPUT rCustObj).
    rCustObj:CheckCredit ( ).
    rCustObj2 = rCustObj.

    /* OUTPUT: An interface is used to receive a class that implements that
       interface */
    rHelperClass:ReportOutput (OUTPUT rIBusObj).
    IF piInfoCount <> ? AND piInfoCount > 1 THEN
      rIBusObj:printObj(piInfoCount).
    ELSE
      rIBusObj:printObj( ).
    rIBusObj:logObj (cOutFile).
    rIBusObj = rCustObj.
  END METHOD.
END CLASS.
```

This is the procedure that instantiates the sample classes to run with two different sets of sample data, depending on the constructor used to instantiate `Main`:

Driver.p

```
/** This procedure drives the class example **/

VAR CLASS Main rClassExample.

/* Run the example for all Customers         */
rClassExample = NEW Main ( ).                                  /* 10 */
rClassExample:ObjectInfo (0).

/* Run the example for New England Customers */
rClassExample = NEW Main ("email.txt").                        /* 10 */
rClassExample:ObjectInfo (2).
```

Comparative procedures

The procedure-based sample application consists of several procedure files. Most of these files represent persistent procedures (procedure objects), and two of the files represent separate main-line procedures, each of which drives the application in a manner corresponding to one of the two `Main` class constructors in the class-based sample. The procedure objects also form super procedure relationships that are similar to the class hierarchies defined in the class-based sample. The following is a description of the relationships among these procedures, where ProcObject refers to a procedure object:

```
ProcObject MsgProc.p

ProcObject CreditProc.p

ProcObject CommonProc.p
  instantiates and uses MsgProc.p

  ProcObject CustProc.p
    instantiates and uses CommonProc.p as a super procedure
    instantiates and uses CreditProc.p
    uses MsgProc.p

    ProcObject NECustProc.p
      instantiates and uses CommonProc.p as a super procedure
      instantiates and uses CustProc.p as a super procedure

Procedure Main.p
  instantiates and uses CustProc.p

Procedure NEMain.p
  instantiates and uses NECustProc.p
```

The descriptions and code listings for these files follow.

This is the top-level super procedure that provides common error handler and time-tracking routines:

CommonProc.p

```
/* Define dtTimestamp as SHARED to illustrate the counterpart to inherited
   data members in classes (#5 in Table 5.1) */

DEFINE SHARED VARIABLE dtTimestamp AS DATETIME NO-UNDO.            /* 5 */
VAR HANDLE hMsg.

PROCEDURE updateTimestamp:
  DEFINE OUTPUT PARAMETER pdtTimeStamp AS DATETIME NO-UNDO.
  ASSIGN
    dtTimestamp = NOW
    pdtTimeStamp = dtTimestamp.
END PROCEDURE.

FUNCTION MessageHandler RETURNS HANDLE (INPUT ProcType AS CHARACTER).
  RUN MsgProc.p PERSISTENT SET hMsg (INPUT ProcType).
  RETURN hMsg.
END FUNCTION.

PROCEDURE CleanUp:
  IF VALID-HANDLE (hMsg) THEN
     DELETE OBJECT hMsg.
END PROCEDURE.
```

This is a procedure that extends `CommonProc.p` to provide general functionality for handling customers and is a super procedure for the `NECustProc.p` procedure, that handles New England customers:

CustProc.p

```
/* Main Block */                                                    /* 2 */
DEFINE NEW SHARED VARIABLE dtTimestamp AS DATETIME NO-UNDO.        /* 5 */
DEFINE     SHARED VARIABLE iNumCusts AS INTEGER  NO-UNDO.

VAR HANDLE hCommon.
VAR HANDLE hCreditProc.
VAR HANDLE hMsg.

DEFINE TEMP-TABLE ttCustomer NO-UNDO                               /* 14 */
  FIELD RecNum  AS INTEGER
  FIELD CustNum LIKE Customer.CustNum
  FIELD Name    LIKE Customer.Name
  FIELD State   AS CHARACTER.

RUN CommonProc.p PERSISTENT SET hCommon.                           /* 1 */
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hCommon).

RUN CreditProc.p PERSISTENT SET hCreditProc.

FUNCTION GetCreditLimit RETURNS INTEGER ( ) IN hCreditProc.
FUNCTION MessageHandler RETURNS HANDLE
  (INPUT ProcType AS CHARACTER) IN SUPER.

/* Fill temp table and get row count */
FOR EACH Customer WHERE CreditLimit > 50000:
  CREATE ttCustomer.
  ASSIGN
    iNumCusts          = iNumCusts + 1
    ttCustomer.RecNum  = iNumCusts
    ttCustomer.CustNum = Customer.CustNum
    ttCustomer.Name    = Customer.Name
    ttCustomer.State   = Customer.State.
END.
hMsg = MessageHandler(INPUT "CustProc").                           /* 3 */

PROCEDURE GetTT:                                                   /* 14 */
  DEFINE OUTPUT PARAMETER TABLE FOR ttCustomer BIND.
END PROCEDURE.

PROCEDURE GetCustomerName:
  DEFINE INPUT  PARAMETER piRecNum AS INTEGER   NO-UNDO.
  DEFINE OUTPUT PARAMETER poName   AS CHARACTER NO-UNDO.

  FIND ttCustomer WHERE ttCustomer.RecNum = piRecNum NO-ERROR.     /* 15 */
  IF AVAILABLE ttCustomer THEN
    poName = ttCustomer.Name.
  ELSE DO:
    RUN Alert IN hMsg("Customer number" + STRING(ttCustomer.RecNum)
                      + "does not exist").
    poName = ?.
  END.
END PROCEDURE.

PROCEDURE CheckCredit:
  IF VALID-HANDLE(hCreditProc) THEN DO:
    FOR EACH ttCustomer:
      RUN SetCurrentCustomer IN hCreditProc (ttCustomer.CustNum).
      RUN CheckCustCredit IN hCreditProc.

      RUN InfoMsg IN hMsg (ttCustomer.Name + " is in good standing." +
        "  Credit Limit has been increased to " +
        STRING (GetCreditLimit( )) ).                              /* 12 */
```

```
      CATCH e AS Progress.Lang.AppError:                          /* 13 */
        IF e:ReturnValue = "Over Limit" THEN
          RUN Alert IN hMsg (ttCustomer.Name + " is on Credit Hold." +
            "  Balance exceeds Credit Limit of " +
            STRING(GetCreditLimit( ))).    /* 12 */
        ELSE
          RUN Alert IN hMsg ("Customer not found").
      END CATCH.
    END. /* FOR EACH */
  END.
  ELSE
    RUN Alert IN hMsg ("Unable to check credit").
END PROCEDURE.

/* dtTimestamp is a SHARED variable defined in CommonProc.p as well */
PROCEDURE printProc:                                              /* 11 */
  DEFINE INPUT PARAMETER piCopies AS INTEGER NO-UNDO.
  VAR INT iCnt.
  OUTPUT TO PRINTER.
  IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
    DISPLAY dtTimestamp.                                          /* 5 */
    FOR EACH ttCustomer:
      DISPLAY ttCustomer.
    END.
  END.
  OUTPUT CLOSE.
  IF ABS(piCopies) > 1 THEN                                       /* 16 */
    PUBLISH "OutputGenerated" (STRING(piCopies)
                               + " copies of report sent to printer").
  ELSE
    PUBLISH "OutputGenerated" ("One copy of report sent to printer").
END PROCEDURE.

PROCEDURE logProc:
  DEFINE INPUT PARAMETER pcFilename AS CHARACTER NO-UNDO.
  OUTPUT TO VALUE(pcFilename).
  DISPLAY dtTimestamp.                                            /* 5 */
  FOR EACH ttCustomer:
    DISPLAY ttCustomer.
  END.
  OUTPUT CLOSE.
  PUBLISH "OutputGenerated" ("One copy of report sent to "        /* 16 */
                             + pcFilename + " file").
END PROCEDURE.

PROCEDURE CleanUp:
  EMPTY TEMP-TABLE ttCustomer.
  DELETE OBJECT hMsg.
  DELETE OBJECT hCreditProc.
  RUN CleanUp IN hCommon.
  DELETE OBJECT hCommon.
END PROCEDURE.
```

This procedure extends `CustProc.p` to handle New England customers by overriding the `GetCustomerName` procedure to return the customer's E-mail address along with their name:

NECustProc.p

```
DEFINE INPUT PARAMETER EmailFile AS CHARACTER NO-UNDO.

DEFINE NEW SHARED VARIABLE dtTimestamp AS DATETIME NO-UNDO.

VAR HANDLE hCustProc.
VAR HANDLE hCommonProc.
```

```
DEFINE TEMP-TABLE ttCustomer NO-UNDO REFERENCE-ONLY                /* 14 */
  FIELD RecNum  AS INTEGER
  FIELD CustNum LIKE Customer.CustNum
  FIELD Name    LIKE Customer.Name
  FIELD State   AS CHARACTER.

DEFINE TEMP-TABLE ttEmail NO-UNDO
  FIELD RecNum AS INTEGER
  FIELD Name   AS CHARACTER FORMAT "X(20)"
  FIELD Email  AS CHARACTER FORMAT "X(20)".

/* Super procedures are searched in LIFO order */
RUN CommonProc.p PERSISTENT SET hCommonProc.                       /* 8 */
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hCommonProc).                   /* 7 */
RUN CustProc.p PERSISTENT SET hCustProc.                           /* 8 */
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hCustProc).                     /* 7 */

RUN GetTT (OUTPUT TABLE ttCustomer BIND).

INPUT FROM VALUE(EmailFile).
FOR EACH ttCustomer
  WHERE ttCustomer.State = "MA" OR                                 /* 14 */
    ttCustomer.State = "VT" OR
    ttCustomer.State = "NH" OR
    ttCustomer.State = "CT" OR
    ttCustomer.State = "RI" OR
    ttCustomer.State = "ME":
  CREATE ttEmail.
  ASSIGN
    ttEmail.RecNum = ttCustomer.RecNum
    ttEmail.Name   = ttCustomer.Name.
  IMPORT ttEmail.Email.
END.
INPUT CLOSE.

PROCEDURE GetCustomerName:                                         /* 9 */
  DEFINE INPUT  PARAMETER piRecNum AS INTEGER   NO-UNDO.
  DEFINE OUTPUT PARAMETER poName   AS CHARACTER NO-UNDO.

  RUN SUPER(INPUT piRecNum, OUTPUT poName).
  FIND FIRST ttEmail WHERE ttEmail.Name = poName NO-ERROR.         /* 15 */
  IF AVAILABLE(ttEmail) THEN
    RETURN poName + ";" + ttEmail.Email.
END PROCEDURE.

/* dtTimestamp is a SHARED variable defined in CommonProc.p as well */
PROCEDURE printProc:                                               /* 11 */
  DEFINE INPUT PARAMETER piCopies AS INTEGER NO-UNDO.
  VAR INT iCnt.
  OUTPUT TO PRINTER.
  IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
    DISPLAY dtTimestamp.                                           /* 5 */
    FOR EACH ttEmail:
      DISPLAY ttEmail.
    END.
  END.
  OUTPUT CLOSE.
  IF ABS(piCopies) > 1 THEN                                        /* 16 */
    PUBLISH "OutputGenerated" (STRING(piCopies)
                               + " copies of report sent to printer").
  ELSE
    PUBLISH "OutputGenerated" ("One copy of report sent to printer").
END PROCEDURE.

PROCEDURE logProc:
  DEFINE INPUT PARAMETER pcFilename AS CHARACTER NO-UNDO.

  OUTPUT TO VALUE(pcFilename).
```

```
  DISPLAY dtTimestamp.                                              /* 5 */
  FOR EACH ttEmail:
    DISPLAY ttEmail.
  END.
  OUTPUT CLOSE.
  PUBLISH "OutputGenerated" ("One copy of report sent to "          /* 16 */
                             + pcFilename + " file").
END PROCEDURE.
```

This procedure provides a notification of customers who have exceeded their credit limit:

CreditProc.p

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.                                 /* 13 */

VAR INT TempCustNum.
VAR DECIMAL CustCreditLimit = ?.

FUNCTION GetCreditLimit RETURNS DECIMAL ( ):
  /* Returns the credit limit of the current customer. If there is no current
     customer, it returns Unknown (?). */
  RETURN CustCreditLimit.
END FUNCTION.

FUNCTION SetCreditLimit RETURNS LOGICAL PRIVATE (INPUT iCL AS DECIMAL):
  /* Raise Credit Limit for Customers in good standing */
  /* Current increase is $1,000                        */
  IF Customer.Balance > iCL THEN DO:
    CustCreditLimit = Customer.Creditlimit.
    RETURN FALSE.                                                   /* 13 */
  END.
  ELSE DO:
    ASSIGN
      Customer.Creditlimit = iCL + 1000
      CustCreditLimit      = Customer.Creditlimit.
    RETURN TRUE.
  END.
END FUNCTION.

PROCEDURE SetCurrentCustomer:
  DEFINE INPUT PARAMETER iCustNum AS INTEGER NO-UNDO.
  FIND FIRST Customer WHERE Customer.CustNum = iCustNum NO-ERROR.   /* 15 */
END PROCEDURE.

PROCEDURE CheckCustCredit:
  IF AVAILABLE (Customer) THEN DO:
    IF SetCreditLimit (Customer.Creditlimit) THEN                   /* 12 */
      RETURN "Credit Good".
    ELSE
      UNDO, THROW NEW Progress.Lang.AppError( "Over Limit" ).       /* 13 */
  END.
  ELSE
    UNDO, THROW NEW Progress.Lang.AppError( "No Customer" ).        /* 13 */
END PROCEDURE.
```

This procedure provides a common error handler for other procedures to report error information:

MsgProc.p

```
DEFINE INPUT PARAMETER ProcType AS CHARACTER NO-UNDO.

PROCEDURE Alert:
  DEFINE INPUT PARAMETER ErrorString AS CHARACTER NO-UNDO.
  MESSAGE "Error in" ProcType "!" SKIP ErrorString
    VIEW-AS ALERT-BOX ERROR.
END PROCEDURE.

PROCEDURE InfoMsg:
  DEFINE INPUT PARAMETER MsgString AS CHARACTER NO-UNDO.
  MESSAGE MsgString VIEW-AS ALERT-BOX.
END PROCEDURE.
```

This procedure is the main driver for running the comparative procedures based on the `CustProc.p` procedure object:

Main.p

```
/** This procedure drives the CustProc.p example **/              /* 10 */

/* Define iNumCusts as SHARED to illustrate the counterpart to PUBLIC data
   members in classes. */
DEFINE NEW SHARED VARIABLE iNumCusts AS INTEGER NO-UNDO.
VAR HANDLE hCustProc.
VAR INT idx.
VAR INT piCustNum.
VAR CHAR pcCustName.
VAR DATETIME pdtTimeStamp.

DEFINE TEMP-TABLE ttCustNames NO-UNDO
  FIELD CustName AS CHARACTER.
RUN CustProc.p PERSISTENT SET hCustProc.
SUBSCRIBE TO "OutputGenerated" IN hCustProc.                       /* 16 */

RUN updateTimestamp IN hCustProc (OUTPUT pdtTimeStamp).            /* 4 */
MESSAGE "Initializing reports for all customers at" STRING(pdtTimeStamp)
  VIEW-AS ALERT-BOX.

DO idx = 1 TO iNumCusts:
  CREATE ttCustNames.
  RUN GetCustomerName IN hCustProc(INPUT idx, OUTPUT pcCustName).
    ttCustNames.CustName = pcCustName.
END.

RUN CheckCredit IN hCustProc.

RUN printProc IN hCustProc.
RUN logProc IN hCustProc ("CustomersProc.out").
RUN CleanUp IN hCustProc.                                          /* 6 */

DELETE OBJECT hCustProc.

PROCEDURE OutputGenerated:                                         /* 16 */
  DEFINE INPUT PARAMETER pcOutputType AS CHARACTER NO-UNDO.
  MESSAGE pcOutputType "for all customers." VIEW-AS ALERT-BOX.
END PROCEDURE.
```

This procedure is the main driver for running the comparative procedures based on the `NECustProc.p` procedure object:

NEMain.p

```
/* This procedure drives the NECustProc.p example */              /* 10 */

/* Define iNumCusts as SHARED to illustrate the counterpart to PUBLIC data
   members in classes. */
DEFINE NEW SHARED VARIABLE iNumCusts AS INTEGER NO-UNDO.

VAR HANDLE hCustProc.
VAR INT idx.
VAR INT piCustNum.
VAR CHAR pcCustName.
VAR DATETIME pdtTimeStamp.


DEFINE TEMP-TABLE ttCustNames NO-UNDO
  FIELD CustName AS CHARACTER.

RUN NECustProc.p PERSISTENT SET hCustProc ("email.txt").
SUBSCRIBE TO "OutputGenerated" IN hCustProc.                      /* 16 */

RUN updateTimestamp IN hCustProc (OUTPUT pdtTimeStamp).           /* 4 */
MESSAGE "Initializing reports for New England customers at"
        STRING(pdtTimeStamp) VIEW-AS ALERT-BOX.

DO idx = 1 TO iNumCusts:
  CREATE ttCustNames.
  RUN GetCustomerName IN hCustProc (INPUT idx, OUTPUT pcCustName).
  ttCustNames.CustName = pcCustName.
END.

RUN CheckCredit IN hCustProc.
RUN printProc IN hCustProc (2).
RUN logProc IN hCustProc ("NECustomersProc.out").
RUN CleanUp IN hCustProc.                                         /* 6 */

DELETE OBJECT hCustProc.

PROCEDURE OutputGenerated:                                        /* 16 */
  DEFINE INPUT PARAMETER pcOutputType AS CHARACTER NO-UNDO.
  MESSAGE pcOutputType "for New England customers." VIEW-AS ALERT-BOX.
END PROCEDURE.
```

Summary comparison of classes and procedures

The following table compares the application implemented by the sample classes (see Sample classes on page 225) with the equivalent application implemented by comparative procedures (see Comparative procedures on page 234). The code numbers in the table match the commented numbers in the code for both classes and procedures.

Table 9: Comparing sample classes to similar sample procedures

Code No.	Sample classes	Comparative procedures
1	The `acme.myObjs.CustObj` class inherits from `acme.myObjs.Common.CommonObj`. (Note that `CustObj` can reference its super class as `CommonObj` because a `USING` statement specifies its package.)	`CustProc.p` defines a handle variable and runs `CommonProc.p` persistently, setting that handle. `CustProc.p` then uses `ADD-SUPER-PROCEDURE( )` to define `CommonProc.p` as its super procedure.

Code No.	Sample classes	Comparative procedures
2	All the setup for `acme.myObjs.CustObj` takes place in its constructor.	All the setup for `CustProc.p` takes place in the main block of the procedure.
3	To establish an object reference to `acme.myObjs.Common.MsgObj`, the `MessageHandler( )` method is implemented and invoked inside the `acme.myObjs.CustObj` class. The method is defined as abstract in the super class (`CommonObj`) and returns an instance of `acme.myObjs.Common.MsgObj`.	To establish a handle to `MsgProc.p`, the `MessageHandler` user-defined function, which returns a handle to `MsgProc.p` is first defined as a prototype `IN SUPER`. The function is defined in the super procedure, `CommonProc.p`. It is later invoked `CustProc.p` and returns a handle to `MsgProc.p`.
4	The `updateTimestamp( )` method is invoked on an instance of the abstract `acme.myObjs.Common.CommonObj`, where it is implemented, and also inherited by `acme.myObjs.CustObj`. This method is also called on a passed `CommonObj` reference from `InitializeDate( )`, which is implemented in the `acme.myObjs.Common.HelperClass` class and called, in turn, from `ObjectInfo( )` of the `Main` class.	The `updateTimestamp` internal procedure is called directly from `Main.p` in `CustProc.p`, and is found in its super procedure, `CommonProc.p`. The same internal procedure is called directly from `NEMain.p` in `NECustProc.p`, and is found in its super procedure, `CommonProc.p`.
5	The `updateTimestamp( )` method initializes the value of the `PROTECTED dtTimestamp` data member in `acme.myObjs.Common.CommonObj`. Because it is an inherited data member, it can be referenced in both the subclass, `acme.myObjs.CustObj`, and its subclass, `acme.myObjs.NECustObj`.	The `updateTimestamp` procedure in `CommonProc.p` initializes the value of the `SHARED` variable, `dtTimestamp`. In order for both `CustProc.p` and `NECustProc.p` to use `dtTimestamp`, they must both define it as `NEW SHARED` for reference by their own `CommonProc.p` super procedures. Note that super procedures do not provide data inheritance, requiring the use of either shared variables (as in this case) or internal procedures and user-defined functions to encapsulate and provide access outside the defining context.
6	When `acme.myObjs.CustObj` is deleted, the destructor automatically runs and cleans up program resources.	Before deleting the procedure object, the internal procedure `CleanUp` is called in both `Main.p` and `NEMain.p` to clean up resources.
7	The `acme.myObjs.NECustObj` class inherits from `acme.myObjs.CustObj`, which inherits from `acme.myObjs.Common.CommonObj`, as described in for Code No. 1. This allows the `Main` class to access all of the methods on `NECustObj` that it accesses on `CustObj`, depending on the constructor used to instantiate `Main`.	`NECustProc.p` must go through the same process to add `CommonProc.p` as a super procedure, as did `CustProc.p`, as described in Code No. 1. In addition, `NECustProc.p` must also add `CustProc.p` as a super procedure in order for `NEMain.p` to access all of the internal procedures in `NECustProc.p` that `Main.p` accesses in `CustProc.p`.

Code No.	Sample classes	Comparative procedures
8	The constructor for `acme.myObjs.NECustObj` invokes the constructor for `acme.myObjs.CustObj` using the SUPER statement. If the `CustObj` constructor took parameters, (which it does not in this case), `NECustObj` would pass them in this statement.	If `CustProc.p` took parameters (which it does not in this case), as for any calling procedure, `NECustProc.p` would then have to pass them when it ran `CustProc.p`.
9	The `acme.myObjs.NECustObj` class overrides the `GetCustomerName( )` method in `acme.myObjs.CustObj`. The override uses the SUPER system reference to invoke the super class implementation of this method.	`NECustProc.p` defines a separate version of the `GetCustomerName` internal procedure defined in the super procedure `CustProc.p`. The `NECustProc.p` version uses the `RUN SUPER` statement to invoke the super procedure implementation of this internal procedure.
10	The `Main` class defines two (overloaded) constructors, one without parameters that initializes an `acme.myObjs.CustObj` instance and one that takes, as a parameter, the name of a text file containing E-mail addresses used to initialize an `acme.myObjs.NECustObj` instance. The driver procedure, `Driver.p`, instantiates the `Main` class once for each constructor, followed by an associated call to the `Main` method, `ObjectInfo( )`, which drives the reporting for the currently instantiated customer object. Note that `ObjectInfo( )` uses reflection to identify the type name of the particular customer object currently referenced by `rCustObj`. It can then display an alert box indicating the time when it starts initializing reports for the identified customer object.	`Main.p` and `NEMain.p` each implement separate procedure mainlines. `Main.p` instantiates and drives the `CustProc.p` procedure object and `NEMain.p` instantiates and drives the `NECustProc.p` procedure object, which takes as a parameter the name of a text file containing E-mail addresses used to initialize a `NECustProc.p` object. Because each driver procedures knows the identity of the particular procedure object it is instantiating, it can display a specific alert box to indicate the initialization time of reports for that object.
11	The `acme.myObjs.CustObj` class defines two overloadings of the `printObj( )` method, one that prints a single copy of a report of data for all customers to the default printer and another that prints the number of copies specified by a parameter. `CustObj` defines these method overloadings because it implements the interface, `acme.myObjs.Interfaces.IBusObj`, which specifies them. In addition, the `acme.myObjs.NECustObj` class overrides this overloaded method to print a report of e-mail addresses for New England customers in a similar manner. The main driver method, `ObjectInfo( )`, defined in the `Main` class and called by `Driver.p`, determines which overloading of `printObj( )` to call from its own input parameter.	The `CustProc.p` procedure defines a single `PrintProc` internal procedure that prints one copy, or as many copies specified by a parameter, of the report for all customers. The `NECustProc.p` procedure defines its own version of `PrintProc` that effectively "overrides" the version in `CustProc.p` to print the mail address report for New England customers in a similar manner.The `printObj( )` method overloadings in the class samples might be more simply implemented like `PrintProc` in the procedure samples. However, the use of overloading does allow a different distribution of logic in the same decision tree while conserving the method name.

Code No.	Sample classes	Comparative procedures
12	The `acme.myObjs.CreditObj` class defines a property, `CustCreditLimit`, that is readable from outside the class hierarchy (`PUBLIC`), but writable only from the defining class or its subclasses (`PROTECTED`). In this case, the property is written by the `CheckCustCredit( )` method that is defined within the `CreditObj` class, and is read by the `CheckCredit( )` method that is defined within the unrelated `acme.myObjs.CustObj` class.	`CreditProc.p` defines a private `CustCreditLimit` variable and two user-defined functions to access it. The `GetCreditLimit` function reads the variable value and is publicly accessible. The `SetCreditLimit` function writes the variable value, but is only accessible from within `CreditProc.p` (`PRIVATE`) and is called by the internal procedure, `CheckCustCredit`. The `GetCreditLimit` function is called by the `CheckCredit` internal procedure in `CustProc.p`.
13	The `acme.myObjs.CreditObj` class provides error handling for both the `CustCreditLimit` property and the `CheckCustCredit( )` method. The `CustCreditLimit` property throws an error object initialized with an error string when it is set for a customer whose balance is over their current credit limit. The `CheckCustCredit( )` method automatically re-throws the property error object, if thrown, up to its caller, and also throws its own error object initialized with an appropriate error string for an unavailable `Customer` record. The `CheckCredit( )` method in the `acme.myObjs.CustObj` class responds to both of these error conditions raised by the method.	`CreditProc.p` provides a similar structured error handling capability for its corresponding user-defined functions and internal procedure, and the `CheckCredit` internal procedure of `CustProc.p` responds to these error conditions in a similar manner, as well.
14	The `acme.myObjs.CustObj` class defines a `PROTECTED` temp-table (`ttCustomer`) that is accessed by its subclass, `acme.myObjs.NECustObj`. Because of data inheritance, both `CustObj` and `NECustObj` directly access the same instance of `ttCustomer`.	`CustProc.p` defines a private temp-table (`ttCustomer`) that it provides to `NECustProc.p` using one of its internal procedures (`GetTT`), which passes the temp-table as an `OUTPUT` parameter. Note that the temp-table parameter is passed using the `BIND` option to a `REFERENCE-ONLY` definition of `ttCustomer` in `NECustProc.p`. This allows both `CustProc.p` and `NECustProc.p` to access the same instance of `ttCustomer`.

Code No.	Sample classes	Comparative procedures
15	Several methods in the `acme.myObjs.CustObj`, `acme.myObjs.NECustObj`, and `acme.myObjs.CreditObj` classes execute `FIND` statements with the `NO-ERROR` option. The choice is made to use traditional error handling for these statements to show its use together with structured error handling.	The `CustProc.p`, `NECustProc.p`, and `CreditProc.p` procedures rely on the same traditional error handling technique to handle their `FIND` statement results.
16	The public class event, `OutputGenerated`, is initially defined as abstract in `acme.myObjs.Common.CommonObj`, along with the protected `PublishOutputGenerated( )` method intended for publishing the event. Both are implemented by the derived class, `acme.myObjs.CustObj`. Providing `PublishOutputGenerated( )` allows not only `CustObj`, but the derived class, `acme.myObjs.NECustObj`, to publish the event in their respective versions of the `printObj( )` and `logObj( )` methods, which generate the application output for each class. The `Main` class subscribes a different handler for the `OutputGenerated` event depending on the class constructor that is running, one of which instantiates `CustObj` and the other of which instantiates `NECustObj`, so the event is handled appropriately for the class instance that publishes it.	The sample procedures use an `OutputGenerated` named event to provide similar behavior. In this case, `CustProc.p` and `NECustProc.p` each publish the event from their respective versions of the `printProc` and `logProc` internal procedures, which generate the application output.Each of the customer procedure objects, `CustProc.p` and `NECustProc.p`, is instantiated by a dedicated driver procedure, `Main.p` and `NEMain.p` (respectively), and each driver procedure subscribes its own handler for the `OutputGenerated` event published by its particular customer procedure object.

6

Develop and deploy classes

OpenEdge supports the development and deployment of classes with several features of the ABL development environment.

For details, see the following topics:

- Access class definition files in the Procedure Editor
- Access class definition files in Progress Developer Studio for OpenEdge
- Compile class definition files
- Use procedure libraries
- Use the XCODE utility

Access class definition files in the Procedure Editor

The OpenEdge Procedure Editor provides support for class definition files to create, save, and run the files. However, the editor manages class definition files differently from procedure files.

Save and open class definition files

The OpenEdge Procedure Editor **Save As** dialog box allows a file to be saved as a procedure file (`.p`), a window file (`.w`), an include file (`.i`), or a class definition file (`.cls`). This allows you to create a class or interface in the procedure editor and save it as a class definition file.

Similarly, the OpenEdge Procedure Editor **Open File** dialog box allows you to filter and open any of the file types that you have saved using the **Save As** dialog box.

Check and run a class from the Procedure Editor

The RUN functionality from the Procedure Editor (**F2** for GUI, **F1** for character mode) allows you to instantiate a class definition file from the current edit buffer, based on the file extension of the edit buffer. If the current edit buffer is an untitled edit buffer (that is, has not been explicitly saved as a `.cls` file), the Procedure Editor, by default, interprets it as a procedure file, regardless of its content. Thus, if you try to compile an untitled edit buffer that contains a CLASS or INTERFACE statement, the Procedure Editor notifies you that you have class definition file syntax and need to save the edit buffer to a file before compiling it.

If the code has been explicitly saved as a class definition file, the Procedure Editor interprets it as a class definition file, again, regardless of its content. Thus, it will compile **only** if it contains a CLASS or INTERFACE statement. That is, when running the current edit buffer, the Procedure Editor does not initially parse the source code to check for the presence of the CLASS or INTERFACE statement, but relies entirely on the filename extension with which the file has been saved to determine what syntax it needs to check for (class or procedure syntax).

In addition, depending on whether the current edit buffer has been saved to or loaded from a file or is untitled (never saved to a file), the Procedure Editor handles code a little differently when checking the syntax or running the code.

Check Syntax option

For an edit buffer loaded from or saved to a file, the **Compile Check Syntax** option of the Procedure Editor determines what syntax it **must** contain based on the filename extension. A class (`.cls`) file **must** contain CLASS or INTERFACE syntax, or the compiler generates a syntax error when checking the syntax. Any other type of file must **not** contain CLASS or INTERFACE syntax, or the compiler generates a syntax error when checking the syntax. For an already loaded or saved edit buffer, you do not have to save changes again in order to check syntax.

For an untitled edit buffer, the Procedure Editor determines if the edit buffer contains CLASS or INTERFACE syntax and checks the syntax accordingly. Once you save the untitled edit buffer to a file, the Procedure Editor checks syntax as a loaded or saved edit buffer according to the filename extension.

Run option

For an edit buffer loaded from or saved to a file, the **Compile** > **Run** option of the Procedure Editor determines what syntax it **must** contain based on the filename extension. A class (`.cls`) file **must** contain CLASS or INTERFACE syntax, or the Procedure Editor generates a syntax error when it tries to run the file. Any other type of file must **not** contain CLASS or INTERFACE syntax, or the Procedure Editor generates a syntax error when trying to run the file.

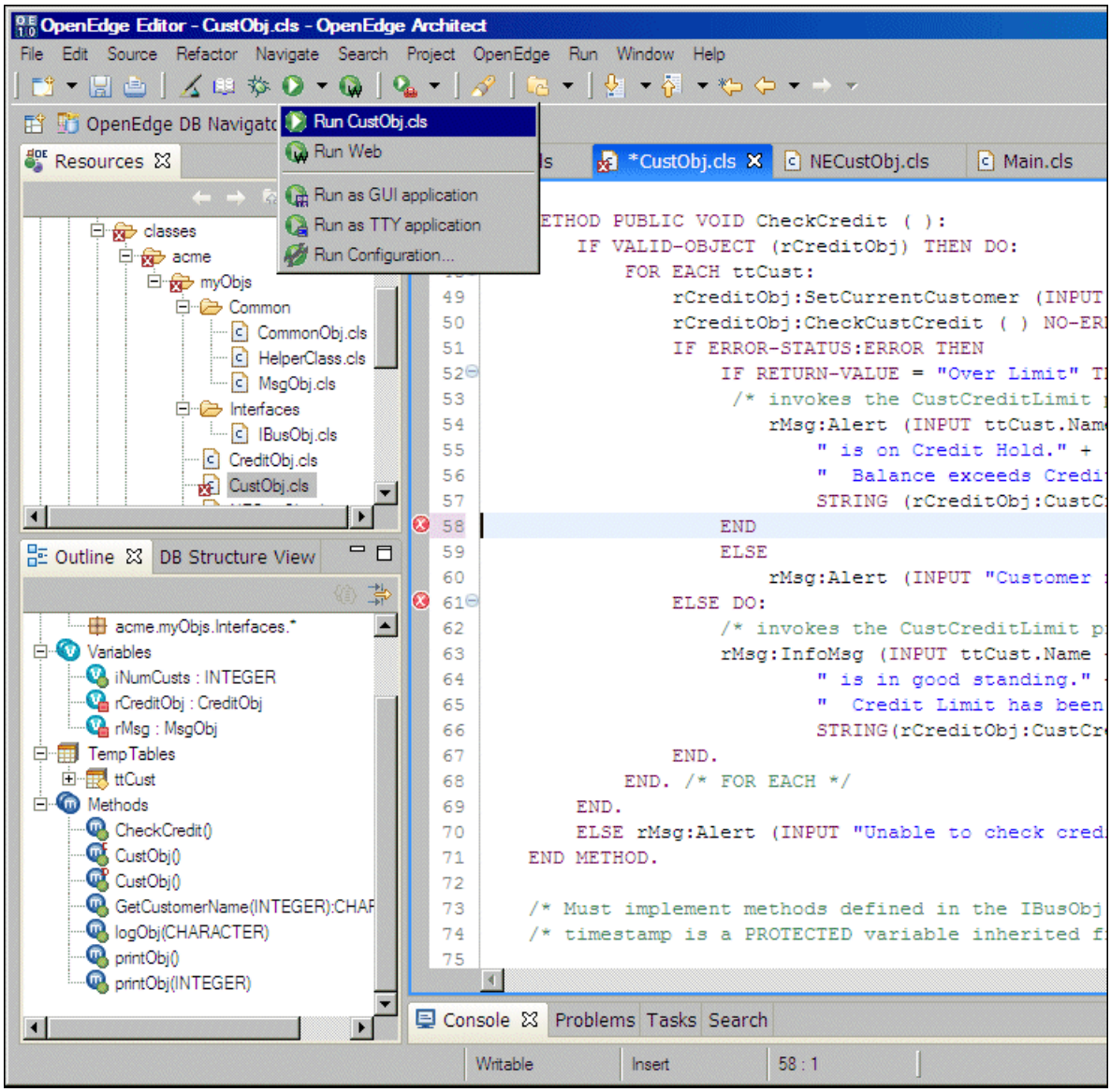
Note that for a class definition file, you must save any changes to a loaded or saved edit buffer in order for the Procedure Editor to execute them. In other words, the Procedure Editor always executes the last saved version of the class definition file in the current edit buffer.

For an untitled edit buffer, the Procedure Editor does not allow you to run the code if the edit buffer contains CLASS or INTERFACE syntax. You must first save the untitled edit buffer to a file with the `.cls` extension and to the PROPATH-relative filename specified by the class or interface type name defined in the file. For more information on class or interface type names, see Use the CLASS construct on page 54 or Use the INTERFACE construct on page 93, respectively. Again, remember to save your latest changes before trying to run them in a saved edit buffer, as the Procedure Editor always runs the code that is actually on disk.

Access class definition files in Progress Developer Studio for OpenEdge

Progress Developer Studio for OpenEdge provides support for creating new class and interface definition files (`.cls`) through wizards. Class definition files are identified by an icon (document image with small "c" inside) to differentiate class definition files from procedure and include files. This class definition file icon can be seen in the Resource View (**Resources** tab) when navigating through project directories and in the Editor when modifying files, as shown for `CustObj.cls` in the following figure.

Figure 13: Class definition file open in Progress Developer Studio for OpenEdge



The Outline View (**Outline** tab) gives a high level structural view of the current file being edited. The Outline View contains special symbols and decorators to highlight constructors, destructors, and regular methods. For example, class members that are `PRIVATE` (member symbol with red square) or `PROTECTED` (member symbol with yellow diamond) are decorated to show the different access modes.

Syntax check, compile, and run a class

When you save a class definition file in the Editor or manually initiate a syntax check, any syntax errors are highlighted in the Editor using a red circle with an "X" in it during editing, as shown for the `CheckCredit( )` method definition that is missing a terminator in an `END` statement. If you enable the **Save R-code** option for the project, the class definition file, including its entire class hierarchy is compiled and built in the target r-code directory. The exact location is relative to the project and the class (or interface) type name. You can run a class by using the **Run** toolbar icon (green circle with triangle).

For more information on class topics, such as creating a new class, Progress Developer Studio for OpenEdge online help.

Compile class definition files

An ABL class definition represents a user-defined type (object type) that encompasses all classes and interfaces in its hierarchy. When you compile a class, the compiler does not only compile the `.cls` file identified to the compiler. Instead the compiler compiles all the `.cls` files that compose its hierarchy. For each class definition file that is in the class's hierarchy, the compiler generates in-memory r-code and if the compiler `SAVE` option is specified, generates an r-code file.

Thus, the files compiled by the compiler include:

- The requested class definition file
- Class definition files for the super classes in the class hierarchy
- Class definition files for any implemented interfaces for this class
- Class definition files for any implemented interfaces for classes in the class hierarchy
- Any include files that the class and interface files use

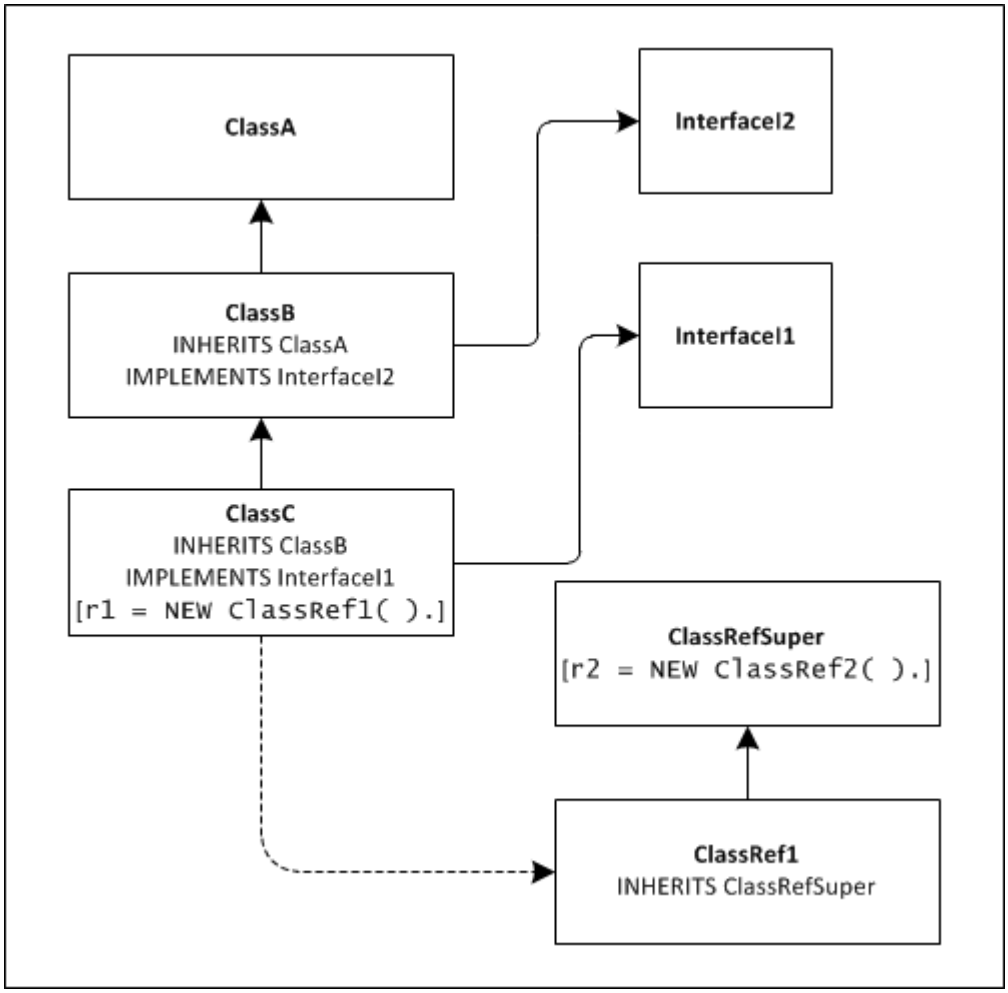
The compiler does not fully compile or generate an r-code file for other classes that are referenced within a class. The compiler instead inspects referenced class files to determine their external interfaces (their public data members, properties, methods, and events). This information is used to validate the use of these classes in the class being compiled.

Caution: When you compile a class, although ABL automatically compiles all super classes in its class hierarchy, it does not know about and cannot automatically compile any of the subclasses of the class you are compiling. Thus, when you change the API of a super class, you must compile the class and all subclasses of the class to ensure that all objects with this class in their hierarchy inherit the updated definition. ABL, by itself, has no knowledge of these subclasses, so you must keep track of them manually or by using configuration management tools.

Protocol for class hierarchy and references

In the following figure, **ClassC** inherits from both **ClassB** and **ClassA**. **ClassC** also implements **InterfaceI1**, and **ClassB** implements **InterfaceI2**.

Figure 14: Compiling class definition files



During the compilation of **ClassC**, the compiler generates separate r-code files for **ClassC**, **ClassB** and **ClassA**, as well as for **InterfaceI1** and **InterfaceI2**. Also, if InterfaceI1 or InterfaceI2 inherit additional interfaces, the compiler generates separate r-code files for these inherited interfaces. Because **ClassRef1** is referenced by **ClassC** (dotted arrow), the compilation of **ClassC** causes the compiler to determine the public interface for **ClassRef1**, but does not cause the compiler to build r-code for **ClassRef1** or **ClassRefSuper**. Also, the compiler does not check the public interfaces to **ClassRef2** (not shown) that is referenced from **ClassRefSuper**.

If you modify a method signature as well as executable code in both **ClassA** and **ClassRefSuper**, and then compile **ClassC**, the compiler recompiles **ClassC**, **ClassB** and **ClassA** and retrieves the updated signature from **ClassRefSuper**, but does not recompile **ClassRefSuper**. In order to give the application access to the updated executable code (including the updated signature) in **ClassRefSuper**, you need to recompile **ClassRef1** or **ClassRefSuper** before recompiling and running **ClassC** or any other code that instantiates **ClassRef1**.

Match data types

When calling a method, the compiler checks that the data types of passed parameters are consistent with the method definition. The data type passed by the caller must either exactly match the data type of the parameter definition in the called method, or it must have an appropriate widening relationship with data type of the parameter definition. This strong data type matching is exactly the same as when you invoke a user-defined function. The compiler does not employ the flexible conversion rules used by the `RUN` statement to call procedures, which attempt to convert an argument of almost any data type that is passed to a parameter defined as any other. For more information, see the Parameter passing syntax reference entry in *ABL Reference*.

Use the COMPILER system handle

The `COMPILER` system handle provides several attributes and methods that are helpful for either compiling class definition files in particular or for writing ABL tools that manage the compilation of ABL source files in general. These include attributes and methods to:

- Optimize compiler performance in certain situations involving multiple class definition files in a single compilation.
- Identify the object type name of the most recently compiled file, if it defines a class or interface.
- Identify the number of compilation messages returned from the most recent `COMPILE` statement, and to identify each message and its location in the source file where it occurred. This supports compilation messages generated from multiple ABL source files in a single compilation, which can happen when compiling procedures, but which always happens when compiling several class definition files that are part of a class hierarchy.

The following sections describe the attributes and methods that support these compilation features:

- MULTI-COMPILE attribute on page 250
- CLASS-TYPE attribute on page 251
- NUM-MESSAGES attribute on page 251
- GET-MESSAGE(n) method on page 251
- GET-NUMBER(n) method on page 251
- GET-FILE-NAME(n) method on page 251
- GET-ERROR-ROW(n) method on page 252
- GET-ERROR-COLUMN(n) method on page 252
- GET-FILE-OFFSET(n) method on page 252

MULTI-COMPILE attribute

The `MULTI-COMPILE` attribute on the `COMPILER` system handle is a read-write attribute that improves the performance of the compiler in certain situations. Normally, the compiler compiles all class definition files in a class hierarchy. The compiler does not try to determine if the files in the class hierarchy have changed since the last compilation.

To improve performance, you can set the MULTI-COMPILE attribute to TRUE. This causes the compiler to only compile those class definition files within a class hierarchy that it has not already compiled since MULTI-COMPILE was set to TRUE. Your build application can set this attribute to TRUE when you are building all class definition files within an application, and no files are going to be modified during the process. While this attribute is TRUE, the ABL session caches class and interface compilations such that when a class or interface is thereafter compiled, ABL will not recompile any classes or interfaces that are in the cache. ABL flushes this cache when MULTI-COMPILE is set to FALSE.

The following example demonstrates the use of the MULTI-COMPILE attribute:

```
COMPILER:MULTI-COMPILE = TRUE.
COMPILE "C:\acme\myObjs\CustObj.cls".
COMPILE "C:\acme\myObjs\NECustObj.cls".
COMPILER:MULTI-COMPILE = FALSE.
```

In this example, any class definition files that have been compiled during the compilation of acme.myObjs.CustObj.cls and also participate in the class hierarchy of acme.myObjs.NECustObj.cls, are not compiled again during the compilation of acme.myObjs.NECustObj.cls.

Caution: Leaving this attribute set to TRUE causes the ABL session to ignore any changes to classes that have already been compiled during the session while the attribute was set.

CLASS-TYPE attribute

The CLASS-TYPE attribute on the COMPILER system handle is a read-only attribute that identifies the class or interface type name of the class definition file that was compiled by the most recently executed COMPILE statement. For example, after the application compiles a class definition file, CustObj.cls, that contains the CLASS statement, CLASS acme.myObjs.CustObj, the COMPILER:CLASS-TYPE attribute contains the string, "acme.myObjs.CustObj". If the last file compiled with the COMPILE statement is not a class definition file, this attribute is set to the empty string ("").

NUM-MESSAGES attribute

The NUM-MESSAGES attribute on the COMPILER system handle is a read-only attribute that returns an INTEGER specifying the number of messages generated by the most recently executed COMPILE statement.

GET-MESSAGE(n) method

The GET-MESSAGE( ) method on the COMPILER system handle returns a character string specifying the text of the *n*th message generated by the most recently executed COMPILE statement.

GET-NUMBER(n) method

The GET-NUMBER( ) method on the COMPILER system handle returns an INTEGER specifying the ABL error message number for the *n*th message generated by the most recently executed COMPILE statement.

GET-FILE-NAME(n) method

The GET-FILE-NAME( ) method on the COMPILER system handle returns a character string specifying the filename of the source code file that caused the *n*th message generated by the most recently executed COMPILE statement.

GET-ERROR-ROW(n) method

The GET-ERROR-ROW( ) method on the COMPILER system handle returns an INTEGER specifying the line number of the source code file that caused the *n*th message generated by the most recently executed COMPILE statement.

GET-ERROR-COLUMN(n) method

The GET-ERROR-COLUMN( ) method on the COMPILER system handle returns an INTEGER specifying the character column in the line of the source code file that caused the *n*th message generated by the most recently executed COMPILE statement.

GET-FILE-OFFSET(n) method

The GET-FILE-OFFSET( ) method on the COMPILER system handle returns an INTEGER specifying the 1-based character offset in the source code file that caused the *n*th message generated by the most recently executed COMPILE statement.

Use procedure libraries

You can store and retrieve the r-code for class files using procedure libraries.

The definition statements (such as VAR, DEFINE VARIABLE) and NEW function search the PROPATH for the specified class r-code file. If the statements encounter a procedure library on the PROPATH, ABL searches these libraries for the specified r-code.

The RUN statement supports the ability to execute an r-code file stored in a procedure library that is not on the PROPATH using the following syntax:

```
RUN procedure-library-path<<member-name>>
```

There is **no** support of this capability for classes. When you define an object reference or create a class instance, the r-code for that object must be found on the PROPATH.

Use the XCODE utility

The XCODE utility for encrypting source code is supported with class definition files. This means that similar to procedure files, class definition files can be encrypted. For more information on the XCODE utility, see *Manage ABL Applications*.

A

Overloaded method and constructor call scenarios

To disambiguate calls to overloaded methods and constructors, ABL handles scenarios that go beyond basic differences in the number, mode, and data types of parameters in order to determine a match. For example, some parameter data types, such as temp-table and object types, can involve complex matching scenarios that are not resolved until run time, or can require resolving the best of several matching scenarios at compile time.

For details, see the following topics:

- Parameter lists that differ only by mode
- Parameter data types that differ only by extent
- Parameters match widened data types
- Match dynamic and static temp-table or ProDataSet parameters
- Object reference parameters search the class hierarchy or the interface
- Match the Unknown value to parameters
- Match values of Unknown data types to parameters

Parameter lists that differ only by mode

If parameter lists differ only by the mode, you must specify the parameter modes when invoking the method or constructor. Failure to find a match raises a compile-time error.

Note: Best practice is that you always specify the mode for all parameters of a class-based method call.

Parameter data types that differ only by extent

Parameter data types differing by extent (arrays) typically match strictly according to scalar type and size extent. However, if the method is called using a parameter with a fixed size, and no method exists that matches a corresponding parameter with that fixed size, the call can be resolved by a method definition whose corresponding parameter has an indeterminate array with no size. In any case, failure to find a match raises a compile-error.

The examples that follow show how different calls to overloaded methods (or constructors) with parameters of the same data type, but with different sizes, are resolved:

```
CLASS Extents:
/* 1 */ METHOD PUBLIC VOID setVal (INPUT piIn AS INTEGER):
        END METHOD.

/* 2 */ METHOD PUBLIC VOID setVal (INPUT piIn AS INTEGER EXTENT 10):
        END METHOD.

/* 3 */ METHOD PUBLIC VOID setVal (INPUT piIn AS INTEGER EXTENT):
        END METHOD.

END CLASS.
```

When the following procedure calls the `setVal( )` method with a given parameter, the overloaded method definition that it matches corresponds to the bold-faced number referenced in the comments:

```
var class[] rExt.

var int i00.
var int[10] i10.
var int[8] i08.
var int[] ixx.

rExt = NEW Extents( ).

rExt:setVal(42 ). /* Calls 1 - constant integer                    */
rExt:setVal(i00). /* Calls 1 - integer variable                    */
rExt:setVal(i10). /* Calls 2 - fixed array variable of extent 10   */
rExt:setVal(i08). /* Calls 3 - no fixed array variable of extent 8:
                               matches an indeterminate extent     */
rExt:setVal(ixx). /* Calls 3 - indeterminate extent                */
```

Parameters match widened data types

ABL always searches for an exact parameter match, but if none is found, it will accept the closest parameter match according to widening in the available method or constructor overloads at compile time. Failure to find a match raises a compile-time error. For more information on matching parameter data types with widening, see the Parameter passing syntax reference entry in *ABL Reference*.

The examples that follow show how widening can resolve calls to overloaded methods (or constructors):

```
CLASS Widening:
/* 1 */ METHOD PUBLIC VOID setVal (INPUT piVal AS INTEGER):
        END METHOD.

/* 2 */ METHOD PUBLIC VOID setVal (INPUT pdVal AS DECIMAL):
        END METHOD.
END CLASS.
```

When the following procedure calls the `setVal( )` method with a given parameter, the overloaded method definition that it matches corresponds to the bold-faced number referenced in the comments:

```
VAR CLASS Widening rWid.
VAR INT iVal  = 42.
VAR INT64 i64Val = 42.
VAR DECIMAL dVal = 4.2.

rWid = NEW Widening( ).

rWid:setVal(iVal).   /* Calls 1 - matches exactly                 */
rWid:setVal(i64Val). /* Calls 2 - INT64 matches the wider DECIMAL */
rWid:setVal(dVal).   /* Calls 2 - matches exactly                 */
```

Match dynamic and static temp-table or ProDataSet parameters

If methods or constructors are overloaded with one dynamic and several static data object (temp-table or ProDataSet) parameters, a passed dynamic data object matches any corresponding static data object parameter whose schema definition is identical to the schema of the dynamic data object passed at run time. If no static data object overloading matches the passed dynamic schema exactly, the AVM invokes the method or constructor that is overloaded with the corresponding dynamic data object parameter. For a passed dynamic temp-table or ProDataSet parameter, failure to find a match raises a run-time error. A passed static data object similarly matches any identical static data object parameter, and otherwise matches the corresponding dynamic data object parameter. For a passed static temp-table or ProDataSet parameter, failure to find a match raises a compile-time error.

The following examples show how matching combinations of static and dynamic data object parameters resolves calls to overloaded methods (or constructors). These examples use temp-table parameters, but would work the same way using equivalent ProDataSet parameters.

The example that follows shows a class that defines a method, `setTable( )`, which is overloaded only by different types of static temp-table (`TABLE`) parameters; the `setTable( )` method is then called by other method definitions that illustrate a variety of parameter matching scenarios:

```
CLASS DataObjectOverloads:
  DEFINE TEMP-TABLE ttCustomer LIKE Customer NO-UNDO.
  DEFINE TEMP-TABLE ttOrder    LIKE Order    NO-UNDO.

  METHOD PUBLIC VOID setTable(INPUT TABLE ttCustomer):   /* first */
  END METHOD.

  METHOD PUBLIC VOID setTable(INPUT TABLE ttOrder):  /* second */
  END METHOD.

  METHOD PUBLIC VOID Scenario1( ):
    CREATE ttCustomer.
    setTable(INPUT TABLE ttCustomer).
  END METHOD.

  METHOD PUBLIC VOID Scenario2( ):
    VAR HANDLE hTT.
    CREATE TEMP-TABLE hTT.
    hTT:CREATE-LIKE("Customer").
    hTT:TEMP-TABLE-PREPARE("Customer").
    setTable(INPUT TABLE-HANDLE hTT).
  END METHOD.

  METHOD PUBLIC VOID Scenario3( ):
    VAR HANDLE hTT.
    CREATE TEMP-TABLE hTT.
    setTable(INPUT TABLE-HANDLE hTT).
  END METHOD.
END CLASS.
```

With method `Scenario1( )`, the invocation of method `setTable( )` invokes the first version of `setTable( )` taking an `INPUT` temp-table with a `Customer` table schema. In this scenario, the caller passes a static temp-table and the called method has a matching static temp-table parameter definition.

With method `Scenario2( )`, the invocation of method `setTable( )` invokes the first version of `setTable( )` taking an `INPUT` temp-table with a `Customer` table schema. In this scenario, the caller has a dynamic temp-table, which at run time is populated with a schema that matches the `Customer` table. At compile time, no determination can be made concerning what version of `setTable( )` to invoke; the decision is delayed until run time.

With method `Scenario3( )`, the AVM cannot identify a method to run because the compiler and AVM cannot determine what version of `setTable( )` taking an `INPUT` temp-table parameter is most appropriate. The handle that is passed as a parameter does not have a schema associated with it. Therefore, the AVM is unable to match the invocation to any one of the overloaded methods. The AVM raises a run-time error identifying this ambiguity.

This example shows a class that defines a method, `setTable( )`, which is overloaded by different combinations of parameter modes and types of temp-table parameters, where one is a dynamic temp-table (`TABLE-HANDLE`) parameter. As in the previous example, the method is called by other method definitions that illustrate a variety of parameter matching scenarios:

```
CLASS DataObjectOverloads:
  DEFINE TEMP-TABLE ttCustomer LIKE Customer NO-UNDO.
  DEFINE TEMP-TABLE ttOrder LIKE Order       NO-UNDO.

  METHOD PUBLIC VOID setTable(INPUT TABLE ttCustomer):    /* first */
  END METHOD.

  METHOD PUBLIC VOID setTable(OUTPUT TABLE ttOrder):      /* second */
  END METHOD.

  METHOD PUBLIC VOID setTable(INPUT TABLE-HANDLE ttHndl): /* third */
  END METHOD.

  METHOD PUBLIC VOID Scenario1( ):
    CREATE ttOrder.
    setTable(OUTPUT TABLE ttOrder).
  END METHOD.

  METHOD PUBLIC VOID Scenario2( ):
    VAR HANDLE hTT.
    CREATE TEMP-TABLE hTT.
    hTT:CREATE-LIKE("Customer").
    hTT:TEMP-TABLE-PREPARE("Customer").
    setTable(TABLE-HANDLE hTT).
  END METHOD.

  METHOD PUBLIC VOID Scenario3( ):
    VAR HANDLE hTT.
    CREATE TEMP-TABLE hTT.
    hTT:CREATE-LIKE("Order").
    hTT:TEMP-TABLE-PREPARE("Order").
    setTable(TABLE-HANDLE hTT).
  END METHOD.

  METHOD PUBLIC VOID Scenario4( ):
    VAR HANDLE hTT.
    CREATE TEMP-TABLE hTT.
    setTable(TABLE-HANDLE hTT).
  END METHOD.

  METHOD PUBLIC VOID Scenario5( ):
    VAR HANDLE hTT.
    CREATE TEMP-TABLE hTT.
    setTable(OUTPUT TABLE-HANDLE hTT).
  END METHOD.
END CLASS.
```

With method `Scenario1( )`, the invocation of method `setTable( )` invokes the second version of `setTable( )` taking an `OUTPUT` temp-table with an `Order` table schema. In this scenario, the caller passes a static temp-table and the called method has a matching static temp-table parameter definition.

With method `Scenario2( )`, the invocation of method `setTable( )` invokes the first version of `setTable( )` taking an `INPUT` temp-table with a `Customer` table schema. In this scenario, the caller has a dynamic temp-table, which at run time is populated with a schema that matches the `Customer` table. At compile time, no determination can be made concerning what version of `setTable( )` to invoke; the decision is delayed until run time. At run time, because a match is made against a method version defined with a static temp-table parameter, this method is chosen over the more general method taking the `TABLE-HANDLE` parameter.

With methods `Scenario3( )` and `Scenario4( )`, the invocation of method `setTable( )` invokes the third version of `setTable( )` taking an `INPUT TABLE-HANDLE`. In this scenario, the caller has a dynamic temp-table, which is being passed with a mode that does not match the method that would otherwise qualify (the second version), or which at run time is populated with a schema that does not match a version of the method defined with a static temp-table parameter. At compile time, no determination can be made concerning which version of `setTable( )` to invoke; the decision is delayed until run time. At run time, because no match is made against a static temp-table parameter, the `TABLE-HANDLE` version of the method is chosen.

With method `Scenario5( )`, the invocation of method `setTable( )` invokes the second version of `setTable( )`taking an `OUTPUT` temp-table with an `Order` table schema. In this scenario, the caller has a dynamic temp-table, which at run time has no schema associated with it. The determination is made at compile time based upon the fact there is only one version of `setTable( )` that has an `OUTPUT` temp-table parameter. At run time, during the invocation of the method, the table is populated with the `Order` table schema.

Object reference parameters search the class hierarchy or the interface

ABL searches first for an exact object type match between the passed object reference and the corresponding parameter definition (class type to class type or interface type to interface type). If no exact object type match is found, it searches for a method or constructor whose corresponding parameter definition is for a class type that is a super class in the same class hierarchy as the passed object reference. If there is more than one such super class type represented for that parameter, ABL chooses the method or constructor whose corresponding parameter matches the most derived (closest) of those super classes to the class type of the passed object reference. Failure to find a match raises a compile-time error.

However, there are situations where passing object reference parameters can cause ABL to raise a compile-time ambiguity error, where the corresponding parameters of overloaded methods or constructors define multiple super class and interface type parameter combinations that the class type of the passed object reference parameter inherits and implements, respectively. In such cases, there is no way for ABL to tell what combination of parameters represent the best match for the passed object reference parameters, because more than one combination of parameter definitions can satisfy the combination of passed object reference parameters equally well. In general, if the corresponding parameters of overloaded methods differ by an interface, ABL only accepts an exact object type match to the passed object reference parameter.

For more information on defining and passing object reference parameters, see Define an object reference parameter on page 158 and Pass object reference parameters on page 159.

The following examples define method (or constructor) overloadings that take different combinations of class type and interface type parameters and describes how they respond to invocations that pass object references as a variety of related object types.

This example shows method calls to overloaded methods distinguished by object reference parameters whose class types are related to the following single class hierarchy:

```
ClassA                          <--| Super Classes
  ClassB INHERITS ClassA        <--|
    ClassC INHERITS ClassB      <--|
      ClassD INHERITS ClassC    <-- Most Derived Class
```

The following class defines three overloadings of the `setClass( )` method that take class type parameters from this hierarchy:

```
CLASS MonoClass:
  METHOD PUBLIC VOID setClass (INPUT prObj AS ClassA):
    MESSAGE "setClass in ClassA".
  END METHOD.

  METHOD PUBLIC VOID setClass (INPUT prObj AS ClassB):
    MESSAGE "setClass in ClassB".
  END METHOD.

  METHOD PUBLIC VOID setClass (INPUT prObj AS ClassD):
    MESSAGE "setClass in ClassD".
  END METHOD.
END CLASS.
```

The following procedure invokes different overloads of this `setClass( )` method:

```
VAR ClassA rClassA = NEW ClassA( ).
VAR ClassB rClassB = NEW ClassB( ).
VAR ClassC rClassC = NEW ClassC( ).
VAR ClassD rClassD = NEW ClassD( ).
VAR MonoClass rOne = NEW MonoClass( ).

rOne:setClass( rClassA ). /* 1 */
rOne:setClass( rClassB ). /* 2 */
rOne:setClass( rClassC ). /* 3 */
rOne:setClass( rClassD ). /* 4 */
```

Execution of the procedure results in the following messages displayed with numbers corresponding to the bold-faced code comments:

1. **"setClass in ClassA"** — Matches the ClassA parameter
2. **"setClass in ClassB"** — Matches the ClassB parameter
3. **"setClass in ClassB"** — Matches the ClassB parameter, as there is no ClassC version
4. **"setClass in ClassD"** — Matches the ClassD parameter

The examples that follow show method calls to overloaded methods distinguished by object reference parameters whose class types represent super classes and most derived classes from two different class hierarchies that share the same super classes:

```
ClassZ                        <--| Super Classes of ClassX and ClassW
  ClassY INHERITS ClassZ      <--|
    ClassX INHERITS ClassY    <-- Most Derived Class of ClassX Hierarchy
    ClassW INHERITS ClassY    <-- Most Derived Class of ClassW Hierarchy
```

The following class defines three overloadings of the `setClasses( )` method, each of which take two class type parameters from this hierarchy:

```
CLASS BiClass:
  METHOD PUBLIC VOID setClasses (INPUT prObjY AS ClassY,
                                 INPUT prObjX AS ClassX):
    MESSAGE "First".
  END METHOD.

  METHOD PUBLIC VOID setClasses (INPUT prObjX AS ClassX,
                                 INPUT prObjY AS ClassY):
    MESSAGE "Second".
  END METHOD.

  METHOD PUBLIC VOID setClasses (INPUT prObjY1 AS ClassY,
                                 INPUT prObjY2 AS ClassY):
    MESSAGE "Third".
  END METHOD.
END CLASS.
```

The following procedure invokes different overloads of this `setClasses( )` method:

```
VAR ClassW rObjW = NEW ClassW( ).
VAR ClassX rObjX = NEW ClassX( ).
VAR BiClass rTwo = NEW BiClass( ).

rTwo:setClasses(rObjW, rObjW). /* 1 */
rTwo:setClasses(rObjX, rObjX). /* 2 */
rTwo:setClasses(rObjW, rObjX). /* 3 */
rTwo:setClasses(rObjX, rObjW). /* 4 */
```

The two calls to these `setClasses( )` methods resolve as follows, according to the bold-faced code comments:

1. Matches third version of `setClasses( )`. This is the only possible match.
2. Matches the first and second versions of `setClasses( )`, because each version has an exact match in one of its two parameters (first or second). ABL cannot choose one over the other, resulting in a compile-time ambiguity error.
3. Matches the first version of `setClasses( )`. The call matches both the first and third versions, but the first version is a better match because the second parameter is an exact match.
4. Matches the second version of `setClasses( )`. The call matches both the second and third versions, but the second version is a better match because the first parameter is an exact match.

The following class defines four overloadings of the `setClasses( )` method, each of which takes three class type parameters from the same hierarchy:

```
CLASS TriClass:
  METHOD PUBLIC VOID setClasses (INPUT prObjX AS ClassX,
                                 INPUT prObjY AS ClassY,
                                 INPUT prObjZ AS ClassZ):
    MESSAGE "First".
  END METHOD.

  METHOD PUBLIC VOID setClasses (INPUT prObjX AS ClassX,
                                 INPUT prObjZ AS ClassZ,
                                 INPUT prObjY AS ClassY):
    MESSAGE "Second".
  END METHOD.

  METHOD PUBLIC VOID setClasses (INPUT prObjX  AS ClassX,
                                 INPUT prObjY1 AS ClassY,
                                 INPUT prObjY2 AS ClassY):
    MESSAGE "Third".
  END METHOD.

  METHOD PUBLIC VOID setClasses (INPUT prObjX  AS ClassX,
                                 INPUT prObjZ1 AS ClassZ,
                                 INPUT prObjZ2 AS ClassZ):
    MESSAGE "Fourth".
  END METHOD.
END CLASS.
```

The following procedure invokes one overload of this `setClasses( )` method:

```
VAR TriClass rThree = NEW TriClass( ).
VAR ClassX rObjX  = NEW ClassX( ).

rThree:setClasses(rObjX, rObjX, rObjX).
```

The call to `setClasses( )` matches all four method definitions. However, the third version is the best match because all three of its parameter matches, together, represent the closest match among them.

As noted, previously, using interface types in parameter definitions can easily create ambiguities in calls to overloaded methods. The following examples show calls to overloaded methods whose object reference parameters are distinguished by one or more interface types:

```
ClassA                                          <-- Super Class
  ClassB INHERITS ClassA IMPLEMENTS InterfaceC  <-- Most Derived Class

ClassF IMPLEMENTS InterfaceC and InterfaceD     <-- Most Derived Class
```

This example shows calls within a class definition to two overloaded `setObj( )` methods distinguished by a class type parameter that implements an interface and an interface type parameter that represents the same interface:

```
CLASS InterClass:
  METHOD PROTECTED VOID setObj (INPUT prObjB AS ClassB):
    MESSAGE "First".
  END METHOD.

  METHOD PROTECTED VOID setObj (INPUT prObjC AS InterfaceC):
    MESSAGE "Second".
  END METHOD.

  METHOD PUBLIC INTEGER test ( ):
    VAR ClassB rObjB = NEW ClassB( ).
    VAR InterfaceC rObjC = NEW ClassF( ).

    setObj(rObjB). /* 1 */
    setObj(rObjC). /* 2 */
  END METHOD.
END CLASS.
```

The two calls to these `setObj( )` methods resolve as follows, according to the bold-faced code comments:

1. Matches the first method, because the class types of the passed and defined parameters are an exact match
2. Matches the second method, because the interface types of the passed and defined parameters are an exact match

This example shows calls within a class definition to two overloaded `setObj( )` methods distinguished by a class type parameter that is the super class of a class type that implements an interface and an interface type parameter that represents the same interface:

```
CLASS InterClass:
  METHOD PROTECTED VOID setObj (INPUT prObjA AS ClassA):
    MESSAGE "First".
  END METHOD.

  METHOD PROTECTED VOID setObj (INPUT prObjC AS InterfaceC):
    MESSAGE "Second".
  END METHOD.

  METHOD PUBLIC INTEGER test ( ):
    VAR ClassB rObjB = NEW ClassB( ).
    VAR ClassF rObjF = NEW ClassF( ).

    setObj(rObjB). /* 1 */
    setObj(rObjF). /* 2 */
  END METHOD.
END CLASS.
```

The two calls to these `setObj( )` methods resolve as follows, according to the bold-faced code comments:

1. The passed parameter represents a class type that inherits from the class type of the parameter defined for the first method and implements the interface type of the parameter defined for the second method. ABL cannot choose the best match and raises a compile-time ambiguity error.
2. The passed parameter represents a class type that is unrelated to the class type of the parameter defined for the first method (except that they both implement the same interface type), but because it does implement

the interface type of the parameter defined for the second method, ABL matches the call to the second method.

This example shows calls within a class definition to two overloaded `setObj( )` methods distinguished by parameters defined with two different interface types, where both interfaces are implemented by one class type and one interface is implemented by an unrelated class type:

```
CLASS InterClass:
  METHOD PROTECTED VOID setObj (INPUT prObjC AS InterfaceC):
    MESSAGE "First".
  END METHOD.

  METHOD PROTECTED VOID setObj (INPUT prObjD AS InterfaceD):
    MESSAGE "Second".
  END METHOD.

  METHOD PUBLIC INTEGER test ( ):
    VAR ClassB rObjB = NEW ClassB( ).
    VAR ClassF rObjF = NEW ClassF( ).

    setObj(rObjB). /* 1 */
    setObj(rObjF). /* 2 */
  END METHOD.
END CLASS.
```

The two calls to these `setObj( )` methods resolve as follows, according to the bold-faced code comments:

1. The passed parameter represents a class type that implements the interface type of the parameter defined for the first method, but not the second. So, ABL matches the call to the first method.
2. The passed parameter represents a class type that implements the interface types of the parameters defined for both methods. ABL cannot choose the best match and raises a compile-time ambiguity error.

Match the Unknown value to parameters

When passing the `Unknown value (?)` for a given parameter, ABL only selects a method or constructor to invoke if the `Unknown value (?)` causes no compile-time ambiguity among overloadings. If there are multiple methods or constructors with corresponding parameters being passed the `Unknown value (?)`, ABL raises a compile-time ambiguity error. You can avoid ambiguity with the `Unknown value (?)` by converting it to a known data type using a data type conversion function or by assigning it to a variable, data member, or property, then passing the converted value, variable, data member, or property as the parameter.

The following example shows one overloaded method call using the Unknown value (?) without ambiguity and another overloaded method call using the Unknown value (?) with ambiguity:

```
CLASS Unknown:
  METHOD PRIVATE INTEGER setUnknowns (INPUT pcName AS CHARACTER,
                                      INPUT piPos  AS INTEGER):
    RETURN 1.
  END METHOD.

  METHOD PRIVATE INTEGER setUnknowns (INPUT pcName  AS CHARACTER,
                                      INPUT pcLocal AS CHARACTER):
    RETURN 2.
  END METHOD.

  METHOD PUBLIC VOID test ():
    VAR INT iReturn.

    iReturn = setUnknowns(?, 12).     /* 1 */
    iReturn = setUnknowns("Frank", ?). /* 2 */
  END METHOD.
END CLASS.
```

The two calls to these methods resolve as follows, according to the bold-faced code comments:

1. The first passed parameter with the Unknown value (?) matches the first parameter of either method. However, because the passed parameter value of 12 for the second parameter can match only the second INTEGER parameter of the first method version, ABL matches the call to this first version.
2. The second passed parameter with the Unknown value (?) matches the second parameter of either method. Because the passed parameter value of "Frank" for the first parameter can also match the first parameter of either method, ABL cannot choose a match and raises a compile-time ambiguity error.

However, the following procedure calls the same methods in the previous Unknown class without ambiguity, because it ensures that every Unknown value (?) parameter is converted to a specific data type:

```
VAR INT iUnknown.
VAR Unknown rUObj = NEW Unknown( ).

rUObj:setUnknowns("Bill", STRING(?)). /* 1 */

iUnknown = ?.
rUObj:setUnknowns("Steve", iUnknown). /* 2 */
```

The two calls to these methods resolve as follows, according to the bold-faced code comments:

1. Passes an Unknown value (?) converted to the CHARACTER data type for the second parameter using the STRING built-in function. ABL therefore matches the second version of the setUnknowns( ) method, whose second parameter is defined as a CHARACTER
2. Passes an Unknown value (?) converted to the INTEGER data type by first assigning the Unknown value (?) to an INTEGER variable, then passing this variable as the second parameter value. ABL therefore matches the first version of the setUnknowns( ) method, whose second parameter is defined as an INTEGER.

Match values of Unknown data types to parameters

When passing an expression of unknown data type (such as the BUFFER-VALUE attribute), the AVM only selects a method or constructor to invoke if the unknown data type causes no run-time ambiguity among overloadings. If the data type actually passed at run time does not allow the AVM to select a method or constructor, the AVM raises a run-time error. Similar to the `Unknown value (?)`, you can avoid potential run-time errors with an expression of unknown data type by converting it to a known data type using a data type conversion function or by assigning it to a variable, data member, or property, then passing the converted value, variable, data member, or property as the parameter.

Index

A

Abstract
  class events
    defining syntax *78*
  classes
    defining syntax *54*
    overview *53*
  methods
    defining syntax *72*
  properties
    defining syntax *63*
ABSTRACT option
  CLASS statement *54*
  DEFINE EVENTS statement *78*
  DEFINE PROPERTY statement *63*
  METHOD statement *72*
Access mode
  Class events *52*
  data members and properties *50*
  defined *15*
  methods *51*
Accessing
  data members *153*
  data members and properties
    examples *154*
    overview *39*
  properties *153*
  static members *186*
Accessors
  defining *63*
  GET
    defining *63*
  SET
    defining *63*
Application errors in instance constructors *203*
Assigning object references *177*
Associated block, error handling *197*

B

Built-in system and object references *174*

C

Call stack with classes *224*
Calling class-based methods
  dynamically *146*
  inside a class *142*
  instance
    outside a class *144*
Calling class-based methods *(continued)*
  overloaded *150*
Calling methods
  overview *38*
CAST function
  overview *40*
  using *180*
Casting object references *179*
Catch blocks
  definition *197*
CATCH statement
  overview *40*
Catching error objects
  definition *197*
Checking class syntax
  Procedure Editor *246*
  Progress Developer Studio for OpenEdge *248*
Class definition files (.cls)
  accessing in Procedure Editor *245*
  accessing in Progress Developer Studio for OpenEdge *247*
  compared to procedure source files *48*
  compiling *248*
  object type names *45*
  overview *44*
  structure *44*
  XCODE utility *252*
Class events
  access modes *52*
  compared to named events
    constructs *82*
    overview *34*, *40*
  defined *12*
  defining
    abstract syntax *78*
    access mode *78*
    examples *78*
    name *78*
    overrides *78*
    overview *33*
    signature *78*
    static syntax *78*
    syntax *78*
  overriding in class hierarchies *108*
  publishing
    examples *170*
    overview *39*
    syntax *170*
  scoping in class hierarchies *106*
  subscribing handlers
    examples *168*

Class events *(continued)*
subscribing handlers *(continued)*
overview *39*
syntax *168*
using *167*
Class hierarchies
ABL for constructing *116*
compared to procedure hierarchies *104*
constructing
process of *102*
data scoping *106*
defined *27*
described *17*
event scoping *106*
invoking super class methods *119*
method and constructor overloading *113*
method scoping *105*
overriding data *107*
overriding events *108*
overriding methods *107*
Class names
See also Object type names
defining *45*
See also Object type names
CLASS statement
overview *30*
syntax *54*
terminating *54*
Class type names *45*
See also Object type names
Class types *10*
See also Classes
Class-based objects
comparing *185*
compatibility
deprecated ABL *219*
procedure objects *218*
rules *218*
Class-based programming
ABL overview *30*
compared with procedure-based *41*
defined *12*
CLASS-TYPE attribute *251*
Classes
advantages *10*
compared to persistent procedures *10*
comparing types, interfaces, and objects *14*
compiling *248*
container *20*
creating
overview *37*
defined *27*
defining
abstract overview *53*
abstract syntax *54*
constructors *84*
Classes *(continued)*
defining *(continued)*
data members *58*
default widget pool *54*
destructors *88*
events *77*
examples *54*
interfaces to implement *54*
methods *71*
non-inheritable *54*
overview *30*
properties *62*
super class *54*
syntax *54*
type names *54*
defining in Procedure Editor *245*
defining in Progress Developer Studio for OpenEdge *247*
delegate *20*
described *10*
destroying instances
overview *37*
documented samples *225*
error handling models *197*
event handlers
callbacks *222*
event handling
handle-based objects *221*
foundations *11*
handling error conditions
destructors *205*
instance constructors *203*
methods *199*
properties *201*
static constructors *205*
instantiating
NEW function example *137*
NEW statement example *137*
New( ) method examples *139*
members *12*
preprocessor features *196*
procedure constructs compared *225*
procedure libraries *252*
process of defining
based on other classes *53*
behavior *51*
data *50*
programming conventions *42*
referencing the call stack *224*
reflection *205*
streams in *194*
strong typing of *103*
structured error handling *197*
super class and subclass *17*
supporting ABL elements *40*
traditional error handling *197*

Classes *(continued)*
types of methods *51*
using delegation *130*
using polymorphism *125*
widget pools *223*
widgets in *194*
work tables in *194*
Clear( ) method *211*
Clone( ) method *92*
CommonObj.cls *225*
Comparing
class and procedure constructs *225*
class-based objects *185*
handles and object references *220*
Compatibility of classes and procedures *218*
COMPILER system handle *250*
Compiling classes
class hierarchy and references *249*
COMPILER system handle *250*
data type matching *250*
Procedure Editor *246*
Progress Developer Studio for OpenEdge *248*
Compiling subclasses *248*
Complex data members *58*
Constructor overloading *25*
See also Method overloading
CONSTRUCTOR statement
overview *34*
syntax *85*
Constructors
class instantiation *116*
defining
access mode *85*
data and behavior *85*
examples *85*
name *85*
overloaded *113*
overloaded examples *113*
overview *34*
signature *85*
static features *192*
static syntax *85*
syntax *85*
handling error conditions
instantiation *203*
handling instance error conditions
application logic *203*
handling static error conditions *205*
invoking
examples in the super class *117*
in the super class *116*
overloaded *150*
overloaded examples *150*
overloaded
in class hierarchies *113*
invoking *150*
Constructors *(continued)*
overloaded *(continued)*
invoking from another constructor *117*
Container
class defined *20*
class described *130*
CreditObj.cls *225*
CustObj.cls *225*

D

Data members
access modes *50*
accessing
overview *39*
complex *58*
defined *12*
defining
access mode *58*
examples *58*
name *58*
overview *32*
static features *190*
static syntax *58*
syntax *58*
overriding in class hierarchies *107*
scoping in class hierarchies *106*
Data types
See also Object types
defining for properties *63*
See also Object types
DEFINE EVENT statement
overview *33*
syntax *78*
DEFINE PROPERTY statement
overview *33*
DEFINE statement
data members *58*
parameter object references *158*
properties *63*
property object references *134*
variable object references *134*
Defining
abstract classes
overview *53*
syntax *54*
abstract members
class events *78*
methods *72*
properties *63*
behavior in classes *51*
class constructors *84*
class destructors *88*
class events
overview *33*

Defining *(continued)*
classes
based on other classes *53*
overview *30*
syntax *54*
constructors
overview *34*
data in classes *50*
data members
overview *32*
destructors
overview *35*
interfaces
overview *35*
methods
overview *31*
object references
parameters *158*
return types *163*
temp-table fields *166*
variables and properties *134*
properties
overview *33*
static members
class events *78*
data members *58*
methods *72*
properties *63*
Delegate
class defined *20*
class described *130*
Delegation
defined *27*
described *20*
using with classes *130*
DELETE OBJECT statement
syntax *98*
Derived classes
defined *27*
subclasses compared *17*
Deserialization
Deserializing an instance from file *215*
DESTRUCTOR statement
overview *35*
syntax *88*
Destructors
access mode *88*
defining
data and behavior *88*
examples *88*
name *88*
overview *35*
syntax *88*
handling error conditions *205*
Driver.p *225*
DYNAMIC-CAST function
overview *40*
using *184*
DYNAMIC-INVOKE function
overview *38*, *146*
syntax *146*
DYNAMIC-NEW statement
overview *37*, *136*
syntax *137*

E

Encapsulation
defined *27*
described *15*
Enumerations *43*
Equals( ) method *92*
Error conditions in classes
instance constructor logic *203*
instantiation *203*
Error handling
associated block *197*
complete documentation on *197*
destructors *205*
instance constructors *203*
methods *199*
models available *197*
properties *201*
static constructors *205*
structured model *197*
traditional model *197*
using both models together *197*
Error objects
ABL built-in and user-defined *197*
definition *197*
overview *40*
Event handling
callbacks *222*
class events *167*
handle-based objects *221*
Event methods
overview *33*
syntax and usage *167*
Event modifiers *78*
Events
See also Class events
defining
prototypes for interfaces *94*
named compared to class
constructs *82*
overview *34*, *40*
See also Class events

F

FINAL option
  classes *54*
  methods *72*
FIRST-OBJECT attribute
  overview *40*
FUNCTION statement
  returning object references *163*
Functions
  CAST *180*
  DYNAMIC-CAST *184*
  DYNAMIC-INVOKE *146*
  NEW *136–137*
  PROGRAM-NAM *224*
  TYPE-OF *174*
  VALID-OBJECT *172*

G

Garbage collection *98*
GET accessors *63*
  See also Accessors
GET-ERROR-COLUMN( ) method *252*
GET-ERROR-ROW( ) method *252*
GET-FILE-NAME( ) method *251*
GET-FILE-OFFSET( ) method *252*
GET-MESSAGE( ) method *251*
GET-NUMBER( ) method *251*
GetClass( ) method
  Progress.Lang.Class class *207*
  Progress.Lang.Object *92*
  usage *205*
GetConstructor( ) method *207*
GetConstructors( ) method *207*
GetEvent( ) method *208*
GetEvents( ) method *208*
GetInterfaces( ) method *208*
GetMethod( ) method *208*
GetMethods( ) method *208*
GetProperties( ) method *208*
GetProperty( ) method *208*
GetVariable( ) method *208*
GetVariables( ) method *209*
Glossary of terms *27*

H

Handles
  compared to object references *220*
HasStatics( ) method *209*
HelperClass.cls *225*

I

IBusObj.cls *225*
IMPLEMENTS option *54*
Information hiding *15*
  See also Encapsulation
Inheritance
See also Class hierarchies
  defined *28*
  described *17*
  root class *17*
  single *17*
  See also Class hierarchies
INHERITS option *54*
Instantiation errors *203*
Interface names
See also Object type names
  defining *45*
  See also Object type names
INTERFACE statement
  overview *35*
  syntax *94*
Interface type names. *45*
  See also Object type names
Interface types *11*
  See also Interfaces
Interfaces
  comparing types, classes, and objects *14*
  defined *28*
  defining
    event prototypes *94*
    examples *96*
    method prototypes *94*
    overview *35*
    property prototypes *94*
    syntax *94*
    type name *94*
  described *11*
  process of defining *93*
  referencing and using *96*
Invalid ABL in classes *219*
Invoke( ) method
  example
    instance with parameters *146*
    instance without parameters *146*
    static with parameters *146*
    static without parameters *146*
  overview *146*
  syntax
    methods with parameters *146*
    methods without parameters *146*
    static methods with parameters *146*
    static methods without parameters *146*
  using *146*
Invoking methods
  dynamically *146*

Invoking methods *(continued)*
inside a class *142*
instance
outside a class *144*
overloaded *150*
overview *38*
Invoking overloaded constructors *117*, *150*
Invoking super class constructors *116*
IS-CLASS attribute *220*
IsAbstract( ) method *209*
IsFinal( ) method *209*

L

LAST-OBJECT attribute
overview *40*

M

Main.cls *225*
Managing
object life-cycle *98*
Member
defined *28*
Members
class *12*
static *186*
Messages
defined *28*
Method modifiers *72*
Method overloading
defined *29*
described *25*
Method overriding
defined *29*
METHOD statement
overview *31*
returning object references *163*
syntax *72*
Methods
access modes *51*
class event *167*
defined *12*
defining
abstract syntax *72*
access mode *72*
data and behavior *72*
examples *72*
name *72*
non-overrideable *72*
overloaded *113*
overloaded examples *113*
overrides *72*
overview *31*
prototypes for interfaces *94*
return types *72*
Methods *(continued)*
defining *(continued)*
signature *72*
static features *190*
static syntax *72*
syntax *72*
without return types *72*
handling error conditions *199*
instance
examples calling outside a class *144*
invoking from outside a class *144*
invoking
dynamically *146*
examples inside a class *142*
from inside a class *142*
overloaded *150*
overloaded examples *150*
overview *38*
static inside defining class *189*
static overridden *189*
invoking in super class *119*
invoking polymorphically
example *125*
overriding process *125*
overloading in class hierarchies *113*
overriding in class hierarchies
examples *108*
Progress.Lang.Object *91*
returning object references *163*
scoping in class hierarchies *105*
types of in classes *51*
MsgObj.cls *225*
MULTI-COMPILE attribute *250*

N

Named compared to class events
constructs *82*
overview *34*, *40*
NECustObj.cls *225*
NEW function
example *137*
overview *136*
using *137*
NEW function and statement
overview *37*
syntax *137*
NEW statement
example *137*
overview *136*
using *137*
New( ) method
examples *139*
overview *136*
syntax *139*
NEXT-SIBLING property *92*

NON-SERIALIZABLE
data members 58
NON-SERIALIZABLE option
DEFINE PROPERTY statement 63
NUM-MESSAGES attribute 251
NumParameters property 211

O

Object references
accessing data members 154
assigning 177
built-in defined 174
casting
assignments 182
parameters 182
to invoke methods 183
compared to handles 220
defined 29
defining
parameters 158
return types 163
temp-table fields 166
variable examples 134
variables and properties 134
FIRST-OBJECT and LAST-OBJECT attributes 177
invoking methods 144
passing as parameters
INPUT 159
INPUT-OUTPUT 159
OUTPUT 159
verifying
type 174
validity 172
Object type names
defining 45
defining for classes 54
defining for interfaces 94
examples 46
overview 36, 44
referencing
overview 37
unqualified 48
relative to PROPATH 47
USING statement 48
Object types
See also Classes, Interfaces, Types
naming 45
using 36
See also Classes, Interfaces, Types
Object-oriented programming
ABL general support 9
ABL support for classes
advantages 10
foundations 11
delegation 20
Object-oriented programming *(continued)*
encapsulation 15
inheritance 17
Method overloading 25
overview 14
polymorphism 21
strong typing 26
terms 27
Object( ) constructor 92
Objects
class-based
defining 43
comparing class-based 185
comparing types, classes, and interfaces 14
constructing 116
creating 136
defined
general 10
managing 98
passing between a client and server 159
visual in classes 194
OERA 10
ON statement 222
OpenEdge Reference Architecture 10
Overloaded methods and constructors
calling parameter scenarios
data types differing by extent 254
differing by mode 253
matching object references 258
matching temp-tables and ProDataSets 255
matching the Unknown value (?) 263
matching values of unknown type 265
matching widened data types 254
defining and invoking 113
OVERRIDE option
DEFINE EVENTS statement 78
DEFINE PROPERTY statement 63
METHOD statement 72
Overriding
See also Method overriding
data 107
events 108
methods 107
See also Method overriding

P

Package property 207
PACKAGE-PRIVATE
class events 78
constructors 85
data members 58
methods 72
properties 63
PACKAGE-PROTECTED
class events 78

PACKAGE-PROTECTED *(continued)*
constructors *85*
data members *58*
methods *72*
properties *63*
Packages
defining *45*
USING statement *48*
Parameters
overloading scenarios *253*
passing object references *159*
Persistent procedures
compared to classes *10*
Polymorphism
defined *29*
described *21*
example *125*
overriding process *125*
using with classes *125*
Preprocessor features in classes *196*
PREV-SIBLING property *92*
PRIVATE
class events *78*
constructors *85*
data members *58*
methods *72*
properties *63*
Procedure Editor *245*
Procedure libraries and classes *252*
Procedure-based programming
defined *12*
ProDataSets
interface/class compatibility *93*
PROGRAM-NAME function *224*
Programming conventions *42*
Programming models
class-based *12*
compared *13*
procedure-based *12*
Progress Developer Studio for OpenEdge *247*
Progress.Lang.AppError *197*
Progress.Lang.Class class
properties and methods *206*
using *205*
Progress.Lang.Object
described *91*
properties and methods *91*
temp-table fields *166*
Progress.Lang.ParameterList class
properties and methods *210*
using *209*
Progress.Lang.ProError *197*
Progress.Reflect classes *211*
Properties
access modes *50*
Properties *(continued)*
accessing
examples *154*
overview *39*
defined *12*
defining
abstract syntax *63*
access mode *63*
accessors *63*
data type *63*
example *63*
name *63*
overrides *63*
overview *33*
prototypes for interfaces *94*
static features *190*
static syntax *63*
syntax *63*
GET accessor
defining *63*
handling error conditions *201*
overriding in class hierarchies *107*
Progress.Lang.Object *91*
scoping in class hierarchies *106*
SET accessor
defining *63*
PROTECTED
class events *78*
constructors *85*
data members *58*
methods *72*
properties *63*
PUBLIC
class events *78*
constructors *85*
data members *58*
methods *72*
properties *63*
Publish( ) event method
example *170*
overview *39*
Publishing class events
examples *170*
overview *39*
syntax *170*

R

R-code files
verifying source code type *220*
Reflection
Progress.Lang.Class class *205*
Progress.Lang.ParameterList class *209*
run-time class instantiation *139*
RETURN statement
overview *40*

Return types
method *72*
Root class
defined *29*
described *17*, *91*
properties and methods *91*
ROUTINE-LEVEL ON ERROR UNDO, THROW statement
overview *40*
Running classes
Procedure Editor *246*
Progress Developer Studio for OpenEdge *248*

S

Sample classes
acme.myObjs.Common.CommonObj *225*
acme.myObjs.Common.HelperClass *225*
acme.myObjs.Common.MsgObj *225*
acme.myObjs.CreditObj *225*
acme.myObjs.CustObj *225*
acme.myObjs.Interfaces.IBusObj *225*
acme.myObjs.NECustObj *225*
class hierarchies *225*
comparative procedures *234*
driver procedure *225*
Main *225*
procedure summary comparisons *240*
Sample comparative procedures
class summary comparisons *240*
CommonProc.p *234*
CreditProc.p *234*
CustProc.p *234*
Main.p *234*
MsgProc.p *234*
NECustProc.p *234*
NEMain.p *234*
procedure hierarchy *234*
SERIALIZABLE
data members *58*
SERIALIZABLE option
DEFINE PROPERTY statement *63*
Serialization
Serializing an instance to file *214*
SESSION system handle
object references *177*
SET accessors *63*
See also Accessors
SET-CALLBACK( ) built-in method *222*
SetParameter( ) method *211*
Single inheritance *17*
Singletons and static members *192*
Statements
CATCH *197*
CLASS
terminating *54*
Statements *(continued)*
CONSTRUCTOR *85*
DEFINE
data members *58*
parameter object references *158*
properties *63*
property object references *134*
variable object references *134*
DEFINE EVENT *78*
DELETE OBJECT *98*
DESTRUCTOR *88*
DYNAMIC-NEW *136–137*
FUNCTION
returning object references *163*
INTERFACE *94*
METHOD
returning object references *163*
NEW *136–137*
ON *222*
ROUTINE-LEVEL ON ERROR UNDO, THROW *40*
SUPER *116*
THIS-OBJECT *117*
Static
class events
defining syntax *78*
constructor
defining behavior *192*
defining syntax *85*
data members
defining syntax *58*
members
accessing *186*
accessing inside defining class *189*
accessing outside defining class *188*
creating a singleton *192*
defining *190*
defining behavior *190*
initializing and deleting *192*
options for referencing *187*
scoping *186*
using *186*
members compared to handle objects *58*
methods
calling inside defining class *189*
calling overloaded *190*
calling overridden *189*
defining overriding and overloading *192*
defining syntax *72*
properties
defining syntax *63*
type-name syntax
class named as reserved keyword *194*
qualified table field name conflicts *194*
variable and class names the same *194*
STATIC option
DEFINE EVENT statement *78*

STATIC option *(continued)*
DEFINE PROPERTY statement *63*
METHOD statement *72*
Streams in classes *194*
Strong typing
classes and *103*
described *26*
Structured error handling model
compatibility with traditional *197*
definition *197*
overview *197*
Subclasses
compiling *248*
methods invoked in a super class *119*
super classes compared *17*
Subscribe( ) event method
examples *168*
overview *39*
Subscribing class event handlers
examples *168*
overview *39*
syntax *168*
Super classes
defined *29*
methods invoked from a subclass *119*
subclasses compared *17*
SUPER statement
overview *40*
syntax *116*
SUPER system reference
overview *40*
syntax *119*
usage *176*
SuperClass property *207*
System references
built-in defined *174*
SUPER
usage *176*
THIS-OBJECT *175*

T

Temp-tables
interface/class compatibility *93*
THIS-OBJECT statement
overview *40*
syntax *117*
THIS-OBJECT system reference
overview *40*
THROW *40*
See also UNDO, THROW
Throwing error objects
definition *197*
ToString( ) method *92*
Traditional error handling model
compatibility with structured *197*
Traditional error handling model *(continued)*
definition *197*
Type names *36*
See also Object type names
TYPE-OF function
overview *40*
TypeName property *207*
Types
comparing classes, interfaces, and objects *14*
defined *30*
described *10*
getting information on *205*

U

UNDO, THROW option
functionality *197*
overview *40*
Unqualified
class names
overview *37*
interface names
overview *37*
object type names
referencing *48*
Unsubscribe( ) event method
overview *39*
Unsubscribing class event handlers
overview *39*
syntax *168*
USE-WIDGET-POOL option *54*
User-defined data types *44*
User-defined functions
returning object references *163*
User-defined types
See also Classes, Data types, Interfaces
defining
class types *54*
interface types *93*
naming *45*
See also Classes, Data types, Interfaces
Using
handles and object references *220*
object types *36*
static members *186*
USING statement
overview *37*

V

VALID-OBJECT function
overview *40*
Verifying object references *172*
VOID option *72*

W

Widget pools in classes *223*
Widgets in classes *194*
Work tables in classes *194*

X

XCODE utility and class files *252*