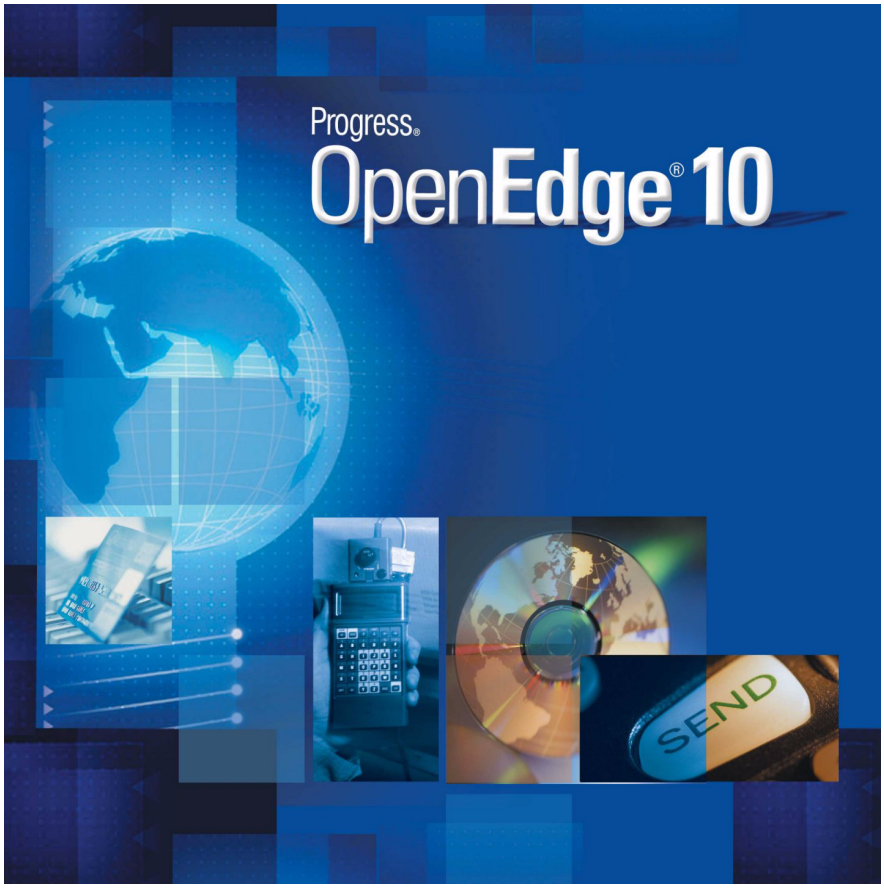

OpenEdge Getting Started: Object-oriented Programming

Third Party Product Notices for OpenEdge

December 2006

Product Code: 4496; R10.1B

Contents

Figures

Tables

Procedures

Preface

This Preface contains the following sections:

Purpose

ABL has long supported the ability to program with objects built from persistent procedures (procedure objects). These objects feature and depend almost entirely on run-time management to organize them for use in an application. With OpenEdge® Release 10.1, ABL includes support for classes. Classes allow an ABL developer to program with objects built from user-defined classes (class-based objects) that can be defined and organized for use by an application at compile time. The developer can define and manage class-based objects using the standard features of object-oriented programming available in programming languages, such as Java. The developer can also use procedure-based and class-based objects together in a single application.

This manual introduces object-oriented programming using classes in ABL and also describes how to work with class-based and procedure objects together.

Audience

The ABL developer who is thoroughly familiar with programming in ABL. For more information, see *OpenEdge Development: ABL Handbook*.

It is also helpful, but not required, for the reader to be familiar with object-oriented programming concepts and have familiarity with another object-oriented programming language, such as Java.

Organization

Chapter 1, "Object-oriented Programming and ABL"

Introduces support for classes in ABL, compares and contrasts programming with procedures and programming with classes, and provides an overview of object-oriented programming for the ABL programmer.

Chapter 2, "Getting Started with Classes, Interfaces, and Objects"

Describes the syntax and provides examples of defining classes and interfaces, and describes basic object life-cycle management.

Chapter 3, "Designing Objects—Inheritance, Polymorphism, and Delegation"

Describes how to use inheritance, polymorphism, and delegation to design and organize class-based objects.

Chapter 4, "Programming with Class-based Objects"

Describes how to instantiate and program with class-based objects.

Chapter 5, "Programming with Class-based and Procedure Objects"

Describes how to use class-based and procedure objects in an ABL application.

Chapter 6, "Developing and Deploying Classes"

Describes how to use ABL development tools and ABL features to create, compile, and deploy class files.

Chapter A, "Overloaded Method and Constructor Calling Scenarios"

Describes several different parameter matching scenarios for calls to overloaded methods and constructors and how ABL handles them.

Using this manual

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

References to ABL compiler and run-time features

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

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

References to ABL data types

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

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

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

Typographical conventions

This manual uses the following typographical conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax statements, code examples, system output, and filenames.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax statements.
`Fixed-width bold`	Fixed-width bold indicates variables with special emphasis.
`UPPERCASE fixed width`	Uppercase words are ABL keywords. Although these are always shown in uppercase, you can type them in either uppercase or lowercase in a procedure.
	This icon (three arrows) introduces a multi-step procedure.
	This icon (one arrow) introduces a single-step procedure.
Period (.) or colon (:)	All statements except `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` end with a period. `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.

Convention	Description
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
...	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

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

Syntax

```
ACCUM aggregate expression
```

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

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

Long syntax descriptions split across lines

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

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

Syntax

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

Complex syntax descriptions with both required and optional elements

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

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

Syntax

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

Example procedures

This manual provides numerous example classes and procedures that illustrate syntax and concepts. You can access the example files and details for installing the examples from the following locations:

- The Documentation and Samples CD that you received with your product.
- The OpenEdge Documentation page on PSDN:

```
http://www.psdn.com/library/kbcategory.jspa?categoryID=129
```

To compile and run these sample classes and procedures:

1. Install the samples from either location.
2. In the installation directory for the documentation samples locate the samples for this book in the following directory path:

```
\src\prodoc\getstartoop
```

   The directory contains two subdirectories, `classes` and `procedures`. The `classes` subdirecory is the root of a directory tree that contains the sample class definition and related files. You must work with the sample class definition files in their locations within this directory tree. The `procedures` subdirectory contains a set of sample procedure source files that provide procedure-based code that is virtually equivalent to the class-based code found in the sample classes.

3. Before compiling and running these files in your OpenEdge development environment, add these `classes` and `procedures` subdirectories to your PROPATH.

OpenEdge messages

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

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

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

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

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

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

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

Obtaining more information about OpenEdge messages

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

- Choose **Help→Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help→Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

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

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

```
OpenEdge-install-dir/bin/pro
```

2. Press **F3** to access the menu bar, then choose **Help→Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File→Exit**.

1

Object-oriented Programming and ABL

Object-oriented programming is a popular programming model within the software development industry. There are standard concepts that all object-oriented programming languages support, such as abstraction, encapsulation, inheritance, and strong typing. In releases prior to OpenEdge® 10.1A, ABL provides limited support for these concepts in the language using persistent procedures. The current release of OpenEdge provides language extensions to more completely 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 can seamlessly coexist 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.

The following sections provide an overview of this support:

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 the OpenEdge Reference Architecture (OERA) introduced with OpenEdge Release 10. For more information on the OERA, see *OpenEdge Getting Started: OpenEdge Reference Architecture*.

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.

In releases of OpenEdge prior to Release 10.1A, ABL persistent procedures allow you to create and manage objects in which most of the relationships between them are created and managed at run time. ABL classes, on the other hand, contain relationships that you create at compile time, 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* 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 technologies 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 in these previous releases.

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 classes in ABL

This ABL support for object-oriented programming 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 testing.
- 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.

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 user-defined data types in ABL. 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. For more information on inheritance, see the "Inheritance" section on page 1–10.

Classes can also implement one or more interfaces, each of which is defined by an `INTERFACE` statement. An *interface* declares a common public mechanism to access behavior that one or more classes can define and that these classes do not inherit from a common class. Interfaces allow you to more easily define and manage common behavior that might be implemented differently in different objects and for different purposes. An interface also represents a user-defined data type, but never contains an implementation of that type. Only a class can implement the type specified by an interface.

Thus, a class 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.

A single CLASS or INTERFACE statement identifies a source code file as representing 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

As described in the previous section, ABL supports two basic types of user-defined objects—objects based on procedures and objects based on classes. Although you use much of the same syntax to program each type of object, each of these object types supports an entirely different programming model:

- Procedure-based
- Class-based

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, which includes various types of data and internal procedures and user-defined functions that provide the 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 includes various types of data (data members and properties), but instead of internal procedures and user-defined functions, a class-based object's behavior is provided by methods. 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 *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. As such, each method, data member, and property defined within a class is a *member* 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 are almost entirely user-defined and associated with the user-defined 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 most such relationships at compile time by syntax designed to associate class-based objects with each other in a well-defined hierarchy. As a result, you can organize class-based objects using interfaces that are well defined before you even compile the objects.

If you change an 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.

Comparing programming 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 and extended with procedures, 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, now that they are available. 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 executing 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 manual, there are frequent comparisons between class-based programming and procedure-based programming. To simplify the documentation, 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. Also, this manual contains a number of sections and subsections with the common title, “Comparison with procedure-based programming.” These sections and subsections appear where use of a specific class-based programming feature is compared with the use of procedure-based programming to provide 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*, and *object* are closely related but it is important to understand the difference between these three terms. A type defines the interface to the state and behavior of a class, regardless of how it is implemented. A class provides a definition—primarily data members, properties, and methods—and an implementation for the methods. (As described previously, an interface can also represent a type that declares methods that a class implements.) An object is an instance of a class. So, a class defines a set of objects that share a common structure and provide a common interface that specifies how other classes (sets of objects) can interact with it.

The object-oriented extensions to 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 names of data elements serve as sufficient denotations of type.

The following sections describe these basic object-oriented concepts:

- Abstraction
- Encapsulation
- Inheritance
- Delegation
- Polymorphism
- Method overloading
- Strong typing
- Glossary of terms

Abstraction

An *abstraction* defines the contract for a class to the consumers of the class, which states the set of functionality that the class agrees to provide in a certain way. This contract represents general functionality that can be specified in greater detail. To support the abstraction it represents, a class defines a public interface that other objects can use to communicate with it. (A *public* interface specifies the means by which any class can access another class's functionality.) The importance of abstraction is that the compiler is able to enforce the contract that it defines between different classes.

ABL supports interface types (described previously) to help a class enforce the public interface that it provides as a contract to its class consumers.

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 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*, then, 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.

Note: A message is not necessarily a physical data packet, but is an abstraction that conveys the idea that one object is using some means of requesting an action to be performed on the part of a second object that 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 methods and properties 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, protected, or public) that controls where and how the member can be accessed. Thus, a private method can be accessed only within the class where it is defined. A protected method can be accessed within the class where it is defined and within any class that inherits from that class (see the "Inheritance" section on page 1–10). 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 private, protected, and public data members and private, protected, and public properties of a class, and you can further define properties so that they can be read and written according to different access modes.

Figure 1–1 shows a simple example of encapsulation.

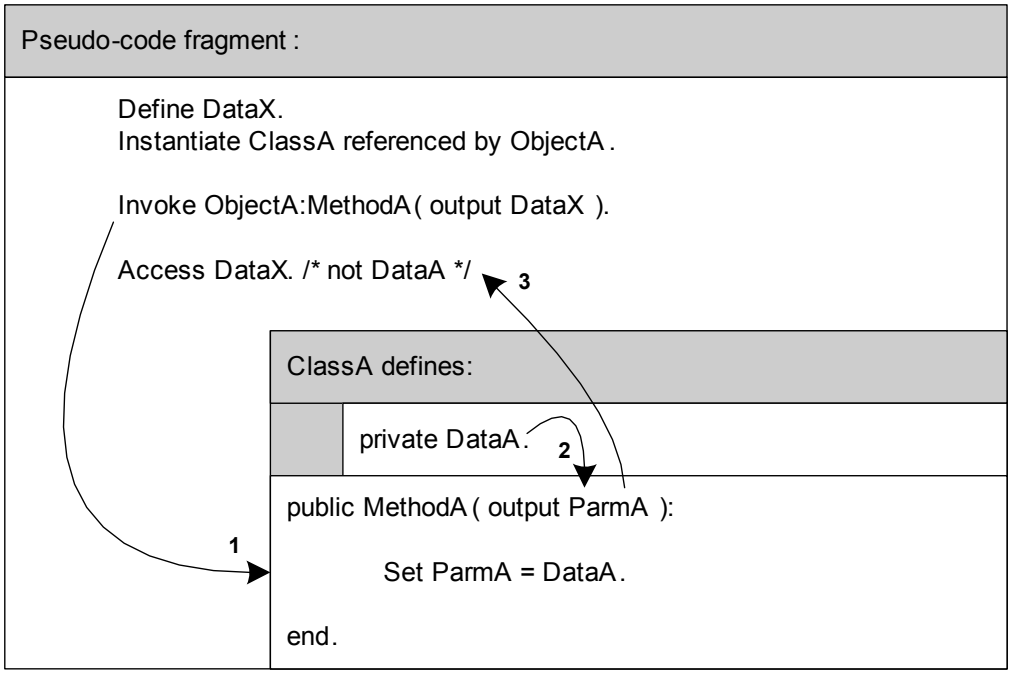

Figure 1–1: Encapsulation

In this figure, ClassA defines DataA as private and directly inaccessible from outside the class definition, and it 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, also allow you to define them 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 methods and properties to access any of that data from outside the class.

Comparison with procedure-based programming

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, and methods 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, and methods for the definition of that class.

Figure 1–2 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.

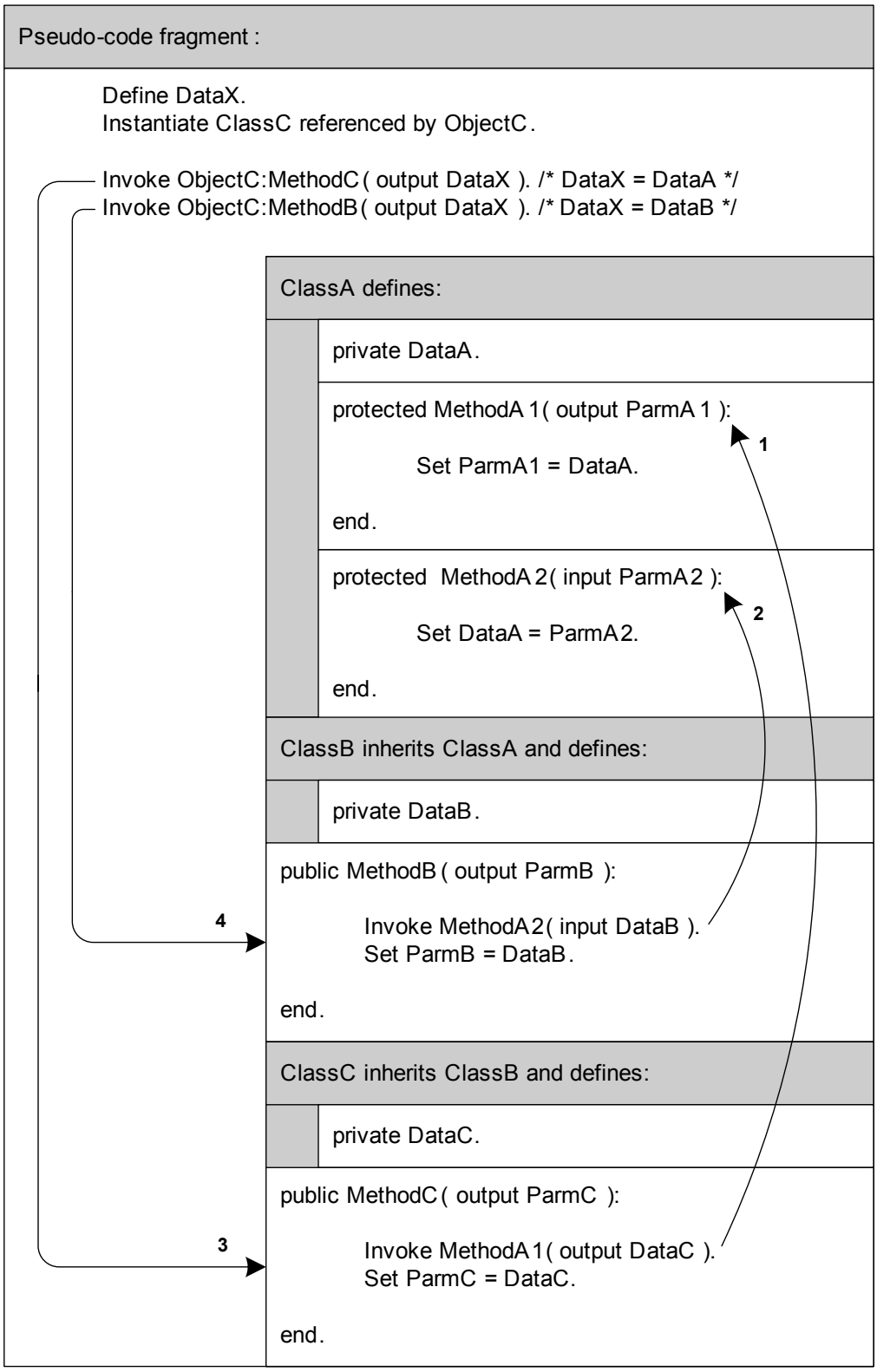

Figure 1–2: Inheritance

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 an abstraction of that 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, static 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 the "Polymorphism" section on page 1–14.

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 the "Using the root class—Progress.Lang.Object" section on page 2–35). A subclass and its class hierarchy can also implement one or more interface types that the subclass explicitly specifies. However, interfaces cannot inherit from a class or another interface, nor can they, themselves, be inherited by a class.

Comparison with procedure-based programming

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, and there by providing an instance of the container class 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

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.

Figure 1–3 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.

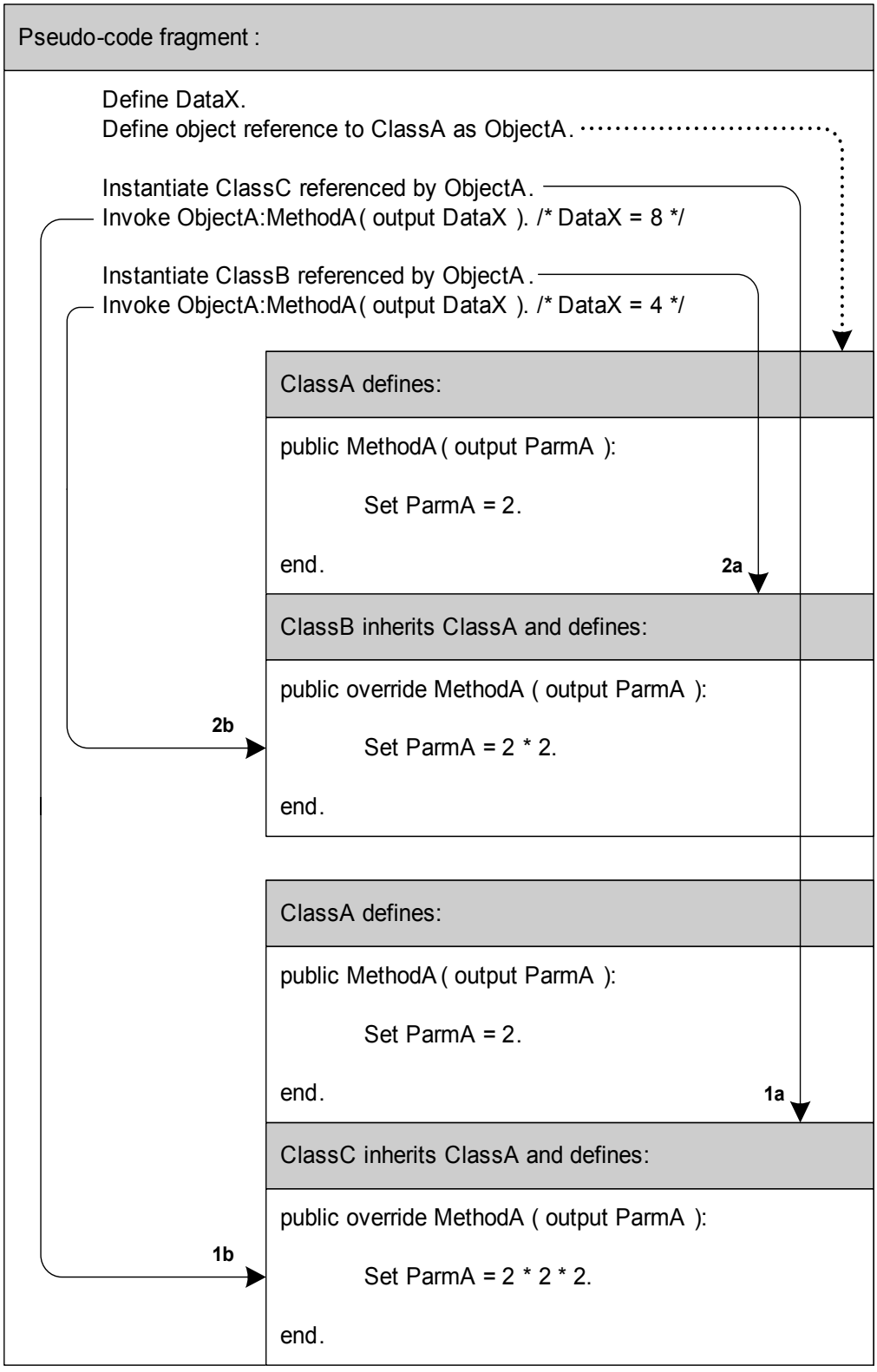

Figure 1–3: Polymorphism with method overriding

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 type 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.

Figure 1–4 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 using a super class shown in Figure 1–3.

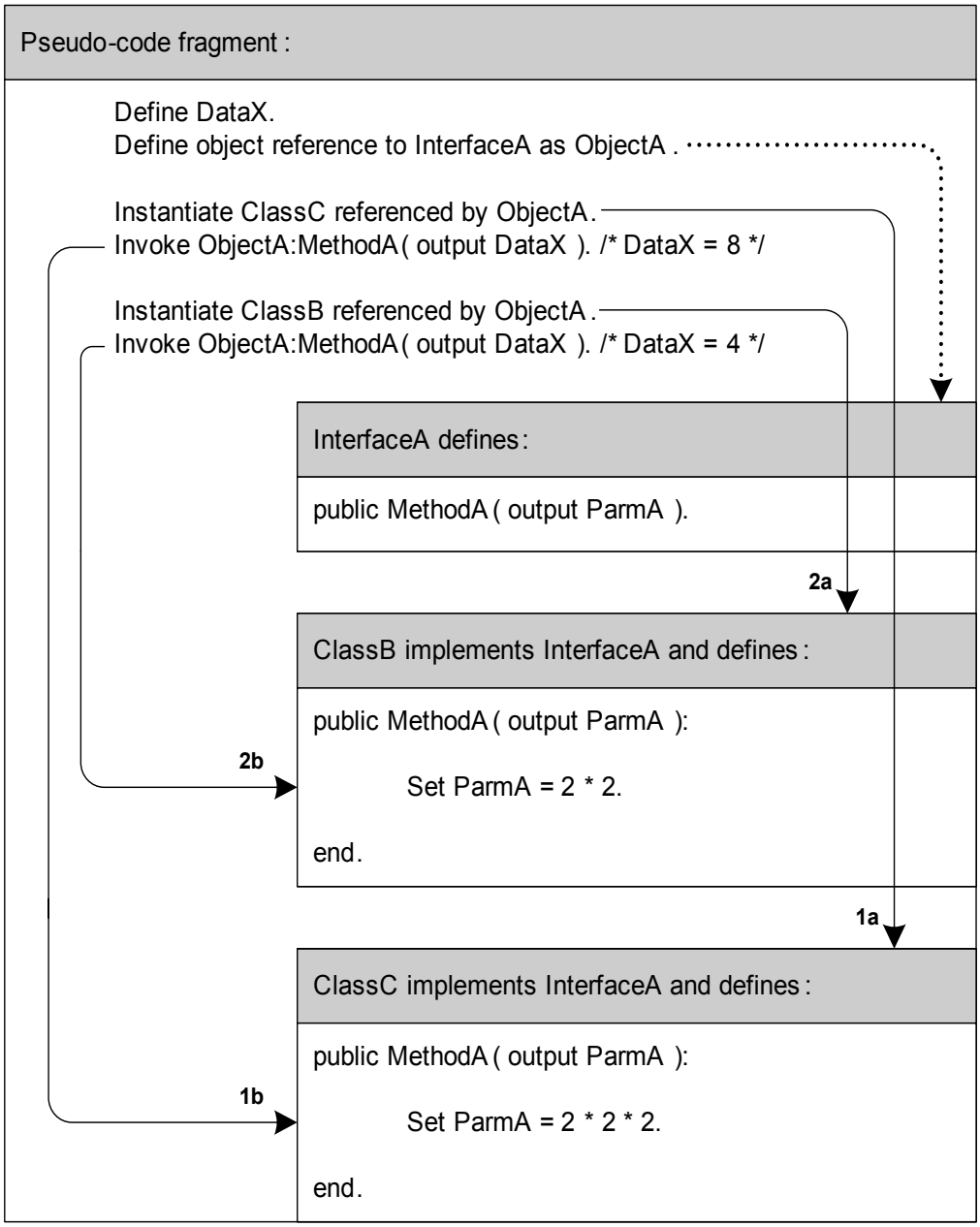

Figure 1–4: Polymorphism with interfaces

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 type of polymorphism, using interfaces, can also be used to implement delegation. For more information on delegation, see the "Delegation" section on page 1–13.

Note that method overloading also provides a less powerful mechanism, similar to (and sometimes referred to as a type 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 the "Method overloading" section on page 1–19.

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.

Figure 1–5 shows an example of overloading where a ClassA defines two overloads of a MethodA( ).

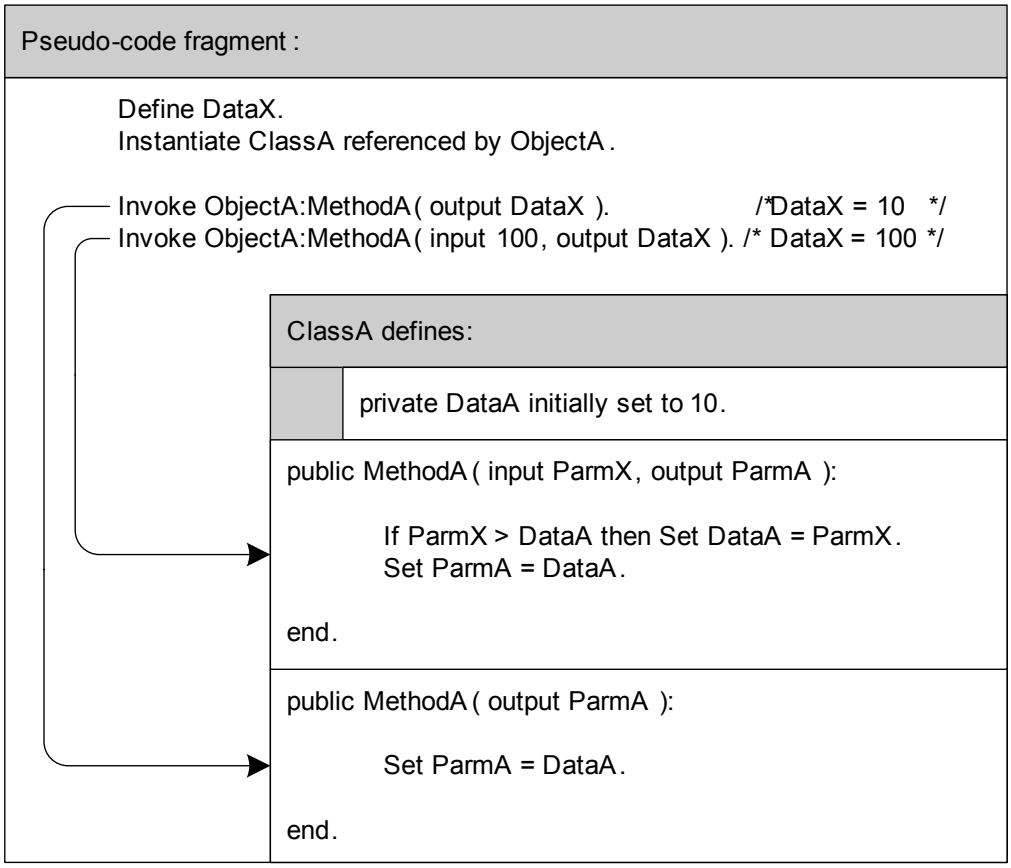

Figure 1–5: Method overloading

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 the "Polymorphism" section on page 1–14.

Strong typing

Strong typing is the enforcement of type conformance. Classes essentially define data types, which are analogous to the built-in data types of the language. Classes are identified with a type name that includes the name of the class. The name of the class matches the filename of the file that stores the definition of a user-defined class. The type name that includes this class name allows ABL to locate the specified class file at both compile time and run time. From the defined type name, the class file also implicitly defines the class as a data type, which the compiler uses to validate references to the class.

This is similar to data type checking for variables. ABL uses the entire class definition, including all of its data members, properties, and methods, as a distinct type which must be perfectly matched by all references to the class. All type usage is validated at compile time ensuring that a method cannot be invoked on a class unless a method with the exact 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 multiply two character strings by each other, because only numeric data types can be multiplied.

Comparison with procedure-based programming

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

Table 1–1 defines the terms used throughout this manual to define basic object-oriented concepts.

Table 1–1: Glossary of terms *(1 of 3)*

Term	Meaning
Abstraction	Defines the essential characteristics of an object that are visible from outside the object context. An object's abstraction separates its defined behavior from its implementation of that behavior. You can think of the object's behavior as the services it provides to consumers of that object, including procedures and other objects. The object’s abstraction, then, establishes a contract agreed on between the object and its consumers for how consumers can access the object’s behavior.
Class	Provides the definition of a type, which represents the state and behavior for instances of the class (objects). A class defines the data members and properties (representing the state) and methods (implementing the behavior) for any object belonging to the class.
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.

Table 1–1: Glossary of terms *(2 of 3)*

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` and `PROTECTED` data members, properties, and methods from the super class. The subclass can then provide its own implementation of inherited methods (see Method overriding) and also add additional data members, properties, and methods of its own. Inheritance can thus be viewed as a specialization mechanism. 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.
Interface	Provides the definition of a type that specifies a contract consisting of methods that must be defined by any class that implements the interface. An interface assures a common way for accessing functionality that might differ depending on its implementation. All methods in an interface are abstract, meaning that the interface provides only a method name and signature definition (prototype) for each method, but no implementation.
Member	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, and methods.
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 class type.
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 the ABL.

Table 1–1: Glossary of terms *(3 of 3)*

Term	Meaning
Object reference	A value that provides access to an instance of a class and its public members. An object reference is strongly typed and can be stored as a data member or property of a class or as 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 the "Using the root class—Progress.Lang.Object" section on page 2–35.
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 sublcass in a given class hierarchy is called the *most derived class* (*or subclass*).
Type	A name used to identify the state and behavior of an object. The type specifies the structure and semantics of the object, but not its implementation. The implementation of an object is specified by its class. Types are used to enforce strong typing.

Overview of class-based ABL

This section introduces ABL support for programming with classes. The syntax and behavior of each ABL element described in this section is described more completely in other chapters of this manual.

Defining 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 CLASS statement can only appear in a source file defined as a class definition file (.cls file type), and it defines the class type name that any other class or procedure can use to reference this class and that any subclass can use to inherit from this class. (For more information on class definition files, see the "Class definition files and type names" section on page 2–2.) 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 interface classes that define methods that the class implements. The class can define itself as FINAL, which prevents it from being inherited by a subclass. A class can contain essentially the same kinds of definitions for variables and other data elements as procedures, but instead of internal procedures and user-defined functions, a class contains definitions for various types of methods. Classes are always PUBLIC.

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

- Any number of data members, including ProDataSets, temp-tables, and simple variables. (See the "Defining data members" section on page 1–26.)
- Any number of properties, which are similar to simple variables. (See the "Defining properties" section on page 1–27.)
- Any number of ON statements, which specify ABL triggers for widget and low-level events. (See the ON statement reference entry in *OpenEdge Development: ABL Reference*.)
- Any number of named methods. (See the "Defining methods" section on page 1–25.)
- Any number of user-defined function prototypes (not the functions themselves). (See the FUNCTION statement reference entry in *OpenEdge Development: ABL Reference*.)
- Any number of optional constructors. (See the "Defining constructors" section on page 1–25.)
- An optional destructor. (See the "Defining the destructor" section on page 1–26.)

The main block of a class cannot contain any executable statements that are outside of a method, constructor, destructor, or ON statement definition. For more information on defining classes, see the "Defining classes" section on page 2–8.

Comparison with procedure-based programming

A persistent procedure can contain executable statements in its main block and can define parameters. Classes provide a constructor 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.

Defining methods

Within the main block of a class, you implement executable behavior using methods. 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 of PRIVATE, PROTECTED, or PUBLIC, where PUBLIC is the default. Where and how a method can be invoked depends on its access mode:

- PRIVATE — You can invoke a private method only from within the class itself.
- PROTECTED — You can invoke a protected method only from within the class hierarchy.
- PUBLIC — You can invoke a public method from inside or outside the class hierarchy.

A method can have parameters and a defined return type (similar to a user-defined function); 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. Each method defined within a class must have a unique identity, but multiple methods in the class can have the same name (can be overloaded) as long as they all have unique calling signatures. For more information on defining methods, see the "Defining methods within a class" section on page 2–25.

Comparison with procedure-based programming

Methods generally combine the characteristics of internal procedures and user-defined functions, and add features unique to classes. Also, unlike user-defined functions, methods can raise the ERROR condition.

Defining constructors

A class can define a special method called a constructor, which allows you to execute behavior 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 an access mode of PRIVATE, PROTECTED, or PUBLIC, where PUBLIC is the default. Where and how a constructor can be invoked depends on its access mode:

- PRIVATE — You can invoke a private constructor only from another constructor of the class in which it is defined. If all constructors of the class are defined as PRIVATE, this class cannot be instantiated.
- PROTECTED — You can only invoke it from another constructor of the class or from a constructor of a subclass. 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.
- PUBLIC — You can use it to directly instantiate the class in which it is defined.

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. For more information on defining constructors, see the "Defining class constructors" section on page 2–29.

Comparison with procedure-based programming

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.

Defining 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 more information on defining destructors, see the "Defining the class destructor" section on page 2–33.

Comparison with procedure-based programming

Persistent procedures have no equivalent for a destructor.

Defining data members

Variables and most 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 (`PRIVATE`, `PROTECTED`, or `PUBLIC`) differ depending on the type of data member. By default, all data members are `PRIVATE`. For more information on defining data members, see the "Defining data members within a class" section on page 2–15. You can also define data elements known as a properties. For more information, see the "Defining properties" section on page 1–27.

Comparison with procedure-based programming

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.

Defining properties

Properties are data elements similar to variables with defined behavior associated with them. This behavior specifies if a property can be read or written and 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 (PRIVATE, PROTECTED, or PUBLIC), which can be separately defined for one of the two specified accessors. By default, all properties are PUBLIC (unlike data members). For more information on defining properties, see the "Defining properties within a class" section on page 2–19. You can also define data elements known as a data members. For more information, see the "Defining data members" section on page 1–26.

Comparison with procedure-based programming

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.

Defining interfaces

An interface declares method prototypes and certain related data definitions that must be implemented by other classes that specify the interface as part of their own class definitions. 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 INTERFACE statement can only appear in a source file defined as a class definition file (.cls file type), and it 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 the "Class definition files and type names" section on page 2–2.) 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. The only data that an interface can define are temp-tables and ProDataSets that are used as parameters to its methods. Methods are the only members of an interface and are limited to defining method prototypes that contain no executable code. Members of an interface are always PUBLIC. For more information on defining interfaces, see the "Defining interfaces" section on page 2–37.

Comparison with procedure-based programming

Persistent procedures have no equivalent for an interface.

Defining data 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 data 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.

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 declaration.

The data type name for a class or interface consists of two parts—a class or interface name preceded by a qualifying package name. 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. For more information on class or interface type names, see the "Defining and referencing user-defined type names" section on page 2–3

An object reference is comparable to the `HANDLE` data type, with the important distinction that the object reference identifies the specific user-defined data type that the variable or field can be used to 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 also define a class or interface data type for the following object references:

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

For more information on defining and using object references to class and interface data types, see Chapter 4, "Programming with Class-based Objects."

Specifying 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 type names you can reference using their unqualified class or interface names (without specifying the package). For more information, see the "Referencing a type name without its package qualifier" section on page 2–6.

Creating and destroying 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 Assignment (=) statement with the NEW phrase to create an instance of a class, and assign the object reference for that instance to an appropriate object reference data element. This statement invokes the specified class constructor to complete class instantiation. You are responsible for deleting (destroying) any instance of a class that you create once it is no longer needed. You can delete class instances using the DELETE OBJECT statement. This statement invokes any destructor specified for the class instance.

The first statement in an ABL application that creates an instance of a class with the NEW phrase must appear in a procedure file (not a class definition file). You cannot create the first instance of a class within a class file, which itself must also be instantiated as a class. Thus, you must use an initial procedure file to startup a class-based application.

For more information on managing the creation and destruction of class instances, see the "Managing the object life-cycle" section on page 2–41. For more information on how the class hierarchy of an object is created and destroyed, see Chapter 3, "Designing Objects—Inheritance, Polymorphism, and Delegation." For more information on using the NEW phrase to instantiate a class, see the "Creating a class instance using the NEW phrase" section on page 4–4.

Comparison with procedure-based programming

Using theAssignment (=)statement with the NEW phrase 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 (through other procedure files) creates procedure objects, a class-based ABL application must also begin with a startup procedure file that directly or indirectly creates class-based objects. (You cannot directly startup 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 either a persistent procedure or an instance of a class.

Invoking methods

You can invoke a method defined or inherited within a class simply by naming it and, if it returns as value, by including it in an expression, much as with a user-defined function. 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 instance separated by a colon (:). Invoking a method *on* (defined in) another object is analogous to invoking a built-in method on an ABL handle-based object.

Comparison with procedure-based programming

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.

Accessing data members and properties

You can access an accessible data member or property defined or inherited within a class simply by naming it. 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 (:). 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.

Comparison with procedure-based programming

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.

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** — Invokes a specified constructor of the immediate super class from a constructor of the invoking subclass. For more information, see the "Constructing an object" section on page 3–16.
- **SUPER system reference** — Invokes the implementation of a specified method within a class hierarchy as implemented in the most derived (lowest) super class above the invoking subclass. Analogous to the RUN SUPER statement or SUPER function in procedure-based programming. For more information, see the "Calling a super class method" section on page 3–21.

- **CAST function** — Allows a super class object reference to be assigned to a related subclass object reference. For more information, see the "Assignment and the CAST function" section on page 4–37.

- **TYPE-OF function** — Verifies the type of an object. For more information, see the "TYPE-OF function" section on page 4–30.

- **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 the "VALID-OBJECT function" section on page 4–29.

- **THIS-OBJECT statement** — Invokes a specified constructor of the defining class from another constructor of the same class. For more information, see the "Constructing an object" section on page 3–16.

- **THIS-OBJECT system reference** — Returns an object reference to the current object from a method within the class hierarchy of the object. Analogous to the THIS-PROCEDURE system handle for procedures. For more information, see the "THIS-OBJECT system reference" section on page 4–31.

- **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 user-defined 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 the "ABL session object references" section on page 4–33.

- **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 user-defined 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 the "ABL session object references" section on page 4–33.

- **RETURN statement** — Sets the return value for a method. With the ERROR option:

  – From within a method, raises the ERROR condition on the statement that invoked the method and returns an optional error string that is accessible using the RETURN-VALUE function. For more information, see the "Defining methods within a class" section on page 2–25.

  – From within a constructor, cancels and reverses any class instantiation, raising the ERROR condition on the statement that attempted to create the object and returning an optional error string that is accessible using the RETURN-VALUE function following statement execution. For more information, see the "Defining class constructors" section on page 2–29.

  For more information on error handling with classes, see the "Raising and handling error conditions" section on page 4–46.

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 *OpenEdge Development: ABL Handbook*).

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 startup an application (whether procedure-base or class-based) using a procedure file.

Programming conventions for classes

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

- All class names and interface names should begin with an upper case letter and each word in the name starts with an upper case letter. This is known as camel case, for example, `MyCustomerClass`.
- Keywords should not be used for class names, method names, data member, or property names. Some keywords are marked illegal by the compiler if used as class names.
- 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`.

2

Getting 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 prototypes for methods that any class implementing the interface must define. Thus, an interface type allows multiple user-defined 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 in order to access the class-specific implementation of a given interface-declared method.

Thus, you can use the class hierarchies and interfaces that define your objects to manage the implementation of an application. The following sections describe how to define classes and interfaces, and manage the life-cycle of objects that you create with them:

- Class definition files and type names
- Defining classes
- Using the CLASS construct
- Defining interfaces
- Using the INTERFACE construct
- Managing the object life-cycle

Class definition files and 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 the "Comparing class definition files and procedure source files" section on page 2–7.

Each class or interface definition represents a user-defined type that you identify with a class or interface type name. This 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 or interface type names, see the "Defining and referencing user-defined type names" section on page 2–3.

Class definition file structure

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

Syntax

```
[ USING-statement ] ...

{ CLASS-statement | INTERFACE-statement }

    class-or-interface-definition

END-statement
```

Element descriptions for this syntax diagram follow:

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. For more information, see the "Referencing a type name without its package qualifier" section on page 2–6.

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 the "Defining classes" section on page 2–8.

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 other class or interface types. For more information, see the "Defining interfaces" section on page 2–37.

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 prototypes (and related data definitions) for methods that can later be implemented by user-defined classes. For more information, see the "Defining classes" section on page 2–8. For more information, see the "Defining interfaces" section on page 2–37.

END-statement

The statement that terminates a class or interface definition. For classes, see the "Defining classes" section on page 2–8. For interfaces, see the "Defining interfaces" section on page 2–37.

Defining and referencing user-defined type names

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

Syntax

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

Element descriptions for this syntax diagram 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 pathname 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 pathname 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. A 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 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.

Note: You cannot define a class name using an ABL reserved keyword or built-in ABL scalar data type name, such as `INTEGER`.

For example, 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 be found at run time in the file, `SomeClass.r`, and in the same directory path, `topdir/subdir`, relative to `PROPATH`.

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 user-defined type names, see the User-defined type name syntax reference entry in *OpenEdge Development: ABL Reference*.

Once you define a class or interface type, you can reference it by using its fully qualified 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 the "Referencing a type name without its package qualifier" section on page 2–6.

Valid and invalid type name references

Any user-defined 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 user-defined type names are specified, most of which are described in this book. Although you can specify a type name with or without surrounding quotes, you cannot use a character variable or character expression to specify a type name.

Assuming appropriate package configurations on `PROPATH`, these are examples of valid type names that you can specify anywhere a user-defined 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 user-defined type names, where myLocalVariable is a character variable:

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

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

These invalid examples illustrate one of the key aspects of using classes, namely strong typing. In the first example of an invalid type name, if a character variable, like myLocalVariable, is allowed to specify a 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 type name, because character literals that are concatenated using the "+" operator are only evaluated at run time, the compiler cannot assemble a type name from a character expression.

However, you can combine a preprocessor name with other parts of a user-defined type name, as in the following example, because this combination is evaluated and combined into a single name element at compile time:

```
{&AccountingPackage}myClassName
```

How type names are used to locate type definitions on the PROPATH

The examples in Table 2–1 show several type names and where the corresponding class files must be located for the specified type names to be valid.

Table 2–1: User-defined type names and PROPATH

The file that defines the following user-defined 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.

Referencing a type name without its package qualifier

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 user-defined 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:

Syntax

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

You can specify *type-name* as the fully qualified 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. For more information on user-defined type names and packages, see the "Defining and referencing user-defined type names" section on page 2–3.

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.

Comparing 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 or between a starting `INTERFACE` statement and a terminating `END INTERFACE` statement. The `CLASS` or `INTERFACE` statement is necessary to specify the user-defined type 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 user-defined data 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` statement main block outside of method definitions. However, unlike procedure files that allow executable code in the main block that executes when the procedure file is executed, class definition files do not allow any executable code to appear in the main block of the `CLASS` statement that is outside of method definitions or other blocks that you can define within the `CLASS` statement main block.

Therefore, unlike the executable code that can run in the main block when you instantiate a procedure object, there is no code that you can run in the main block of the `CLASS` statement when you instantiate a class-based object. 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 startup 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.

Defining 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 that provide the behavior of the class. All the members of a user-defined class constitute a user-defined data type. As with other data types, the compiler verifies that references to a class are consistent with its type. Another user-defined type is an interface, which provides a means to enforce a consistent definition for similar class types. (For more information, see the "Defining interfaces" section on page 2–37.) Class-based objects are running instances of classes. At run time there can be many instances of a class, each with its own set of data values. There can also be many object references that point to a single object instance.

After beginning a class definition with a CLASS statement specifying 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 properties that define the state of the class.
- Methods to define behavior of the class, including special methods associated with properties.
- Optional constructors, special methods that are invoked when the class is instantiated.
- An optional destructor, which is a method that is invoked when the object is destroyed.
- ON statements to define ABL widget events for a class.
- User-defined function prototypes to specify user-defined functions that are defined in an external procedure.

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

- Defining state in a class
- Defining behavior in a class
- Defining classes based on other classes

Defining state in a class

Data can exist in three basic forms within a class:

- Data members of the class.
- As properties of a class
- As local variables of a method within the class.

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 the scope of the data member or property. Thus, private data members or properties are only accessible from within the class where they are defined; protected data members or properties are only accessible from within the class where they are defined and in any subclass of the defining class; and public data members or properties are accessible both within the class hierarchy and from outside the class hierarchy of the instantiated object where they are defined. Thus, only public data members and properties of a given class-based object are accessible from another object or procedure that instantiates the class. To fully encapsulate the data members of a class, you typically make all data members private and access them through public property accessors or public methods, as appropriate.

You can define data members as a wide variety of ABL data elements, including variables and static temp-tables, ProDataSets, and other handle-based objects, among others. Of these data members, only variables can be public. You can define variables and some other data members as object references to other class-based objects that the class instantiates. You can define properties as a specific subset of variable data types, including object references.

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

Defining behavior in a class

The behavior within a class is defined using methods. Like data members and properties, methods have an access mode, which defines the scope of the method. The scope of a method determines the context within an application where the method is visible and available for execution. Thus, private methods are only accessible from within the class where they are defined; protected methods are only accessible from within the class where they are defined and in any subclass of the defining class; and public methods are accessible both within the class hierarchy and from outside the class hierarchy of the instantiated object where they are defined. Thus, only public methods of a given class-based object are accessible from another object or procedure that instantiates the class.

Classes support four types of methods:

- 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 when you delete the class instance. 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.

In addition to the various types of methods, you can also define triggers (ON statements) that specify behavior for ABL events, especially events for the widgets that you can define in a class. These triggers handle events only for widgets activated by methods of the defining class and therefore support the behavior of these methods.

Note: Although triggers can be part of a class definition, they are not class members that can be inherited by subclass or affected by an APPLY statement executed in another class definition.

Defining 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, and methods 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) 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 the "Class hierarchies and inheritance" section on page 3–2.

A user-defined class can also implement methods whose prototypes are declared by an interface. An interface is not a class in itself, but similar to a class, it does represent a data type (interface type) that specifies methods for classes to implement. The implementing class does not inherit the methods of an interface, as the interface does not define method implementations, but only the prototypes for methods that the class implements. An interface (defined with the INTERFACE statement) also does not inherit method prototypes from any other interface, but is the primary and common source for a particular set of method prototypes that different user-defined classes can then implement as they require. In fact, any class must define some implementation for every method provided by any interface that it agrees to implement, and it is a compiler error for it not to do so. The implementing class can do this by inheriting method implementations from a super class as well as defining its own method implementations. In this way, any class that implements an interface is guaranteed to support all the methods of that interface. Thus, an object instantiated from a class that implements an interface can also represent 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 represent an instance of each interface type defined by each implemented interface, and the object can be referenced as any one of these interface types. However, note that you cannot instantiate an interface the way you can instantiate a class. You can only represent a class that you have already instantiated as an instance of an interface type that the class implements.

Note: Interfaces can also be used to guarantee the API for applications that implement a delegation model. For more information, see the "Using delegation with classes" section on page 3–28.

Using the CLASS construct

The CLASS construct represents all of the statements that can comprise a user-defined class definition. The class definition specifies a main block that begins 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 type-name [ INHERITS super-type-name ]
  [ IMPLEMENTS interface-type-name [ , interface-type-name ] ... ]
  [ USE-WIDGET-POOL ] [ FINAL ] :

    [ data-member ... ]
    [ property ... ]
    [ constructor ... ]
    [ method ... ]
    [ destructor ]
    [ trigger ... ]
    [ udf-prototype ... ]

END [ CLASS ] .
```

Element descriptions for this syntax diagram follow:

type-name

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

Syntax

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

package

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

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 the "Defining and referencing user-defined type names" section on page 2–3.

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 type name for the class or its *package*. For more information, see the "Referencing a type name without its package qualifier" section on page 2–6.

`INHERITS` *super-type-name*

The immediate super class that this class inherits, where *super-type-name* is the type name of this super class. Logically the super class's non-private members can be thought of as if they were 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 *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 the "Referencing a type name without its package qualifier" section on page 2–6. Based on this information, ABL must be able to locate the specified super class 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 the "Using the root class—Progress.Lang.Object" section on page 2–35.

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

One or more interfaces that this class implements. The current class hierarchy must implement all of the methods declared in each interface specified by *interface-type-name*.

The syntax for *interface-type-name* is the same as for *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 the "Referencing a type name without its package qualifier" section on page 2–6. 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`

An unnamed widget pool scoped to the current class instance. This 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.

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.

The AVM deletes the associated class unnamed widget-pool (specified using this option) when each instance of the class is deleted. For more information on using widget pools with classes, see the "Using widget pools" section on page 5–7.

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.

data-member

A data member definition. A class can define zero or more data members of the class. For more information, see the "Defining data members within a class" section on page 2–15.

property

A property definition. A class can define zero or more properties of the class. For more information, see the "Defining properties within a class" section on page 2–19.

constructor

A constructor definition. A class can define zero or more constructors for the class. For more information, see the "Defining class constructors" section on page 2–29.

method

A method definition. A class can define zero or more methods of the class. For more information, see the "Defining methods within a class" section on page 2–25.

destructor

A destructor definition. A class can define one destructor for the class. For more information, see the "Defining the class destructor" section on page 2–33.

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 ABL events, mostly from interactions with ABL widgets. For more information, see the "Handling events" section on page 5–5.

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 *OpenEdge Development: 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 a class definition:

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

...

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 chapters (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 chapter of this manual. For more information, see the "Comparing constructs in classes and procedures" section on page 5–9.

Defining 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 ProDataSets, temp-tables, queries, buffers, data sources, variables, streams, work tables, and static GUI objects as data members of a class, with restrictions as noted below. You can also define certain types of instance data as properties. For more information, see the "Defining properties within a class" section on page 2–19.

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 and user-defined functions.

This is the syntax for defining a variable data member:

Syntax

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

This is the syntax for defining other non-visual data members, which cannot be defined as PUBLIC:

Syntax

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

This is the syntax for defining widgets and other data members that cannot be defined as PUBLIC or PROTECTED:

Syntax

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

Element descriptions for these data member syntax diagrams follow:

[PRIVATE | PROTECTED | PUBLIC]

The access mode for the data member. The default access mode is PRIVATE. PRIVATE data members can only be accessed by the class defining the data member. PROTECTED data members can be accessed by the class defining them and by any class that inherits from that class. PUBLIC data members can be accessed by the class defining them, by any class that inherits from that class, and by other classes and procedures that reference an instance of that class. For more information on accessing data members, see the "Accessing data members and properties" section on page 4–16.

Only data members in the main block of a class can have an access mode. Local variables defined within methods cannot have an access mode. They are always scoped to the method, similar to the way local variables defined within an internal procedure or function are scoped.

Visual and other objects that share the same definition syntax can only be defined as PRIVATE. That means that these visual objects, streams, and work-tables can only be used within the class file that defines them. For more information on using visual objects within a class definition file, see the "Defining and using widgets in classes" section on page 4–43.

data-member-name

The name of the variable, buffer, temp-table, query, ProDataSet, data-source, visual object, stream, or work-table.

The *data-member-name* must be unique among all data members in the class and its class hierarchy.

If you define a PRIVATE or PROTECTED ProDataSet, then the temp-tables and buffers of the ProDataSet must also be defined with the same access mode.

There is limited support for the PUBLIC access mode. It is only valid when defining a variable. However, defining a PUBLIC variable with an EXTENT is also not supported.

data-member-definition

The remaining syntax required for the variable, buffer, temp-table, query, ProDataSet and data source.

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 ttCust 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 scopes 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 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 this variation of a sample class, NECustObj, notice that both the PUBLIC and PROTECTED data members, iNumCusts and ttCust, 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 ttCust:
            DISPLAY ttCust.
        END.
    END METHOD.
END CLASS.
```

Following is another class, MyMain, that uses the NEW phrase to create an instance of the acme.myObjs.NECustObj class in an Assignment (=) statement that assigns the object reference of this instance 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 ttCust */
        DELETE OBJECT rNECust.
        rNECust = ?.
    END METHOD.
END CLASS.
```

For more information on accessing data members, see the "Accessing data members and properties" section on page 4–16.

Comparison with procedure-based programming

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, they are scoped to the procedure, including its internal procedures and user-defined functions, but cannot be accessed directly from other external procedures. In the same way, data members defined in a class with no access mode are PRIVATE and accessible from all methods defined in the class, but they cannot be accessed directly from other classes, whether inside or outside of the class hierarchy.

Procedures can define SHARED variables, which are scoped to and accessible from other external procedures. The variable definitions must be defined as NEW SHARED in a parent procedure and the definitions repeated as SHARED in each called external procedure that needs to access them. This is necessary because the compiler does not look at other external procedures when compiling a procedure.

Classes do not support SHARED variables, which must be defined both for where they are stored and for where they are referenced. PROTECTED and PUBLIC data members in classes provide wider access to class data members and without the need to repeat the data member definitions. However, they 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 an object reference to the instantiated class that defines them. These referential restrictions both conform to class and object relationships and allow the compiler to validate data member references for consistency based on these relationships.

Defining properties within a class

You can define certain types of instance 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 return from a method. You must define all other types of instance data as data members. For more information, see the "Defining data members within a class" section on page 2–15.

Unlike data members, the DEFINE statement for defining properties provides a built-in mechanism for encapsulating instance data using special methods called accessors. The accessors that 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 in other class instances.
4. Any internal data defined within the accessors themselves.

Thus, depending on how you program these accessors, you can directly access the default memory defined for the property, or you can access other data that is defined as a data member of the class definition, or even data that is available from outside the class definition itself. No matter how you program property accessors, you can access properties in exactly the same way as you access equivalently defined data members.

So, for example, if the accessors for a property read or write to a data member defined as a temp-table, and the property is defined as an integer, accessing the integer value of the property actually invokes its accessors, which read or write the temp-table in whatever manner you have defined. In this case, the integer could be a record index for temp-table access, or it might have no role in accessing the temp-table at all. Thus, a property can encapsulate data access that is completely hidden when accessing the apparent value of the property.

Typically, however, the data that a property accesses is of the same data type defined for the property itself, and this is the most common and effective use of a property definition. Otherwise, you might not be able to explicitly read and write the same data for the property as its accessors (such as records of a temp-table). To effectively encapsulate types of data that you cannot define explicitly as a property, like temp-tables, you must define named methods with interfaces designed to handle these data types. For more information, see the "Defining methods within a class" section on page 2–25.

This is the syntax for defining a property:

Syntax

```
DEFINE [ PRIVATE | PROTECTED | PUBLIC ] PROPERTY property-name
  type-spec [ NO-UNDO ] accessor-spec
```

Element descriptions for this syntax diagram follow:

[PRIVATE | PROTECTED | PUBLIC]

The access mode for the property. The default property access mode is PUBLIC. PRIVATE properties can only be accessed by the class defining the property. PROTECTED properties can be accessed by the class defining them and by any class that inherits from that class. PUBLIC properties can be accessed by the class defining them, by any class that inherits from that class, and by other classes and procedures that reference an instance of that class. For more information on accessing data members, see the "Accessing data members and properties" section on page 4–16.

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*.

property-name

The property name, which identifies either the property or its default memory, depending on the context.

type-spec

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

Syntax

```
AS { datatype [ INITIAL constant ] | [ CLASS ] type-name }
```

Any ABL data type (*datatype*) that you specify must be one that is allowed as a return type for a method. For any *datatype* that accepts initial values, you can specify an initial literal value (*constant*). If you specify a class or interface type, *type-name* can be a fully qualified class name (specifying the package), or if you have specified the fully qualified class name or its package with a USING statement, *type-name* can be the unqualified class or interface name. For more information on referencing type names, see the "Defining and referencing user-defined type names" section on page 2–3.

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:

Syntax

```
[ PRIVATE | PROTECTED ] get-accessor set-accessor
  | get-accessor [ PRIVATE | 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. (More detailed syntax for each type of accessor follows.) 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 of the accessors, it must be more restrictive than the property access mode that applies to the other accessor.

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 and can be read only from the defining class or 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:

Syntax

```
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:

Syntax

```
GET . | GET ( ) : 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.

`GET ( ) : 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, except statements that block for input. 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. 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 is handled by the inner-most enclosing block according to its ON ERROR definition. If the inner-most block is the accessor, itself, it handles the condition according to the default ON ERROR handling for method and procedure blocks (UNDO, LEAVE).

A RETURN statement in *get-logic* with the ERROR option 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. You can also specify a return string with the ERROR option, whose value can be retrieved using the RETURN-VALUE function after control returns from the statement that is reading the property. For more information on handling errors returned from properties, see the "Raising and handling error conditions" section on page 4–46.

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

Syntax

```
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:

Syntax

```
SET . | SET ( parameter-definition ) : 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.

`SET ( parameter-definition ) : 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:

Syntax

```
INPUT parameter-name AS { datatype | [ CLASS ] 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 (*datatype* or *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, except statements that block for input. 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.

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 is handled by the inner-most enclosing block according to its ON ERROR definition. If the inner-most block is the accessor, itself, it handles the condition according to the default ON ERROR handling for method and procedure blocks (UNDO, LEAVE).

A RETURN statement in *set-logic* with the ERROR option raises the ERROR condition on the statement that is writing 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. You can also specify a return string with the ERROR option, whose value can be retrieved using the RETURN-VALUE function after control returns from the statement that is writing the property. For more information on handling errors returned from properties, see the "Raising and handling error conditions" section on page 4–46.

Note: You must terminate every accessor definition with a period (.), as shown.

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

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

CLASS acme.myObjs.CreditObj INHERITS CommonObj:

    DEFINE PUBLIC PROPERTY CustCreditLimit AS DECIMAL INITIAL ?
        /* 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.
                RETURN ERROR "Over Limit".
            END.
            ELSE DO:
                Customer.Creditlimit = piCL + 1000.
                CustCreditLimit = Customer.Creditlimit.
            END.
        END SET.

    ...

    METHOD PUBLIC VOID CheckCustCredit ( ):
        /* invokes the CustCreditLimit property SET accessor */
        IF AVAILABLE (Customer) THEN DO:
            CustCreditLimit = Customer.Creditlimit NO-ERROR.
            IF ERROR-STATUS:ERROR THEN DO:
                /* Indicates Customer balance is greater than credit limit */
                /* and returns error string from property SET              */
                RETURN ERROR RETURN-VALUE.
            END.
        END.
        ELSE
            RETURN ERROR "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). 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 raises ERROR for an application condition and the public method, CheckCustCredit( ), defined in the same class writes the property value and checks for this ERROR.

For more information on accessing properties, see the "Accessing data members and properties" section on page 4–16. For more information on handling property error conditions, see the "Errors within a property" section on page 4–49.

Comparison with procedure-based programming

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.

Defining methods within a class

Methods define the behavior of a class. Methods can access any data members and properties of the class including non-private (PUBLIC or PROTECTED) data members and properties of each super class in the class hierarchy. Methods are only valid in a class and cannot be defined in a procedure. Methods also have an access mode that defines the scope of the method. The possible access modes are PRIVATE, PROTECTED, and PUBLIC, and PUBLIC is the default. PRIVATE methods can only be accessed by the class that defines them. PROTECTED methods can be accessed by the class that defines them and by any class that inherits from that class. 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 an object reference. For more information, see the "Calling methods from outside a class hierarchy" section on page 4–9.

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 is 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 return ERROR, along with an optional return string, using the RETURN statement that raises the ERROR condition on the statement invoking the method. Any return string can then be returned using the RETURN-VALUE function. For more information on handling errors returned from methods, see the "Raising and handling error conditions" section on page 4–46.

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 in the super class. Only 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. In addition, the access mode of the method should match. However, a PUBLIC subclass method can override a PROTECTED super class method, if you want. Any class that inherits from the subclass will inherit the method of the subclass and will not "see" the method of the super class. The subclass can access the method of the same name in the super class using the SUPER system reference. For more information on overriding methods, see the "Overriding methods within a class hierarchy" section on page 3–7.

A class can define a method using the same name as another method defined in the 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. These overloaded methods can each be overridden by a subclass. 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 also supports certain method overloadings that it disambiguates only at run time. For more information on method overloading, see the "Overloading methods and constructors" section on page 3–12.

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 (except for the RETURN ERROR statement that a method does allow). 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.

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:

Syntax

```
[ PRIVATE | PROTECTED | PUBLIC ] [ OVERRIDE ] [ FINAL ]
```

[PRIVATE | PROTECTED | PUBLIC]

The access mode for the method. The default method access mode is PUBLIC. PRIVATE methods can only be invoked by the class defining the method. PROTECTED methods can be invoked by the class defining them and by any class that inherits from that class. PUBLIC methods can be invoked by the class defining them, by any class that inherits from that class, and by other classes and procedures that reference an instance of that class.

OVERRIDE

If specified, this method overrides a method defined in a super class in the class hierarchy. The compiler verifies that this method and a method of the same name in a super class have the same return type, access mode, and number of parameters. Each parameter's corresponding mode and data type must also match in order. If any of these tests fail, the compiler generates an error. If you want, however, a PUBLIC subclass method can override a PROTECTED super class method.

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 is an object reference to a class. The method can also return VOID which means the method does not return a value.

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. If the name is the same as the name of a non-private method in a super class within its class hierarchy, the subclass method overrides the method in the super class and must use the OVERRIDE option. If the signatures are not the same (see the OVERRIDE option), a compiler error is generated.

[*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 *OpenEdge Development: ABL Reference*.

method-body

The logic of the 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. One statement for use in the *method-body* is a call to a method defined in a super class using the SUPER system reference.

If the method has a return type, the code you can use in the method is limited in almost the same way as for a user-defined function. A method with a return type cannot invoke statements that block for input, such as the UPDATE, SET, PROMPT-FOR, CHOOSE, INSERT, WAIT-FOR, and READKEY statements. You must also return the value for the return type using the RETURN statement. However, unlike a user-defined function, you can raise ERROR on the statement that invokes the method using the RETURN ERROR statement, including use of the error string option to set the value of the RETURN-VALUE function available after the invoking statement returns control to 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 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 ttCust 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.

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

END CLASS.
```

Defining class constructors

A constructor of a class is a special method that gets called when a class is instantiated using the NEW phrase. 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.

A class is not required to define a constructor. A default constructor with no parameters is provided by ABL. This default constructor is only provided when the class has not explicitly defined a constructor for the class. A class can also define multiple constructors that are overloaded with different parameter signatures. If you choose to define a constructor without parameters, that constructor becomes the default constructor for the class. For more information, see the "Overloading methods and constructors" section on page 3–12.

In a class, it is a 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 a constructor can be PRIVATE, PROTECTED, or PUBLIC, and PUBLIC is the default. In order for a class to be instantiated using the NEW phrase, 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 phrase, 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 the "Creating a class instance using the NEW phrase" section on page 4–4)

If no constructor of a class is PUBLIC, the class cannot be referenced in a NEW phrase, because it has not defined itself as being 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 the "Constructing an object" section on page 3–16.) 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 cannot be invoked except from another constructor of the defining class using the THIS-OBJECT statement. 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).

Constructors can raise ERROR, as well as return an optional error string, using the RETURN statement. Any constructor that invokes a RETURN ERROR statement immediately halts class instantiation, destroys any classes that have already completed construction in the class hierarchy, and raises the ERROR condition on the statement that instantiated the class. Any error string can also be returned from the RETURN-VALUE function after the statement that instantiated the class returns control to the instantiating block. 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 the "Raising and handling error conditions" section on page 4–46.

This is the syntax for defining a class constructor:

Syntax

```
CONSTRUCTOR [ PRIVATE | PROTECTED | PUBLIC ] class-name
    ( [ parameter [ , parameter ] ... ] ):

    constructor-body

END [ CONSTRUCTOR ] .
```

Element descriptions for this syntax diagram follow:

[PRIVATE | PROTECTED | PUBLIC]

The access mode for the constructor. The default constructor access mode is PUBLIC. PRIVATE constructors can only be invoked by another constructor defined in the class using the THIS-OBJECT statement. PROTECTED 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. 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 phrase of the Assignment (=) statement.

class-name

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

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

Any parameters that you define for the constructor. If this constructor overloads another 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 reference entry in *OpenEdge Development: 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. Additional statements allowed in a constructor include the SUPER statement, which invokes a constructor defined in the immediate super class, and the THIS-OBJECT statement, which invokes another constructor defined in the current class. The constructor logic is normally used to initialize the data members and properties of the class.

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 the RETURN ERROR statement in 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 the "Constructing an object" section on page 3–16.

Adding a constructor 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 ttCust 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.

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

        iNumCusts = 0.
        /* Fill temp table and get row count */
        FOR EACH Customer WHERE CreditLimit > 50000:
            CREATE ttCust.
            ASSIGN
                iNumCusts = iNumCusts + 1
                ttCust.RecNum = iNumCusts
                ttCust.CustNum = Customer.CustNum
                ttCust.Name = Customer.Name
                ttCust.State = Customer.State.
        END.
        rMsg = MessageHandler (INPUT "acme.myObjs.CustObj").
    END CONSTRUCTOR.

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

END CLASS.
```

Comparison with procedure-based programming

A 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 phrase, the specified constructor executes as part of the class constructor hierarchy and returns to the caller assigning 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.

Defining 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 a DELETE OBJECT or class instantiation is halted by a RETURN ERROR statement. 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 and are always PUBLIC. A destructor cannot have any parameters.

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. As with persistent procedure instances and dynamic data objects, an application is responsible for deleting all class instances that are created by the application. This is done using the DELETE OBJECT statement for each class instance.

The AVM does not automatically delete class instances until the entire ABL session ends. When your application exits a module or other part of the application where class instances have been used, it is your responsibility to delete all class instances that are still running but no longer needed. By deleting these objects, you ensure not only that the resources allocated for each class instance itself are deleted, but also that the destructor for each of these objects is invoked. 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 the "Using widget pools" section on page 5–7.) Otherwise, you can code the destructor to explicitly delete resources that have otherwise been created by the class instance.

When the session does end, the AVM does not automatically run the destructor for any objects that have not been deleted by the application. The AVM does automatically delete any ABL-related resources, such as persistent procedure and class instances, visual objects, and data objects. However, any other tasks that you have programmed in the destructor, such as writing to a log file, do not occur unless your application has explicitly deleted the associated object.

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.

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 (the DELETE OBJECT statement) that it failed. It is illegal for a destructor to execute the RETURN ERROR statement to raise an ERROR condition in the caller.

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 *type-name* used in the CLASS statement.

destructor-body

The logic of the destructor, which can be composed of any ABL statements allowed within a method block except the RETURN statement (with or without the ERROR option). 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 ttCust 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.

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

        iNumCusts = 0.
        /* Fill temp table and get row count */
        FOR EACH Customer WHERE CreditLimit > 50000:
            CREATE ttCust.
            ASSIGN
                iNumCusts = iNumCusts + 1
                ttCust.RecNum = iNumCusts
                ttCust.CustNum = Customer.CustNum
                ttCust.Name = Customer.Name
                ttCust.State = Customer.State.
        END.
        rMsg = MessageHandler (INPUT "acme.myObjs.CustObj").
    END CONSTRUCTOR.

    ...

    DESTRUCTOR PUBLIC CustObj( ):
        EMPTY TEMP-TABLE ttCust.
        DELETE OBJECT rMsg.
        rMsg = ?.
        DELETE OBJECT rCreditObj.
        rCreditObj = ?.
    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 and deleting other class instances that the class creates. For information on deleting class-based objects, see the "Managing the object life-cycle" section on page 2–41.

Comparison with procedure-based programming

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.

Using 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` 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.

Table 2–2 describes the common properties and methods on `Progress.Lang.Object`.

Table 2–2: Progress.Lang.Object public properties and methods *(1 of 3)*

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.

Table 2–2: Progress.Lang.Object public properties and methods *(2 of 3)*

Property or method	Description
`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 the "Reflection—Using the Progress.Lang.Class class" section on page 4–54.
`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 )`	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 determine if two different object references are equivalent and return `TRUE` or `FALSE` depending on the result. The two objects to compare are the current object instance and the object instance referenced by *`other-obj`*. **Note:** Use the equals operator (`EQ` or `=`) to determine if the two object reference point to the same object.

Table 2–2: Progress.Lang.Object public properties and methods *(3 of 3)*

Property or method	Description
`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 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 the "Assignment and the CAST function" section on page 4–37

Defining interfaces

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

A user-defined 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 behavior from its super class. If some behaviors represent a set of methods that other objects need to invoke, but for which there is no common implementation, an interface provides a reliable way of defining a uniform contract for all implementations of its methods to satisfy. Any caller can then count on the consistent definition of a set of methods and their signatures, even though the implementation of these same methods might differ completely from one class to another.

Using the INTERFACE construct

The INTERFACE construct represents all of the statements that can comprise an interface definition. The interface definition specifies a main block that begins with an INTERFACE statement that identifies the class. 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 method names and signatures that any class implementing the interface must define. Interfaces do not contain any data members or properties but might contain temp-table and ProDataSet definitions that are used for parameters in specified method prototypes. No allocation is associated with these definitions. These temp-table and ProDataSet definitions must be specified before any method prototypes in the main block of the interface. 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.

This is the syntax for defining an interface:

Syntax

```
INTERFACE type-name :

    [ { temp-table | ProDataSet } ... ]

    [ method ... ]

END [ INTERFACE ] .
```

Element descriptions for this syntax diagram follow:

type-name

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

Syntax

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

package

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

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 the "Defining and referencing user-defined type names" section on page 2–3.

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 type name for the interface or its *package*. For more information, see the "Referencing a type name without its package qualifier" section on page 2–6.

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. The class that implements this interface must define a data member that matches each data definition in the interface.

method

A method prototype. Each method prototype is declared by a single `METHOD` statement, a prototype with a signature, but with no implementation and no `END METHOD` statement. All methods are public and a `PUBLIC` access mode must be explicitly specified for each method prototype contained in an interface definition. Method prototypes can be overloaded. Interfaces cannot contain constructors or destructors. The class that implements this interface must fully define a corresponding method that matches each method prototype in the interface.

Using an interface definition

Interfaces cannot be instantiated but they can be used as the data type to define an object reference, such as in a `DEFINE VARIABLE` statement or a `DEFINE PARAMETER` statement. The value of the object reference can then be assigned to be 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 be used to call these common methods even though the underlying objects are of different classes. This is similar to having multiple classes inheriting from the same super class although an interface provides no method implementation. An object reference to the super class can be used to call the methods that are defined in the super class, even though the underlying objects are of different classes.

The following example shows a simple interface definition:

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

    METHOD PUBLIC VOID printObj ( ).
    METHOD PUBLIC VOID printObj (INPUT copies AS CHARACTER).
    METHOD PUBLIC VOID logObj (INPUT filename AS CHARACTER).

END INTERFACE.
```

Note that the interface declares two overloads of the `printObj( )` method.

The following partial sections of the sample class definition show its update to implement this interface:

```
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.
        ...
        OUTPUT CLOSE.
    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):
        DEFINE VARIABLE iCnt AS INTEGER.
        OUTPUT TO PRINTER.
        IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
            ...
        END.
        OUTPUT CLOSE.
    END METHOD.

    METHOD PUBLIC VOID logObj(INPUT fileName AS CHARACTER):
        OUTPUT TO VALUE(fileName).
        ...
        OUTPUT CLOSE.
    END METHOD.

END CLASS.
```

Managing the object life-cycle

You are responsible for the lifetime of a class instance. You must delete the object when it is no longer needed. If the variable used to hold the object reference obtained from the NEW phrase goes out of scope and the object has not been deleted, the object remains in memory even though there is no reference to the object. This can lead to a potential memory leak, as it can with any other dynamically created objects in ABL.

Therefore, as long as you need the object, you must also 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.

Deleting a class instance

To avoid memory leaks during the execution of the application, you must delete an instantiated object when it is no longer needed. To delete a class instance, you must use the DELETE OBJECT statement.

This is the syntax for deleting a user-defined object:

Syntax

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

Element descriptions for this syntax diagram follow:

object-reference

A reference to an instantiated object.

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. An application can use the destructor for a class to release any resources that it has acquired during the execution of the object instance. If an object is not explicitly deleted, it remains instantiated throughout the entire ABL session.

When the client session is shut down, the AVM deletes all remaining class instances and frees all resources associated with them. However, the destructors for these objects are not automatically executed when the session ends and the application exits. If it is important to invoke object destructors, the application must explicitly delete all instantiated objects using the DELETE OBJECT statement prior to termination of the session. This might be necessary especially if the application has allocated system resources external to the AVM itself.

For example, to delete the instance of the sample class, you can use DELETE OBJECT, and then, as with ABL handles, set the object reference to the Unknown value (?) to assure that it cannot be mistakenly de-referenced, as shown:

```
DELETE OBJECT myCustObj.
myCustObj = ?.
```

Note: You can also use DELETE OBJECT THIS-OBJECT in a constructor to conditionally abort class instantiation, which sets the associated object reference to the Unknown value (?). However, in a constructor, you might prefer to use the RETURN statement with the ERROR option, which can provide more useful information to the instantiating context. For more information, see the "Errors within a constructor" section on page 4–52. For more information on the THIS-OBJECT system reference, see the "THIS-OBJECT system reference" section on page 4–31.

For more information on deleting class-based objects, see the "Deleting an object" section on page 3–20.

3

Designing Objects—Inheritance, Polymorphism, and Delegation

The previous chapters introduce the basic syntax that supports classes in ABL. This chapter provides more conceptual material and shows how you might use classes in an application, as described in the following sections:

- Class hierarchies and inheritance
- Using polymorphism with classes
- Using 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 the "Polymorphism" section on page 1–14). 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, and methods 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 class `Progress.Lang.Object`.

Figure 3–1 shows the sequence class construction and inheritance during instantiation of a class hierarchy.

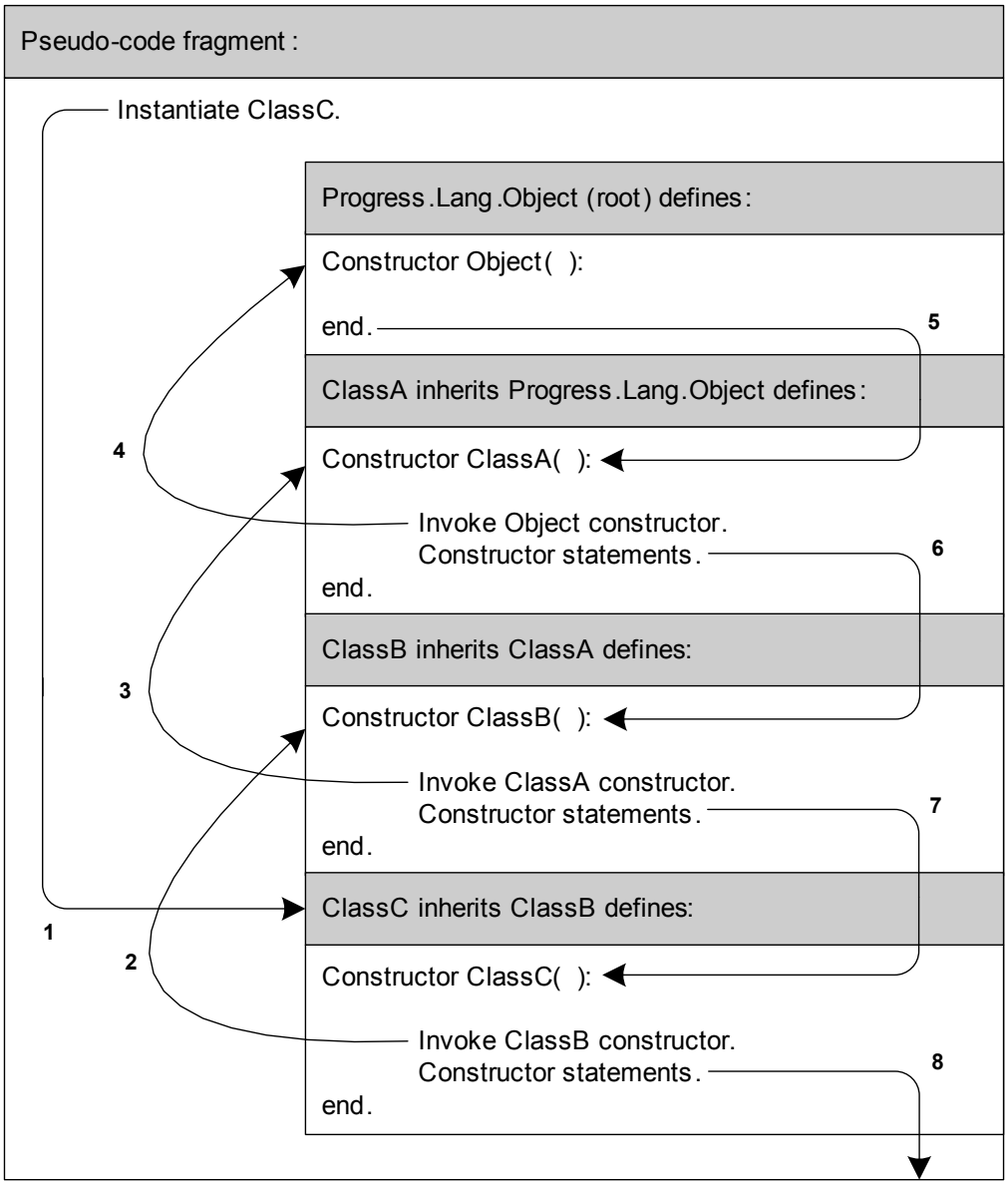

Figure 3–1: Instantiation of a class hierarchy

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.

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 a previous chapter (see Chapter 1, "Object-oriented Programming and ABL"), 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 in its class hierarchy, and as any interface type that it implements. 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 the "Compiling class definition files" section on page 6–5.

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 Figure 3–1). 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 user-defined data 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 classes (see the "Sample classes" section on page 5–10), where `acme.myObjs.Common.CommonObj` is the top user-defined super 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 (see the "Overriding methods within a class hierarchy" section on page 3–7). 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.

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 methods and data defined within the class hierarchy of `CommonObj` can be referenced using the object.

Comparison with procedure-based programming

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.

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 the "Comparing constructs in classes and procedures" section on page 5–9.

Method scoping within a class hierarchy

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. You cannot invoke 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. 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.

Comparison with procedure-based programming

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 PRIVATE, PUBLIC, or PROTECTED. Because of this difference, these built-in data types are referred to as data members when instances are defined for use at the class level. For more information on what access modes are valid for each type of data member definition, see the "Defining data members within a class" section on page 2–15. By default, all data members are PRIVATE.

Properties provide another means to define instances of certain data types for use at the class 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.

Comparison with procedure-based programming

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. However, the PROTECTED access mode also allows you to use data member or property definitions throughout the hierarchy without having to repeat the definitions in each class that uses them.

You can achieve something similar to PUBLIC data members using GLOBAL SHARED variables and other data elements, with the key difference that while PUBLIC data members are scoped to the class that defines them, once a variable or other data element is defined as GLOBAL SHARED it remains allocated for the duration of the session and cannot be deleted.

Overriding data within a class hierarchy

It is invalid to have a data member or property in a subclass with the same name as any PUBLIC or PROTECTED data member or property in one of its super classes, regardless of whether the types of the two data members or properties match or not. 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 to the method.

Overriding methods within a class hierarchy

A class definition (subclass) that inherits from another class can define a method to override a PUBLIC or PROTECTED method in its class hierarchy as long as it is not marked FINAL. The method must have the same method name, access mode, 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. However, a PUBLIC subclass method can override a PROTECTED super class method, if you want. 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.

Once a method has been overridden, all calls to that method invoke the method in the most derived subclass where it is defined. This is true whether the method is called from outside (using an object reference) or from inside the class hierarchy of the object.

Figure 3–2 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**).

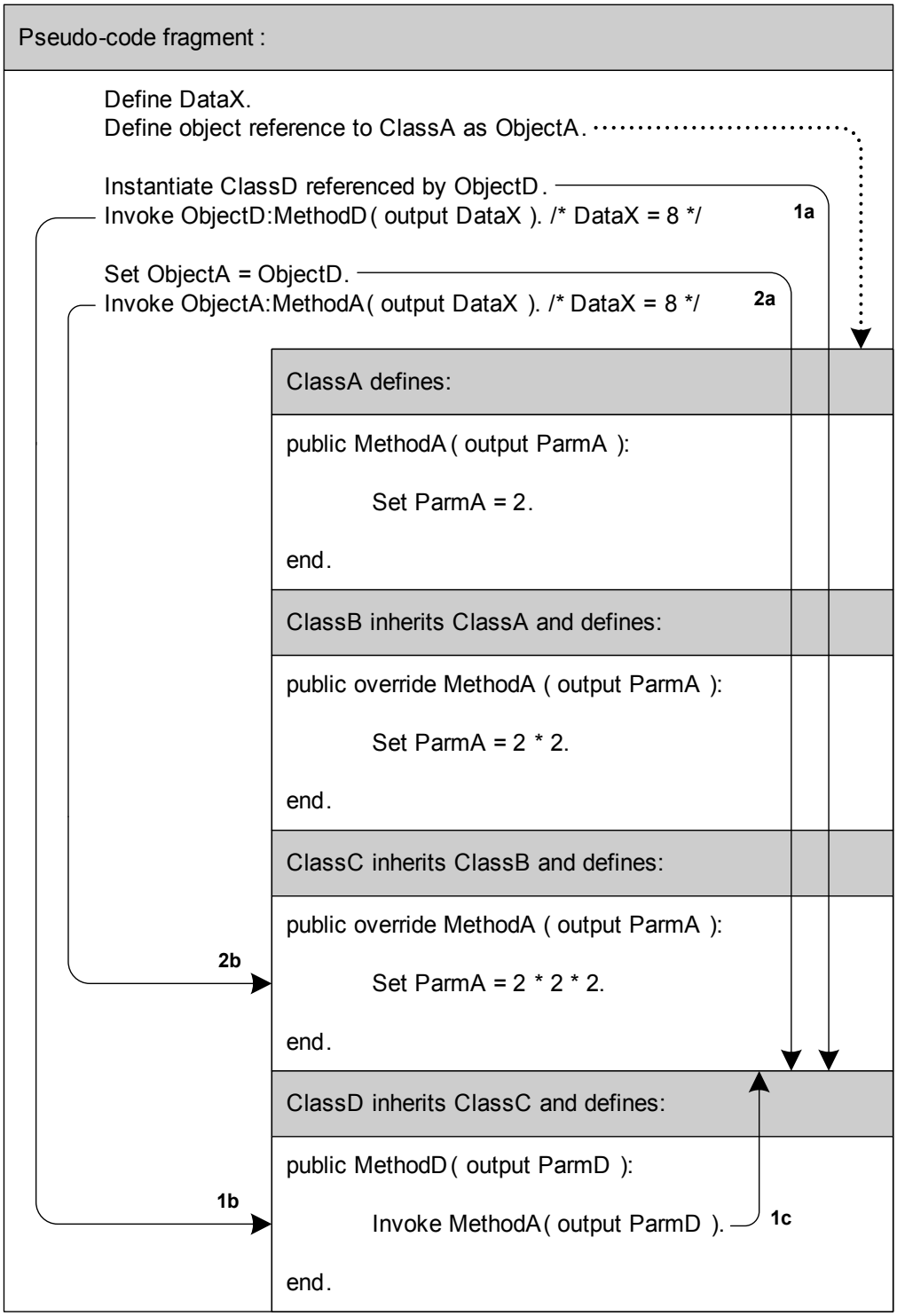

Figure 3–2: Invoking an overridden method in a class

Now, suppose ClassE inherits from the ClassD shown in Figure 3–2 and also overrides MethodA( ), as shown in Figure 3–3.

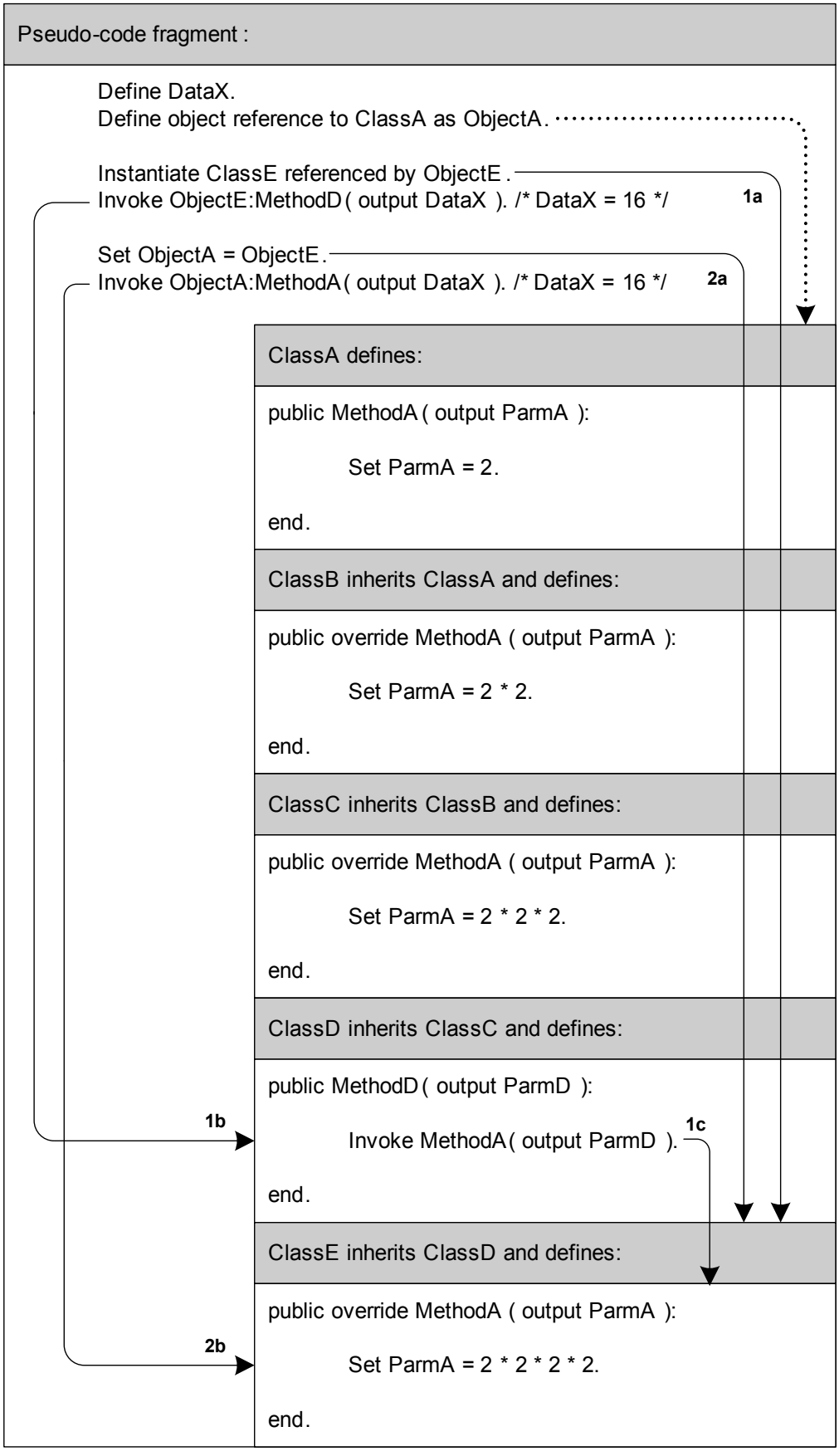

Figure 3–3: Invoking an overridden method in a class extension

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 in Figure 3–3, 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 the "Calling a super class method" section on page 3–21.

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 the "Sample classes" section on page 5–10). 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):
        DEFINE VARIABLE EmailName AS CHARACTER NO-UNDO.

        EmailName = SUPER:GetCustomerName (INPUT 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` (see the "Constructing an object" section on page 3–16). 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 an object referenced as a super class, 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 object assigned to 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 for 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 the "Using polymorphism with classes" section on page 3–22.

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 the "Overloading methods and constructors" section on page 3–12.

Overloading 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 of the number, type, or mode of their parameters. 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 that can instantiate that class, and a method of a given name typically invokes similar behavior to every other overloaded method of the same name. In fact, it does not matter what behavior each method overload 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: Multiple constructors of a class are, by definition, always overloaded, because different signatures provide the only means to distinguish multiple constructors of a given class definition, which always have the same name.

A class definition can overload its own methods and constructors. As a subclass, it can 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 overloads 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 polymorphicly. 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 with overriding. For more information on overriding, see the "Overriding methods within a class hierarchy" section on page 3–7.

Defining 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 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 scalar 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 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.

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 ttCust NO-UNDO
        FIELD ...

...

    /* first version of printObj prints a single copy of a report */
    METHOD PUBLIC VOID printObj ( ):
        OUTPUT TO PRINTER.
        DISPLAY timestamp. /* Inherited from CommonObj */
        FOR EACH ttCust:
            DISPLAY ttCust.
        END.
        OUTPUT CLOSE.
    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):
        DEFINE VARIABLE iCnt AS INTEGER.
        OUTPUT TO PRINTER.
        IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
            DISPLAY timestamp.
            FOR EACH ttCust:
                DISPLAY ttCust.
            END.
        END.
        OUTPUT CLOSE.
    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:

    DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
    ...
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    DEFINE PRIVATE VARIABLE outFile AS CHARACTER.

    /* first constructor instantiates a Customer object */
    CONSTRUCTOR PUBLIC Main ( ):
        /* Create an instance of the HelperClass class  */
        rHelperClass = NEW HelperClass ( ).

        /* Create an instance of the CustObj class      */
        rCustObj = NEW CustObj ( ).
        outFile = "Customers.out".
    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):
        /* Create an instance of the HelperClass class */
        rHelperClass = NEW HelperClass( ).

        /* Create an instance of the NECustObj class   */
        rCustObj = NEW NECustObj (INPUT EmailFile).
        outFile = "NECustomers.out".
    END CONSTRUCTOR.

...

END CLASS.
```

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

Invoking 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 the "Calling an overloaded method or constructor" section on page 4–12.

Constructing an object

You construct a class instance (object) by using a NEW phrase with the type name of the class and any parameters required by the specified constructor. (For more information on using the NEW phrase, see the "Creating a class instance using the NEW phrase" section on page 4–4.) As described previously (see the beginning of the "Class hierarchies and inheritance" section on page 3–2), 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.

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 phrase) 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 phrase automatically invokes the default constructor for the instantiated subclass when the NEW phrase 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 descriptions 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 *OpenEdge Development: ABL Reference*.

Notes: 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 phrase. For more information, see the "Raising and handling error conditions" section on page 4–46.

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 descriptions 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 *OpenEdge Development: ABL Reference*.

Notes: 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 phrase. For more information, see the "Raising and handling error conditions" section on page 4–46.

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 an overload of itself.

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 phrase 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 phrase 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 the "Overriding methods within a class hierarchy" section on page 3–7. 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):
        DEFINE VARIABLE EmailName AS CHARACTER NO-UNDO.

        EmailName = SUPER:GetCustomerName (INPUT 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 or PROTECTED. 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 phrase of 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.
```

Comparison with procedure-based programming

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.

Deleting an object

When you delete an object using the DELETE OBJECT statement, the destructors, if any, for every class in the hierarchy are automatically run. The destructors are 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.

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.

Calling a super class method

You can invoke the super class definition of a 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 override of that method, in order to 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 the "Constructing an object" section on page 3–16.

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 method invoked can be any method name valid in the class hierarchy. It does not need to match the method name from where the call is made.

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

The parameters of the method. For more information on the syntax of *parameter*, see the Parameter passing syntax reference entry in *OpenEdge Development: 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.

Using 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 (see the "Overriding methods within a class hierarchy" section on page 3–7), 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 phrase that instantiates the class to an object reference 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.

Figure 3–4 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**).

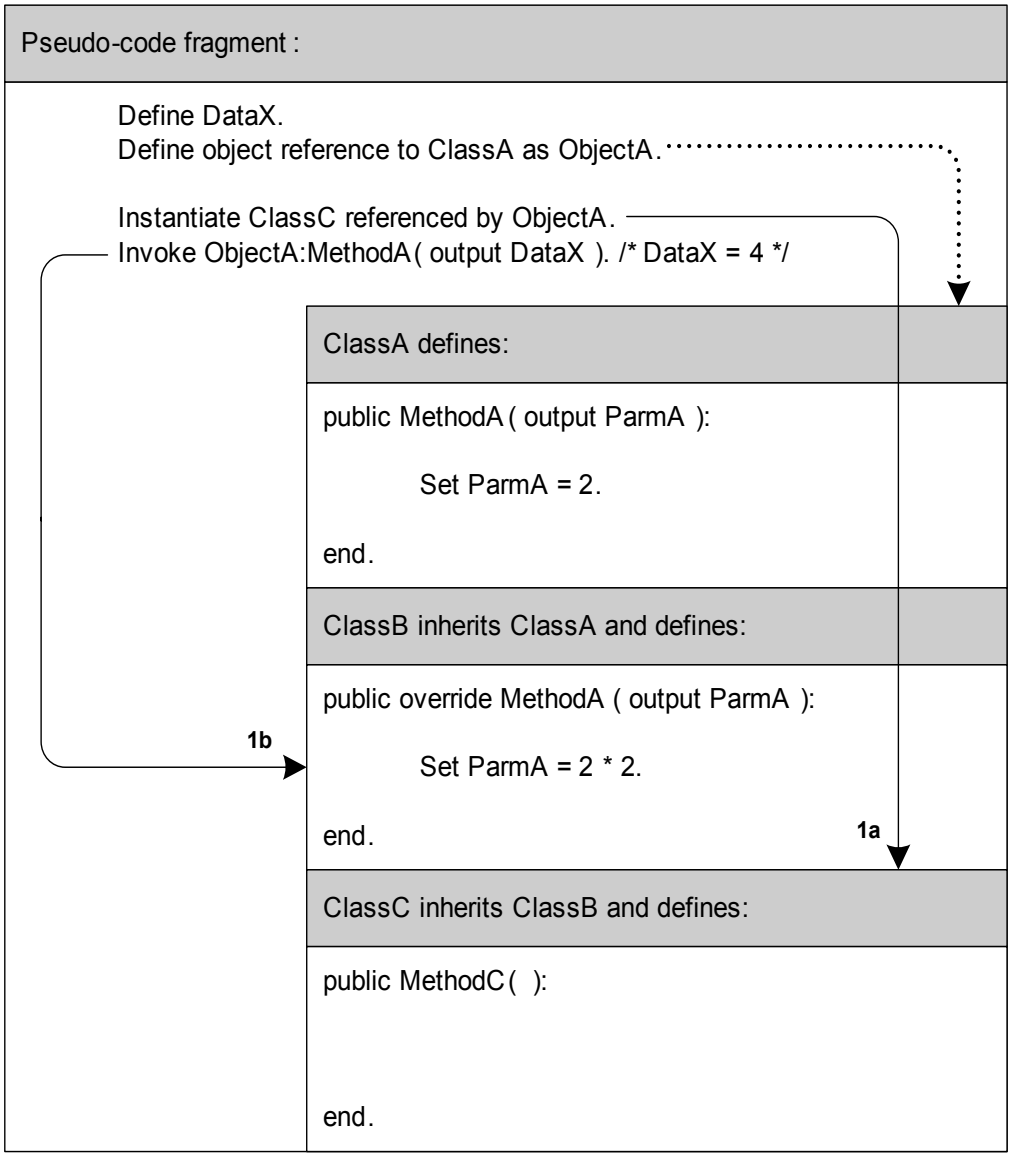

Figure 3–4: Invoking a method polymorphically (one subclass)

Figure 3–5 shows another polymorphic method call, similar to Figure 3–4. 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**).

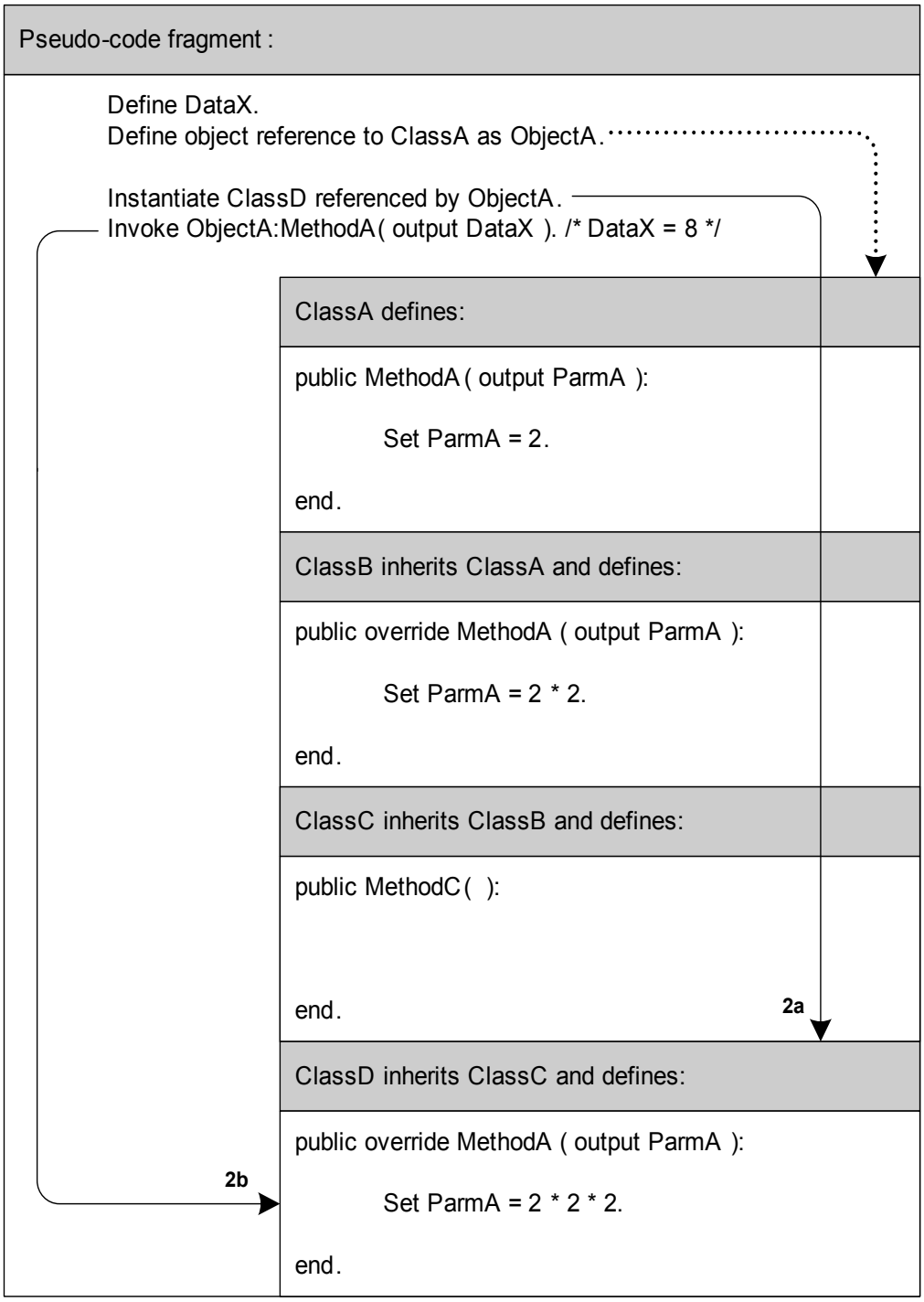

Figure 3–5: Invoking a method polymorphically (another subclass)

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:

    DEFINE PRIVATE VARIABLE length AS DECIMAL NO-UNDO.
    DEFINE PRIVATE VARIABLE width AS DECIMAL NO-UNDO.

    CONSTRUCTOR PUBLIC RectangleClass
                          (INPUT l AS DECIMAL, INPUT w AS DECIMAL):
        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:

    DEFINE PRIVATE VARIABLE radius AS DECIMAL NO-UNDO.
    DEFINE PRIVATE VARIABLE pi AS DECIMAL INITIAL 3.14159 NO-UNDO.

    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:

    DEFINE VARIABLE rRectangle AS CLASS RectangleClass NO-UNDO.
    DEFINE VARIABLE rCircle AS CLASS CircleClass NO-UNDO.
    DEFINE VARIABLE width AS DECIMAL NO-UNDO INITIAL 10.0.
    DEFINE VARIABLE length AS DECIMAL NO-UNDO INITIAL 5.0.
    DEFINE VARIABLE radius AS DECIMAL NO-UNDO INITIAL 100.0.

    CONSTRUCTOR PUBLIC Main ( ):

        rRectangle = NEW RectangleClass( INPUT width, INPUT length).
        rCircle = NEW CircleClass (INPUT radius).

        displayArea( INPUT rRectangle ).
        displayArea( INPUT 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 phrase 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 data 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.

Using 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 section 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` phrase. 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, and methods 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( )`. The natural mechanism to enforce this design is the use of an interface. The following code shows the example interface definition:

```
INTERFACE Ilog:

    METHOD PUBLIC VOID openLog (INPUT name AS CHARACTER).
    METHOD PUBLIC VOID writeLog (INPUT txt AS CHARACTER).
    METHOD PUBLIC VOID closeLog ( ).

END INTERFACE.
```

The container class itself implements 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 ilogMode AS INTEGER)
        logMode = ilogMode.
    END METHOD.

    METHOD PUBLIC VOID openLog (INPUT name AS CHARACTER):
        IF logMode EQ 1 THEN
            rlogToFile:openLog (INPUT name).
        ELSE
            rlogToDB:openLog (INPUT name).
    END METHOD.

    METHOD PUBLIC VOID writeLog (INPUT txt AS CHARACTER):
        IF logMode EQ 1 THEN
            rlogToFile:writeLog (INPUT txt).
        ELSE
            rlogToDB:writeLog (INPUT txt).
    END METHOD.

    METHOD PUBLIC VOID closeLog ( ):
        IF logMode EQ 1 THEN
            rlogToFile:closeLog ( ).
        ELSE
            rlogToDB:closeLog ( ).
    END METHOD.

    DESTRUCTOR PUBLIC container ( ):
        DELETE OBJECT rlogToDB NO-ERROR.
        rlogToDB = ?.
        DELETE OBJECT rlogToFile NO-ERROR.
        rlogToFile = ?.
    END METHOD.
END CLASS.
```

Comparison with procedure-based programming

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

Programming with Class-based Objects

Once an instance of a class is created, you will want to access the data members, properties, and methods of the object instance. The following sections describe how to instantiate class-based objects and manage them in an application:

- Instantiating and managing class-based objects
- Verifying the type and validity of an object reference
- Using built-in system and object references
- Assigning object references
- Comparing objects
- Defining and using widgets in classes
- Using preprocessor features in a class
- Raising and handling error conditions
- Reflection—Using the Progress.Lang.Class class

Instantiating and managing class-based objects

Each user-defined class or interface essentially represents a unique data type. ABL supports class and interface types in much the same way as built-in scalar data types (such as INTEGER), and manages class-based objects in a similar fashion to handle-based objects (such as procedure, buffer, or query objects). Thus, class or interface types can be used to define the data types for variables, properties, parameters, and return types used as object references. However, you cannot define an array of classes. The EXTENT keyword is not valid when defining a variable, parameter, return type, or temp-table field whose data type is a class type. (Properties never take an extent.)

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, or methods. Also, when you no longer need the class instance, you must explicitly delete it to prevent memory leaks.

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 an object reference to a class instance that you create. The particular data 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.

The following sections describe how to define and use object references:

- Defining an object reference as a variable or property
- Creating a class instance using the NEW phrase
- Calling class-based methods
- Calling an overloaded method or constructor
- Accessing data members and properties
- Defining an object reference as a parameter
- Passing object reference parameters
- Defining an object reference as a return type
- Defining an object reference as a field in a temp-table

Defining an object reference as a variable or property

This is the syntax to define an object reference as a variable:

Syntax

```
DEFINE [ access-mode ] VARIABLE object-reference
       AS [ CLASS ] type-name [ NO-UNDO ] .
```

This is the syntax to define an object reference as a property:

Syntax

```
DEFINE [ access-mode ] PROPERTY object-reference
       AS [ CLASS ] type-name [ NO-UNDO ] accessor-definitions .
```

Element descriptions for this syntax diagram briefly follow:

Note: For more information on each option, see the specified references in this book and also the DEFINE VARIABLE and DEFINE PROPERTY statement reference entries in *OpenEdge Development: ABL Reference*. For variables, no other options apply (for example, EXTENT), 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 the "Accessing data members and properties" section on page 4–16.

object-reference

The name of a variable or property that will hold an object reference value.

CLASS

The CLASS keyword is required if *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.

type-name

The type name of a class or interface type. This can be the fully qualified 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 type names, see the "Defining and referencing user-defined type names" section on page 2–3. For more information on the USING statement, see the "Referencing a type name without its package qualifier" section on page 2–6.

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 the "Defining data members within a class" section on page 2–15.
- Properties, see the "Defining properties within a class" section on page 2–19.

The following example shows a fragment of the `Main` class from the sample classes that are fully implemented in the "Sample classes" section on page 5–10. This fragment shows several private object references defined as variables:

```
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 rCustObj2 AS CLASS CustObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rCommonObj AS CLASS CommonObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rIBusObj AS CLASS IBusObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    DEFINE PRIVATE VARIABLE outFile AS CHARACTER.

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

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

    ...

    DESTRUCTOR PUBLIC Main( ):
        DELETE OBJECT rCustObj.
        rCustObj = ?.
        DELETE OBJECT rHelperClass.
        rHelperClass = ?.
    END DESTRUCTOR.

END CLASS.
```

Creating a class instance using the NEW phrase

The `NEW` phrase is used to create an instance of a class. At run time, the `NEW` phrase locates the named class on the `PROPATH` using the package information in the type name or in an appropriate `USING` statement, and invokes the specified constructor to create a new instance of the class. The instantiated class is also referred to as a class-based object. An Assignment (=) statement that contains the `NEW` phrase assigns a reference for the new class instance to an object reference variable.

This is the syntax for instantiating a class using the NEW phrase in an Assignment (=) statement:

Syntax

```
object-reference = NEW type-name ( [ parameter [ , parameter ] ... ] )
    [ NO-ERROR ] .
```

Element descriptions for this syntax diagram follow:

object-reference

The name of a variable, writable property, parameter, temp-table field or other writable ABL data element appropriately defined as a class or interface type.

This *object-reference* must be defined as one of the following types:

- The same class as *type-name*.
- A super class of *type-name*.
- An interface implemented by the class specified by *type-name*.

Note: If *object-reference* is a temp-table field, its 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 the "Using the root class—Progress.Lang.Object" section on page 2–35.

type-name

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 type names, see the "Defining and referencing user-defined type names" section on page 2–3. For more information on the USING statement, see the "Referencing a type name without its package qualifier" section on page 2–6.

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

The parameters, if any, passed to a PROTECTED or PUBLIC constructor defined by the *type-name* class. If the class defines more than one PROTECTED or PUBLIC constructor, the constructors are overloaded and these parameters identify the constructor to use for instantiating the class. For more information on the syntax of *parameter*, see the Parameter passing syntax reference entry in *OpenEdge Development: ABL Reference*. For more information on passing parameters to overloaded constructors, see the "Calling an overloaded method or constructor" section on page 4–12.

Whenever an instance of a class is created, 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 PROTECTED and PUBLIC data members and properties defined by all classes in it 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 the "Constructing an object" section on page 3–16.

This example, from the Main class described previously (see the "Defining an object reference as a variable or property" section on page 4–3), 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( ):
        /* Create an instance of the HelperClass class   */
        rHelperClass = NEW HelperClass( ).

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

    ...

END CLASS.
```

Calling class-based methods

How you call a class-based method depends on whether you are calling it from within the class hierarchy where it is defined or calling it with respect to an object that contains the method definition. 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.

With respect to another object, within whose class hierarchy a given 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:

Syntax

```
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 as part of an Assignment (=) statement. For more information on the syntax and the basic requirements for a *method-call*, see the "Calling methods from inside a class hierarchy" section on page 4–8 and the "Calling methods from outside a class hierarchy" section on page 4–9.

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 the "Passing object reference parameters" section on page 4–20.

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 to another. For more information, see the "Defining an object reference as a parameter" section on page 4–19.

- 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 reference entry in *OpenEdge Development: 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 methods, see the Class-based method call reference entry in *OpenEdge Development: ABL Reference*. For more information on calling overloaded methods, see the "Calling an overloaded method or constructor" section on page 4–12.

Calling methods from inside a class hierarchy

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 and PROTECTED methods implemented in any super class of the class hierarchy. Invoking an overridden method from within a super class definition executes the method defined in the most derived subclass within the class hierarchy of a given object. Thus, 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.

This is the syntax to invoke a method from inside a class:

Syntax

```
[ return-variable = ] 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 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 the Parameter passing syntax reference entry in *OpenEdge Development: ABL Reference*.

NO-ERROR

Optionally redirects error processing when the method is called as a statement.

The following is an example from two of the sample classes, where acme.myObjs.Common.CommonObj is a super class of the class acme.myObjs.CustObj:

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

CLASS acme.myObjs.Common.CommonObj:

    DEFINE PUBLIC VARIABLE timestamp AS DATETIME NO-UNDO.

    METHOD PUBLIC VOID updateTimestamp( ):
        timestamp = NOW.
    END METHOD.

    METHOD PROTECTED CLASS MsgObj
            MessageHandler (INPUT iObjType AS CHARACTER):
        DEFINE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
        rMsg = NEW MsgObj (INPUT iObjType).
        RETURN rMsg.
    END METHOD.

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 (INPUT "acme.myObjs.CustObj").
    END CONSTRUCTOR.

    ...

END CLASS.
```

Calling methods from outside a class hierarchy

You can access PUBLIC 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. 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 the Parameter passing syntax reference entry in *OpenEdge Development: 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 rMsg AS CLASS MsgObj NO-UNDO.
    ...

    CONSTRUCTOR PUBLIC CustObj( ):
        ...
        rMsg = MessageHandler (INPUT "acme.myObjs.CustObj").
    END CONSTRUCTOR.

    ...

    METHOD PUBLIC VOID CheckCredit ( ):
        IF VALID-OBJECT (rCreditObj) THEN DO:
            FOR EACH ttCust:
                rCreditObj:SetCurrentCustomer (INPUT ttCust.CustNum).
                rCreditObj:CheckCustCredit ( ) NO-ERROR.
                IF ERROR-STATUS:ERROR THEN
                    IF RETURN-VALUE = "Over Limit" THEN DO:
                     /* invokes the CustCreditLimit property GET accessor */
                        rMsg:Alert (INPUT ttCust.Name +
                                " is on Credit Hold." +
                                "  Balance exceeds Credit Limit of " +
                                STRING (rCreditObj:CustCreditLimit)).
                    END.
                    ELSE
                        rMsg:Alert (INPUT "Customer not found").
                ELSE DO:
                    /* invokes the CustCreditLimit property GET accessor */
                    rMsg:InfoMsg (INPUT ttCust.Name +
                                " is in good standing." +
                                "  Credit Limit has been increased to " +
                                STRING(rCreditObj:CustCreditLimit)).
                END.
            END. /* FOR EACH */
        END.
        ELSE rMsg:Alert (INPUT "Unable to check credit").
    END METHOD.

    ...

END CLASS.
```

In this case, the `rMsg` object reference is a private class variable 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 the "Calling methods from inside a class hierarchy" section on page 4–8). 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 ObjType AS CHARACTER NO-UNDO.

    CONSTRUCTOR PUBLIC MsgObj (INPUT rObjType AS CHARACTER):
        ObjType = rObjType.
    END CONSTRUCTOR.

    METHOD PUBLIC VOID Alert (INPUT ErrorString AS CHARACTER):
        MESSAGE "Error in " ObjType "!" 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.
```

Comparison with procedure-based programming

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.

Calling 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.

The following fragment of the `acme.myObjs.CustObj` sample class defines two overloads 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 ttCust NO-UNDO
        FIELD ...

...

    /* first version of printObj prints a single copy of a report */
    METHOD PUBLIC VOID printObj ( ):
        OUTPUT TO PRINTER.
        DISPLAY timestamp. /* Inherited from CommonObj */
        FOR EACH ttCust:
            DISPLAY ttCust.
        END.
        OUTPUT CLOSE.
    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):
        DEFINE VARIABLE iCnt AS INTEGER.
        OUTPUT TO PRINTER.
        IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
            DISPLAY timestamp.
            FOR EACH ttCust:
                DISPLAY ttCust.
            END.
        END.
        OUTPUT CLOSE.
    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:

    DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
    ...

    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 ( ):
        ...
        /* INPUT: it is valid to pass a subclass to a method         */
        /*        defined to take a super class                      */
        rHelperClass:InitializeDate (INPUT rCustObj).
        IF rCustObj:iNumCusts > 20 THEN
            rCustObj:printObj( ).
        ELSE
            rCustObj:printObj(INPUT 3).

        ...

    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:

    DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
    ...
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    DEFINE PRIVATE VARIABLE outFile AS CHARACTER.

    /* first constructor instantiates a Customer object */
    CONSTRUCTOR PUBLIC Main ( ):
        /* Create an instance of the HelperClass class  */
        rHelperClass = NEW HelperClass ( ).

        /* Create an instance of the CustObj class      */
        rCustObj = NEW CustObj ( ).
        outFile = "Customers.out".
    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):
        /* Create an instance of the HelperClass class */
        rHelperClass = NEW HelperClass( ).

        /* Create an instance of the NECustObj class   */
        rCustObj = NEW NECustObj (INPUT EmailFile).
        outFile = "NECustomers.out".
    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 **/

DEFINE VARIABLE ClassExample AS CLASS Main.

/* run the example for all Customers         */
ClassExample = NEW Main ( ).
ClassExample:ObjectInfo ( ).
DELETE OBJECT ClassExample.
ClassExample = ?.

/* run the example for New England Customers */
classExample = NEW Main (INPUT "email.txt").
ClassExample:ObjectInfo ( ).
DELETE OBJECT ClassExample.
ClassExample = ?.
```

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 the "Defining overloaded methods and constructors" section on page 3–13. For more information on finer points of method and constructor overloading and how ABL handles some of the more complex overloading scenarios, see Appendix A, "Overloaded Method and Constructor Calling Scenarios."

Accessing 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` and `PROTECTED` data members or properties defined in any super class of the class hierarchy. For properties, access also depends on whether they are defined as readable or writable, or both. For more information, see the "Defining properties within a class" section on page 2–19.

You can access PUBLIC data members or properties from outside the class hierarchy of an object where they are defined by using an object reference to qualify the data member or property name. There is no direct access to PRIVATE or PROTECTED data members or properties from outside the class hierarchy. Also, note that only variable data members and properties can be PUBLIC. Therefore, 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.

Note: While you can also define non-PUBLIC properties, a primary benefit of properties is to provide the object interface to (the encapsulation for) 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.

This is the syntax for referencing a PUBLIC 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 *OpenEdge Development: 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 rMsg AS CLASS MsgObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rCreditObj AS CLASS CreditObj NO-UNDO.

    CONSTRUCTOR PUBLIC CustObj ( ):
        rCreditObj = NEW CreditObj ( ).
    ...
        rMsg = MessageHandler (INPUT "acme.myObjs.CustObj").
    END CONSTRUCTOR.

    ...

    METHOD PUBLIC VOID CheckCredit ( ):
        IF VALID-OBJECT (rCreditObj) THEN DO:
            FOR EACH ttCust:
                ...
                IF ERROR-STATUS:ERROR THEN
                    IF RETURN-VALUE = "Over Limit" THEN DO:
                     /* invokes the CustCreditLimit property GET accessor */
                        rMsg:Alert (INPUT ttCust.Name +
                                " is on Credit Hold." +
                                "  Balance exceeds Credit Limit of " +
                                STRING (rCreditObj:CustCreditLimit)).
                    END.
                    ELSE
                        rMsg:Alert (INPUT "Customer not found").
                ELSE DO:
                    /* invokes the CustCreditLimit property GET accessor */
                    rMsg:InfoMsg (INPUT ttCust.Name +
                                " is in good standing." +
                                "  Credit Limit has been increased to " +
                                STRING(rCreditObj:CustCreditLimit)).
                END.
            END. /* FOR EACH */
        END.
        ELSE rMsg:Alert (INPUT "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 the "Defining properties within a class" section on page 2–19.

Defining an object reference as a parameter

Procedures, user-defined functions, and class-based methods can all define class and interface types as parameters. Class instances are always passed by reference using object references. Thus, only a reference to an object is passed, not the object itself. An object reference parameter cannot be passed to or from a remote application server (neither as a value nor as a field in a temp-table).

This is the syntax to define an object reference as a parameter for a method in a class or a user-defined function in a procedure:

Syntax

```
[ INPUT | INPUT-OUTPUT | OUTPUT ] parameter-name AS [ CLASS ] type-name
```

This is the syntax to define an object reference as a parameter for a procedure:

Syntax

```
DEFINE [ INPUT | INPUT-OUTPUT | OUTPUT ] PARAMETER parameter-name
    AS [ CLASS ] 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 *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.

type-name

The type name of a class or interface type to be passed. This can be the fully qualified 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 type names, see the "Defining and referencing user-defined type names" section on page 2–3. For more information on the USING statement, see "Referencing a type name without its package qualifier" section on page 2–6.

For more information on the full syntax for parameter definitions, see the DEFINE PARAMETER statement and Parameter definition syntax reference entries in *OpenEdge Development: ABL Reference*.

Passing object reference parameters

When an application passes an object reference to a method, user-defined function, or procedure, the effect is identical to assigning one object reference to another, with regard to the parameter mode (`INPUT`, `INPUT-OUTPUT`, or `OUTPUT`). Thus, 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:

1. The same type as the source (for example, to `acme.myObjs.CustObj` from `acme.myObjs.CustObj`).
2. A super class of the source (for example, to `acme.myObjs.Common.CommonObj` from `acme.myObjs.CustObj`).
3. 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 the "Assigning object references" section on page 4–34.

Following are a series of examples that demonstrate the passing of object reference parameters for `INPUT`, `INPUT-OUTPUT`, and `OUTPUT`. These examples highlight code from the `Main` class shown previously (see the "Defining an object reference as a variable or property" section on page 4–3) and related sample classes that are described and fully listed in the "Sample classes" section on page 5–10. 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 the "Assignment and the CAST function" section on page 4–37.

The following examples pass an INPUT parameter:

```
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.
    ...
    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 (INPUT rCustObj).
        ...

    END METHOD.

    ...

END CLASS.
```

```
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 (INPUT "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 the "Calling methods from inside a class hierarchy" section on page 4–8.) This method can be invoked on any object that is a subclass of CommonObj. For more information on acme.myObjs.Common.CommonObj, see the sample classes in the "Sample classes" section on page 5–10.

When you pass an object reference 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:

    DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
    ...
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    ...
    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:

    DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
    ...

    METHOD PUBLIC VOID ListNames (INPUT-OUTPUT prCustObj AS CLASS CustObj):
        DEFINE VARIABLE idx AS INTEGER NO-UNDO.

        DO idx = 1 to prCustObj:iNumCusts:
            CREATE ttNames.
            ttNames.CustName = prCustObj:GetCustomerName (INPUT 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 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 the "Assignment and the CAST function" section on page 4–37.

The following examples pass an OUTPUT parameter:

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

CLASS Main:

    ...
    DEFINE PRIVATE VARIABLE rIBusObj AS CLASS IBusObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    DEFINE PRIVATE VARIABLE outFile AS CHARACTER.
    ...

    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).
        rIBusObj:logObj (INPUT outFile).
        ...

    END METHOD.

    ...

END CLASS.
```

```
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.
    ...

    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.

Defining an object reference as a return type

User-defined functions and class methods can define class and interface types as return types. Class instances are always returned by reference. Thus, only a reference to an object is returned, not the object itself. An object reference cannot be returned to or from a remote application server.

This is the syntax to define an object reference as a return type for a method:

Syntax

```
METHOD { PUBLIC | PROTECTED | PRIVATE } [ OVERRIDE ] [ FINAL ] [ CLASS ]
    type-name method-name ( [ parameter [ , parameter ] ... ] ) :

    [ method-body ]
```

This is the syntax to define an object reference as a return type for a user-defined function:

Syntax

```
FUNCTION function-name RETURNS [ CLASS ] type-name
    ( [ parameter [ , parameter ] ... ] ) :

    [ function-body ]
```

Descriptions of the object reference return type syntax elements (bolded) follow:

CLASS

The CLASS keyword is required if 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.

type-name

The type name of the class or interface type reference to be returned. This can be the fully qualified 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 type names, see the "Defining and referencing user-defined type names" section on page 2–3. For more information on the USING statement, see the "Referencing a type name without its package qualifier" section on page 2–6.

For more information on the syntax of these statements, see the METHOD statement and FUNCTION statement reference entries in *OpenEdge Development: 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 WHERE CreditLimit > 50000:
            CREATE ttCust.
            ASSIGN
                iNumCusts = iNumCusts + 1
                ttCust.RecNum = iNumCusts
                ttCust.CustNum = Customer.CustNum
                ttCust.Name = Customer.Name
                ttCust.State = Customer.State.
        END.
        rMsg = MessageHandler (INPUT "acme.myObjs.CustObj").
    END CONSTRUCTOR.

    ...

END CLASS.
```

The MessageHandler( ) method definition is defined in the immediate super class. This method instantiates the MsgObj class and returns the object to the caller, as shown:

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

CLASS acme.myObjs.Common.CommonObj:

    ...

    METHOD PROTECTED CLASS MsgObj
            MessageHandler (INPUT iObjType AS CHARACTER):
        DEFINE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
        rMsg = NEW MsgObj (INPUT iObjType).
        RETURN rMsg.
    END METHOD.

END CLASS.
```

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 the "Assigning object references" section on page 4–34.

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 the "Defining an object reference as a parameter" section on page 4–19). The caller of the method must define an object reference that is the same class as the return object, 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 variable an object reference to 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.

Defining an object reference as a 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 user-defined classes. This allows a temp-table field to reference classes of any 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 class type, you must cast the object reference to the specific type. For more information on casting, see the "Assignment and the CAST function" section on page 4–37.

As with class type parameters, a temp-table containing a class field cannot be passed to or from a remote application server.

This is the syntax to define a class field in a temp-table:

Syntax

```
FIELD field-name AS [ CLASS ] Progress.Lang.Object
```

Element descriptions 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:

    DEFINE TEMP-TABLE myTT NO-UNDO
            FIELD Shape AS CLASS Object
            FIELD Area AS DECIMAL.
    DEFINE VARIABLE width AS DECIMAL NO-UNDO INITIAL 10.0.
    DEFINE VARIABLE length AS DECIMAL NO-UNDO INITIAL 5.0.
    DEFINE VARIABLE radius AS DECIMAL NO-UNDO INITIAL 100.0.

    CONSTRUCTOR PUBLIC Main ( ):

        CREATE myTT.
        myTT.Shape = NEW RectangleClass (INPUT width, INPUT length).

        CREATE myTT.
        myTT.Shape = NEW CircleClass (INPUT radius).

        FOR EACH myTT:
            /* Cast the field to the common shape super class, ShapeClass */
            displayArea( INPUT CAST(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 in a previous chapter (see the "Using polymorphism with classes" section on page 3–22). 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`).

Verifying 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 descriptions 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.

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 the "Comparing constructs in classes and procedures" section on page 5–9. However, in a more complex application, the validity of object references at any given point might well be far from certain.)

For example, as shown in the following HelperClass fragment, the ReportOutput( ) method verifies that the object reference, 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 (INPUT "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 (INPUT "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, type-name)
```

Element descriptions for this syntax diagram follow:

object-reference

An object reference defined as any class or interface type.

type-name

The type name of a class or interface type. This can be the fully qualified 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 type names, see the "Defining and referencing user-defined type names" section on page 2–3. For more information on the USING statement, see the "Referencing a type name without its package qualifier" section on page 2–6.

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 *type-name*.
2. Is a subclass of the class type specified by *type-name*.
3. Implements the interface type specified by *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 "VALID-OBJECT function" section on page 4–29), the TYPE-OF function returns the Unknown value (?).

Using built-in system and object references

ABL provides built-in system and object references 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* (such as the FIRST-OBJECT attribute on the SESSION system handle) is a value set by the AVM that returns a user-defined class-based object that has been instantiated within the ABL session.

The values of different built-in system and object references are specified by unique keywords in ABL. The following sections describe these references.

THIS-OBJECT system reference

THIS-OBJECT is a system reference available from within a class definition. At run time, it returns the currently running instance of the class as an object reference. Its most important purpose is to allow a method of the class to pass a reference to the currently instantiated object as a parameter or to return a reference to itself as a method return value.

Note: 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. For more information, see the "Constructing an object" section on page 3–16.

The following example shows a class, acme.myObjs.NewCustomer, that inherits from the sample class acme.myObjs.CustObj. 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:

```
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(INPUT THIS-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, as previously:

```
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 (INPUT "Not a valid object").
    END METHOD.

    ...

END CLASS.
```

SUPER system reference

SUPER is a system reference available at run time within the current instance of a subclass to invoke a method implemented in a super class. A common use for this system reference is to invoke a super class method that the subclass has overridden. For more information on the syntax and use of this system reference, see the "Calling a super class method" section on page 3–21.

The only way a super class's version of an overridden method can be invoked is by referencing it with the SUPER system reference and method name within a subclass. 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.

Notes: 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 the "Constructing an object" section on page 3–16.

ABL session object references

The FIRST-OBJECT and LAST-OBJECT attributes on the SESSION system handle provide access to the list of currently instantiated objects. FIRST-OBJECT and LAST-OBJECT are both 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 used in the NEW phrase), you need to CAST the object to the required type. For more information on casting, see the "Assignment and the CAST function" section on page 4–37.

For example, you can display the list of class type names for all classes currently instantiated in the ABL session with the following code:

```
DEFINE VARIABLE myObj AS CLASS Progress.Lang.Object NO-UNDO.
myObj = SESSION:FIRST-OBJECT.
REPEAT:

    IF myObj EQ ? THEN
        LEAVE.

    MESSAGE myObj:GetClass( ):TypeName VIEW-AS ALERT-BOX.
    myObj = myObj:NEXT-SIBLING.
END.
```

Assigning object references

You can freely assign one object reference to another object reference if they are defined with compatible data types. The object references in an assignment have compatible data types if one of the following is true:

- Both references 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 is defined as a super class of the right-hand reference (`acme.myObjs.Common.CommonObj` = `acme.myObjs.CustObj`).
- The left-hand reference is defined as an interface implemented by the right-hand reference (`acme.myObjs.Interfaces.IBusObj` = `acme.myObjs.CustObj`).

This assignment always sets the left-hand object reference 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.

Also, if the left-hand object reference already has a reference to an object, and there is no other available reference to that object, you must make sure to delete that object before reassigning the reference. Otherwise, you have created a memory leak—the previous reference is overwritten with the new value and the previous object continues to exist with no object references available to delete it. This is very similar to the precautions you must take to avoid memory leaks while maintaining persistent procedures and other types of dynamic handle-based objects.

When you assign a subclass object reference to a super class object reference, the new object reference still points to the original object instance, but 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 an interface object reference to any class object reference, where the class implements the specified interface type at any level in its class hierarchy. If you use the interface object reference, you can only access the methods defined in the interface.

So an object reference can be copied to an object reference of the same type, a super type, or an interface that the object implements. The converse of this rule is **not** true. An object reference to a super class cannot be copied to a subclass object reference unless you use the `CAST` function. The compiler rejects this assignment and only allows it if an appropriate `CAST` function is provided on the right-hand side (see the "Assignment and the CAST function" section on page 4–37).

The following variation on the sample `Main` class demonstrates compatible assignment between different types of object references 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 that implements that 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 rCustObj AS CLASS CustObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rCustObj2 AS CLASS CustObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rCommonObj AS CLASS CommonObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rIBusObj AS CLASS IBusObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    DEFINE PRIVATE VARIABLE outFile AS CHARACTER.

    /* first constructor instantiates a Customer object */
    CONSTRUCTOR PUBLIC Main ( ):
        /* Create an instance of the HelperClass class  */
        rHelperClass = NEW HelperClass ( ).

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

    ...

    /* ObjectInfo processes information about the Customer object    */
    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 (INPUT rCustObj).
/*#2*/  rCommonObj = rCustObj.

        /* 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 ( ).
/*#1*/  rCustObj2 = rCustObj.

        /* OUTPUT: an interface is used to receive a class that      */
        /*         implements that interface                         */
        rHelperClass:ReportOutput (OUTPUT rIBusObj).
        rIBusObj:logObj (INPUT outFile).
/*#3*/  rIBusObj = rCustObj.
    END METHOD.

    DESTRUCTOR PUBLIC Main ( ):
        DELETE OBJECT rCustObj.
        rCustObj = ?.
        ...
    END DESTRUCTOR.

END CLASS.
```

Note that all of these object references, in fact, point to the same instance of one object. Thus, the destructor for Main invokes only one DELETE OBJECT statement for the acme.myObjs.CustObj instance. When the destructor executes, this statement deletes the CustObj instance of that all four object references point to, and all four variables become invalid object references.

Assignment and the CAST function

As described in the previous section, you can assign a source object reference to a target object reference if the target object reference is defined as the same class, a super class, or an interface of the source object. When the target object reference is 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, or method 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 the "Using polymorphism with classes" section on page 3–22.)

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 instance to an object reference for a given interface. When you invoke any method on this 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 reference, or from an implementing-class reference to an interface reference, 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` object reference. You can also directly assign an `acme.myObjs.CustObj` object reference to an `acme.myObjs.Common.CommonObj` or `Progress.Lang.Object` object reference.

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 an object reference defined for 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. Therefore, you must cast an object reference downward explicitly using the `CAST` function. In addition, you cannot assign an object reference for an interface to an object reference for a class, even if the class implements the interface. To do this you must also use the `CAST` function.

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 or that an interface object reference is really an object reference to an interface-implementing class.

Casting tells the compiler to trust your judgement and allow the downward cast within the specified class hierarchy. However, the AVM checks at run time to ensure that a cast is valid. For example, if a super class object reference is not referring to an instance of the specified subclass, the AVM returns a run-time error. This means that when you use the `CAST` function, you are bypassing some 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 your increased responsibility for making sure that all types are valid at run time.

You can thus use the `CAST` function wherever an object reference is permitted:

- To assign an object reference to another, especially when assigning a super class or interface reference to a related subclass or interface-implementing class reference, respectively.
- To pass an `INPUT` parameter to a method as a subclass of the defined parameter class type or as an implementing class of the defined parameter interface type.
- To invoke a method on a subclass of a specified class 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 invoke the method.

Syntax descriptions for each of these uses of the `CAST` function follow.

This is the syntax for the CAST function used to assign one object reference to another:

Syntax

```
target-object-reference = CAST ( object-reference , target-type-name )
```

This is the syntax for the CAST function used to pass an INPUT object reference parameter:

Syntax

```
routine-name ( [ INPUT ] CAST ( object-reference , target-type-name ) )
```

This is the syntax for the CAST function used to cast an object reference to access a method, data member, or property:

Syntax

```
CAST ( object-reference , target-type-name ) : member-name [ ( ... ) ]
```

Element descriptions for these syntax diagrams follow:

target-object-reference

The destination object reference returned by the CAST function, typically defined to reference one of the following types:

- A subclass of a class whose type defines the *object-reference*.
- A class that implements an interface whose type defines the *object-reference*.

object-reference

A source object reference, whose type is typically defined as a super class of class type or an interface implemented by a class type specified by *target-object-reference*. 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 refers 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 be for 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*.

An interface type name must be for 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 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 type names, see the "Defining and referencing user-defined type names" section on page 2–3. For more information on the USING statement, see the "Referencing a type name without its package qualifier" section on page 2–6.

routine-name

The name of any procedure, user-defined function, or method that takes an INPUT object reference as the source parameter. The corresponding destination parameter defined by routine-name must have the data type defined for *target-object-reference*.

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 whose *target-object-reference* is returned by the cast. 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 whose *target-object-reference* is returned by the cast. 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 whose *target-object-reference* is returned by the cast. 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.

The following sections describe examples of these uses for casting.

Casting an object reference assignment

For an example of assigning an object reference using the CAST function, see the "Defining an object reference as a field in a temp-table" section on page 4–27.

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 (INPUT 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`.

Comparing 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 (?).

The following example demonstrates the use of the equal operator using the two object references, `rSuper` and `rSubClass`:

```
DEFINE VARIABLE rSubClass AS CLASS SubClass NO-UNDO.
DEFINE VARIABLE rSuper    AS CLASS SuperClass NO-UNDO.

rSubClass = NEW SubClass ( ).
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 are defined as different classes in the class hierarchy.

Defining and using 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 comments (**#1** and **#2**) contain executable code:

```
CLASS Driver:

    DEFINE PRIVATE BUTTON msg.
    DEFINE PRIVATE BUTTON done.
    DEFINE FRAME f msg done.
    DEFINE VARIABLE bGraphical AS LOGICAL NO-UNDO.

/* #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.
```

Using preprocessor features in a class

ABL supports preprocessor features in class definition files, with some restrictions. The following sections describe this support.

Using 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.
```

Using preprocessor names and directives

All built-in preprocessor names are supported in classes. These names include `BATCH-MODE`, `FILE-NAME`, `LINE-NUMBER`, `OPSYS`, `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.

Raising and handling error conditions

Methods, property accessors, and constructors all allow you to raise application error conditions using the RETURN statement with the ERROR option. This includes use of the RETURN ERROR option of any ON ERROR phrase specified for a subblock. However, you cannot use the RETURN ERROR mechanism to raise an application error from within a destructor.

For all supported method types, RETURN ERROR functions as it does in a procedure by raising ERROR in the caller. If the error condition is handled by the calling statement using the NO-ERROR option, the AVM sets the ERROR attribute on the ERROR-STATUS system handle to TRUE, and any optional error return string (provided as part of the RETURN ERROR statement) is available following completion of the calling statement using the RETURN-VALUE function. If the error condition is not handled by the calling statement (with NO-ERROR not specified), execution proceeds according to the specified or default ON ERROR phrase for the enclosing block. For more information, see the ON ERROR phrase reference entry in *OpenEdge Development: ABL Reference*.

Again, in all method types, when you invoke RETURN ERROR, persistent data defined with the NO-UNDO option that are set by the method prior to raising ERROR maintain their most recent settings after ERROR is raised. Any changed data not defined with NO-UNDO roll back to their values prior to being changed by the method.

Additional effects of RETURN ERROR depend on the type of method (named method, property accessor, or constructor) where the error is raised.

Errors within a method

When a method raises an error using RETURN ERROR, the effects depend on the context of the method call. In general, the results of raising error within both VOID and non-VOID methods is similar, with certain exceptions described in the following sections.

Non-VOID Methods that raise ERROR in expressions

The behavior is essentially identical to any other error in an expression, as might appear on the right side of an Assignment (=) statement. 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.

Methods that raise ERROR with NO-ERROR not specified

Some statements, such as the IF statement, do not allow the NO-ERROR option and you might not specify NO-ERROR for others. So, if a non-VOID method raises ERROR in an expression within a statement that does not have NO-ERROR specified, the existing value of any receiving data element for the expression remains unchanged, and the ERROR is raised on the block enclosing the statement that contains the expression. The same is true for a VOID method executed, for example, as part of an IF statement. For any statement that does not allow NO-ERROR to be specified, you can handle the error by enclosing the statement in a block of its own including a specified ON ERROR phrase.

Non-VOID Methods that raise ERROR as parameters

If a method is passed as a parameter, and that method raises ERROR, any statement in which the method parameter is passed also raises ERROR, with one exception. If the errant method parameter is passed to a user-defined function within a statement, that statement will not raise ERROR, because user-defined functions do not raise ERROR. In any case, the method, procedure, or user-defined function to which the errant method parameter is passed does not run. For all statements that do raise ERROR, the AVM returns an error message about attempting to push run-time parameters onto the stack.

Method error handling example

The following class fragments from the sample classes show an example of method error handling. The CheckCustCredit( ) method of acme.myObjs.CreditObj raises ERROR for two different conditions, which are distinguished by the error return string specified for RETURN ERROR:

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

CLASS acme.myObjs.CreditObj INHERITS CommonObj:

    ...

    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 DO:
            CustCreditLimit = Customer.Creditlimit NO-ERROR.
            IF ERROR-STATUS:ERROR THEN DO:
                /* Indicates Customer balance is greater than credit limit */
                /* and returns error string from property SET              */
                RETURN ERROR RETURN-VALUE. /* RETURN-VALUE = "Over Limit" */
            END.
        END.
        ELSE
            RETURN ERROR “No Customer”.
    END METHOD.

END CLASS.
```

In this case, to make a Customer record available, the SetCurrentCustomer( ) method must be successfully invoked prior to invoking CheckCustCredit( ) to check and update the credit limit of a customer.

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 RETURN-VALUE function to handle the error condition appropriately:

```
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 ttCust 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.

    ...

    METHOD PUBLIC VOID CheckCredit ( ):
        IF VALID-OBJECT (rCreditObj) THEN DO:
            FOR EACH ttCust:
                rCreditObj:SetCurrentCustomer (INPUT ttCust.CustNum).
                rCreditObj:CheckCustCredit ( ) NO-ERROR.
                IF ERROR-STATUS:ERROR THEN
                    IF RETURN-VALUE = "Over Limit" THEN DO:
                     /* invokes the CustCreditLimit property GET accessor */
                        rMsg:Alert (INPUT ttCust.Name +
                                " is on Credit Hold." +
                                "  Balance exceeds Credit Limit of " +
                                STRING (rCreditObj:CustCreditLimit)).
                    END.
                    ELSE
                        rMsg:Alert (INPUT "Customer not found").
                ELSE DO:
                    /* invokes the CustCreditLimit property GET accessor */
                    rMsg:InfoMsg (INPUT ttCust.Name +
                                " is in good standing." +
                                "  Credit Limit has been increased to " +
                                STRING(rCreditObj:CustCreditLimit)).
                END.
            END. /* FOR EACH */
        END.
        ELSE rMsg:Alert (INPUT "Unable to check credit").
    END METHOD.

    ...

END CLASS.
```

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 that returns a value and a SET accessor corresponds to a method that does not (is VOID). So, when you read or write a property in a statement, the property accessor can raise ERROR on that statement using RETURN ERROR, and you can handle the error as provided by that statement. If the property accessor invokes RETURN ERROR with an error return string, you can access that string following the statement in which the property is accessed using the RETURN-VALUE function.

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.

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.

GET accessors raising ERROR

A GET accessor runs when the property appears in an expression or is passed as an INPUT parameter. You can handle errors that it raises similar to methods that appear in expressions on the right side of an Assignment (=) statement or that are passed as INPUT parameters.

When a method or property GET accessor on the right side of an assignment raises ERROR, the data element on the left side of the assignment remains unchanged from its value prior to executing the statement. This occurs because the data element on the left has nothing to do with the method or property error processing on the right, and the Assignment (=) statement itself simply ignores any results on the right side, leaving the left-side value unchanged.

In a similar fashion, when you pass a property or method as an INPUT parameter, if the property GET accessor or method raises ERROR, the routine where the parameter is passed never executes, leaving all data that it affects unchanged from their values prior to executing the routine. If it is not a user-defined function, the routine also raises ERROR in the caller.

Again, as with any method that raises ERROR, a GET accessor that raises ERROR leaves any data that it changes internally at their most recent settings prior to raising ERROR, depending on the UNDO processing coded for the accessor. Thus, you must code any GET accessor to retain or roll back data values, including the property default memory value, in the event that the accessor raises ERROR.

SET accessors raising ERROR

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 result that differs from a property or method raising ERROR on the right side of the assignment.

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, depending on NO-UNDO settings, the property default memory, and any other data that the SET accessor changes, retains its latest value prior to raising ERROR. Again, 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 an OUTPUT parameter, if the property SET accessor raises ERROR and depending on NO-UNDO settings, both the routine where the parameter is passed and the SET accessor leaves all data that it affects at their latest values prior to raising ERROR. If it is not a user-defined function, the routine also raises ERROR in the caller.

Again, as with any method that raises ERROR, a SET accessor that raises ERROR leaves any data that it changes internally at their most recent settings prior to raising ERROR, depending on the UNDO processing coded for the accessor. Thus, you must code any SET accessor to retain or roll back data values, including the property default memory value, in the event that the accessor raises ERROR.

Property error handling example

The following code fragment from the sample class, `acme.myObjs.CreditObj`, shows an example of property error handling. 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` if the customer's balance is over their current credit limit, setting the property value to that current limit and specifying an error return string (`"Over Limit"`) with the `RETURN ERROR` statement to indicate this condition:

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

CLASS acme.myObjs.CreditObj INHERITS CommonObj:

    DEFINE PUBLIC PROPERTY CustCreditLimit AS DECIMAL INITIAL ?
        /* 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.
                RETURN ERROR "Over Limit".
            END.
            ELSE DO:
                Customer.Creditlimit = piCL + 1000.
                CustCreditLimit = Customer.Creditlimit.
            END.
        END SET.

    ...

    METHOD PUBLIC VOID CheckCustCredit ( ):
        /* invokes the CustCreditLimit property SET accessor */
        IF AVAILABLE (Customer) THEN DO:
            CustCreditLimit = Customer.Creditlimit NO-ERROR.
            IF ERROR-STATUS:ERROR THEN DO:
                /* Indicates Customer balance is greater than credit limit */
                /* and returns error string from property SET              */
                RETURN ERROR RETURN-VALUE.
            END.
        END.
        ELSE
            RETURN ERROR "No Customer".
    END METHOD.

END CLASS.
```

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 checks for any `ERROR` condition raised by setting the property value. If the property raises `ERROR`, the method simply invokes its own `RETURN ERROR` statement, passing up the same value returned from the property in `RETURN-VALUE` as an error return string to the caller. This return error string, in turn, gets checked by the `CheckCredit( )` method in the sample class, `acme.myObjs.CreditObj` (see the "Method error handling example" section on page 4–47).

Errors within a constructor

There are two types of errors that can occur during the creation of an object:

- **Instantiation error** — 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 such instantiation errors that you cannot catch within a constructor, the AVM automatically raises the ERROR condition on that 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 the "Compiling class definition files" section on page 6–5.

- **Application error** — Your application logic encounters an abnormal condition in one of the constructors in the class hierarchy. Depending on the application error, it can raise ERROR on a statement within the constructor, such as a FIND statement that cannot find a record, or you might catch an error as part of evaluating the results of an operation. For application errors that raise ERROR on a statement, you might catch and handle the ERROR condition within the constructor using NO-ERROR (on the statement) and the ERROR-STATUS system handle. For all application errors that you do catch programmatically, you can also use RETURN ERROR to raise the ERROR condition on the constructor.

Handling constructor errors

So, as with any type of error in a method or procedure, you can either handle errors within a constructor or use RETURN ERROR to raise the ERROR condition in its caller. Whenever you raise the ERROR condition on a constructor, the AVM raises that ERROR condition directly in the statement that instantiated the object with the NEW phrase, regardless of where in the class hierarchy the constructor is running. In other words, raising an ERROR on any constructor immediately terminates object instantiation. In addition, instead of returning an object reference to an instantiated object, the ERROR condition causes the NEW phrase to return no object reference, leaving any object reference set to receive a value from the NEW phrase 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.) You can thus only handle constructor errors on the statement that instantiates the object using the NEW phrase.

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: Other than raising ERROR, the RETURN ERROR behavior is similar to invoking the DELETE OBJECT THIS-OBJECT statement from within a constructor, except that DELETE OBJECT THIS-OBJECT causes the NEW phrase to return an object reference as the Unknown value (?) instead of not returning an object reference at all. However, because of the general facility for returning information about a particular application condition, using the ERROR-STATUS system handle and the RETURN-VALUE function, you might prefer to terminate object instantiation within any constructor using RETURN ERROR.

Constructor error handling example

In the following example, one class (Derived) inherits behavior from another class (Base), each of which can halt object instantiation for a different reasons that are each handled by the instantiating procedure (instantiater2.p):

Base.cls

```
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.
```

Derived.cls

```
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.
```

instantiater2.p

```
DEFINE VARIABLE rDerived AS CLASS Derived NO-UNDO.
DEFINE VARIABLE id AS INTEGER NO-UNDO.

rDerived = NEW Derived (INPUT "fred", OUTPUT id) NO-ERROR.
IF ERROR-STATUS:ERROR = TRUE THEN DO: /* Do error processing. */
  IF RETURN-VALUE = "Specified record not found" ...
  ELSE ...
END.
ELSE ... /* Do normal application processing */
```

In instantiater2.p, the AVM knows if the Derived class is not instantiated properly by checking for an ERROR condition raised on the Assignment (=) 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 catches and raises ERROR using RETURN ERROR with an error string that indicates no records are found. If the application-specified record is not found in the Customer table, the Derived class constructor catches the error and decides to terminate object instantiation using RETURN ERROR with an error string that indicates the specified record cannot be found. In this case, the AVM also runs the default destructor for the Base class to undo its instantiation. The instantiating procedure, instantiater2.p, can then check what type of error caused the failure to instantiate the object according to how the ERROR condition is raised.

Thus, ABL error handling provides a single mechanism to handle errors at any level of an instantiating class hierarchy.

Errors within a destructor

You cannot use RETURN ERROR in destructors. 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 destroyed with no indication that an error occurred during the process.

Reflection—Using the Progress.Lang.Class class

Reflection is the ability to get type information about a class or interface at run time. This information describes the non-private components of the class or interface. ABL provides basic reflection support using the built-in ABL class, Progress.Lang.Class. This support allows you to get basic information about the type and hierarchy of a class-based object.

Note: In a future OpenEdge release, Progress.Lang.Class will provide more complete reflection support.

Getting a Progress.Lang.Class instance

You can get an instance of a Progress.Lang.Class for an instantiated class-based object by invoking the ABL GetClass( ) method on any valid object reference.

This is the syntax to get a Progress.Lang.Class for an instantiated object:

Syntax

```
class-reference = object-reference:GetClass( ).
```

Element descriptions for this syntax diagram follow:

class-reference

An object reference defined as the data type, Progress.Lang.Class.

object-reference

A reference to an instantiated class-based object.

For example:

```
USING Progress.Lang.*.
USING acme.myObjs.*.

DEFINE VARIABLE myCustObj AS CLASS CustObj NO-UNDO.
DEFINE VARIABLE myType AS CLASS Class NO-UNDO.
myCustObj = NEW CustObj ( ).

myType = myCustObj:GetClass( ).
```

Using Progress.Lang.Class properties and methods

Progress.Lang.Class is a built-in ABL class that provides a common set of properties and methods that return type information about a class or interface. Progress.Lang.Class is FINAL and therefore it can not be inherited. Each ABL session contains a single Progress.Lang.Class instance for each type of class-based object instantited in the session. The lifetime of these objects is controlled by the ABL session; therefore, you cannot delete them. Table 4–1 describes the common properties and methods on Progress.Lang.Class.

Table 4–1: Progress.Lang.Class public properties and methods

Property or method	Description
PUBLIC TypeName AS CHARACTER	A read-only property that contains the fully qualified class or interface type name. For example, "acme.myObjs.CustObj" (the sample class defined in the "Sample classes" section on page 5–10).
PUBLIC Package AS CHARACTER	A read-only property that contains the package (relative directory) of the class or interface type name. For example, "acme.myObjs" (the sample package used in the "Sample classes" section on page 5–10).
PUBLIC SuperClass AS CLASS Progress.Lang.Class	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.
METHOD PUBLIC LOGICAL IsInterface ( )	Returns TRUE if the type is a definition for an interface, and FALSE if it is for a class.
METHOD PUBLIC LOGICAL IsFinal ( )	Returns TRUE if the type is defined as FINAL, and FALSE if it is not FINAL.

5

Programming 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 chapters of this manual.

Thus, the sections in this chapter describe:

- Class-based and procedure object compatibility
- Handling events
- Using widget pools
- Referencing routines on the call stack
- Comparing handles and object references
- Comparing constructs in 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 user-defined data type. You must use the syntax described in this manual to instantiate a class-based object 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 phrase 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 phrase to create a class instance.
- You can only use the NEW phrase to create an instance of a class. You cannot use the NEW phrase 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 references from one to another, you cannot cast an object reference 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 PUBLIC, PRIVATE, or PROTECTED 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 identified in the OpenEdge 10 Platform and “Product Availability Guide” available at this Web location:

```
http://www.progress.com/progress_software/products/docs/bu_sep/openedge_10_availability_guide.pdf
```

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.

Verifying 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.

Handling events

ABL supports events with several different constructs. Classes support widget and low-level ABL events using the ON statement, and support ProDataSet, query, and buffer object callbacks using the SET-CALLBACK( ) method. Classes do not support the PUBLISH and SUBSCRIBE statements.

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 code demonstrates this capability:

```
CLASS Disp:
    DEFINE BUTTON msg.    /* These data members are PRIVATE by default. */
    DEFINE BUTTON done.
    DEFINE FRAME f msg done.

    ON 'choose':U OF msg  /* Unconditional ON block valid in the main block. */
        MESSAGE "click" VIEW-AS ALERT-BOX.

    CONSTRUCTOR PUBLIC Disp ( ):
        modal_display ( ).
    END CONSTRUCTOR.

    METHOD PRIVATE VOID modal_display ( ): /* WAIT-FOR is done in a method. */
        ENABLE ALL WITH FRAME f.
        WAIT-FOR CHOOSE OF done.
    END METHOD.

END CLASS.
```

SET-CALLBACK( ) method

The SET-CALLBACK( ) built-in method is used to associate a method and object reference with ABL-defined events on three objects:

- ProDataSet object.
- Buffer object.
- Query 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 *OpenEdge Development: 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.

Using widget pools

Classes instantiated using the NEW phrase, cannot be created in a widget pool. You must assign each instantiated class-based object to an object reference and explicitly delete that object when it is no longer needed.

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 types 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, you can create a single unnamed widget pool scoped to the entire class by specifying the USE-WIDGET-POOL option on the CLASS statement. For more information, see the "Using the CLASS construct" section on page 2–11. The unnamed widget pool created using the USE-WIDGET-POOL option is scoped to the entire class and is implicitly deleted by the ABL session when the class-based object is deleted.

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.

Referencing 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 with the following form:

Syntax

```
"method-name class-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.

Comparing 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, super class, or interface type, or in a temp-table field of type Progress.Lang.Object. These two reference types are not interchangeable.

Using 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, including a procedure, a visual object, a data management object such as a buffer or query, and so on. 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.

Using object references

Because classes in ABL are strongly-typed, you can cast an object reference as needed to allow a single variable or temp-table field to hold a reference to a variety of types. If you define a variable or 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, or methods 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 attempts to reference this subclass functionality as a compiler error. For more information on casting, see the "Assignment and the CAST function" section on page 4–37.

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 the "Defining an object reference as a field in a temp-table" section on page 4–27.

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.
```

Comparing 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 the Progress Software Developers Network Web site:

```
http://www.psdn.com
```

These sample classes and procedures are also available for compiling and running in your OpenEdge development environment. For information on accessing these samples online, see the "Example procedures" section on page Preface–7.

Both versions of this sample application rely on the `Customer` table of the sports2000 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. Following is a description of the relationships among the classes, interface, and procedure, using unqualified names for the application classes and interface for readability:

```
Class MsgObj INHERITS Class Progress.Lang.Object

Class CommonObj INHERITS Class Progress.Lang.Object
        instantiates and uses Class MsgObj
    Class CreditObj INHERITS Class CommonObj
    Class CustObj   INHERITS Class CommonObj and IMPLEMENTS Interface IBusObj
            instantiates and uses Class CreditObj
            instantiates and uses Class MsgObj
        Class NECustObj INHERITS CustObj

Class HelperClass INHERITS Class Progress.Lang.Object
        instantiates and uses Class MsgObj
        uses 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 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 super class for two class hierarchies and provides a common error handler and time-tracking methods for all classes that inherit from it:

CommonObj.cls

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

CLASS acme.myObjs.Common.CommonObj:

    DEFINE PROTECTED VARIABLE timestamp AS DATETIME NO-UNDO.          /* 5 */

    METHOD PUBLIC VOID updateTimestamp ( ):
        timestamp = NOW.
    END METHOD.

    METHOD PROTECTED CLASS MsgObj
             MessageHandler (INPUT iObjType AS CHARACTER):            /* 3 */
        DEFINE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
        rMsg = NEW MsgObj (INPUT iObjType).
        RETURN rMsg.
    END METHOD.

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 copies AS CHARACTER).
    METHOD PUBLIC VOID logObj (INPUT filename 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 *(1 of 3)*

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

CLASS acme.myObjs.CustObj   INHERITS CommonObj                        /* 1 */
                            IMPLEMENTS IBusObj:

    DEFINE PUBLIC VARIABLE iNumCusts AS INTEGER NO-UNDO.
    DEFINE PROTECTED TEMP-TABLE ttCust NO-UNDO                        /* 14 */
        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.
```

CustObj.cls *(2 of 3)*

```
    CONSTRUCTOR PUBLIC CustObj ( ):                                          /* 2 */
        rCreditObj = NEW CreditObj ( ).

        iNumCusts = 0.
        /* Fill temp table and get row count */
        FOR EACH Customer WHERE CreditLimit > 50000:
            CREATE ttCust.
            ASSIGN
                iNumCusts = iNumCusts + 1
                ttCust.RecNum = iNumCusts
                ttCust.CustNum = Customer.CustNum
                ttCust.Name = Customer.Name
                ttCust.State = Customer.State.
        END.
        rMsg = MessageHandler (INPUT "acme.myObjs.CustObj").        /* 3 */
    END CONSTRUCTOR.

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

METHOD PUBLIC VOID CheckCredit ( ):
        IF VALID-OBJECT (rCreditObj) THEN DO:
            FOR EACH ttCust:
                rCreditObj:SetCurrentCustomer (INPUT ttCust.CustNum).
                rCreditObj:CheckCustCredit ( ) NO-ERROR.
                IF ERROR-STATUS:ERROR THEN                              /* 13 */
                    IF RETURN-VALUE = "Over Limit" THEN DO:
                     /* invokes the CustCreditLimit property GET accessor */
                        rMsg:Alert (INPUT ttCust.Name +
                            " is on Credit Hold." +
                            "  Balance exceeds Credit Limit of " +
                            STRING (rCreditObj:CustCreditLimit)).   /* 12 */
                    END.
                    ELSE
                        rMsg:Alert (INPUT "Customer not found").
                ELSE DO:
                    /* invokes the CustCreditLimit property GET accessor */
                    rMsg:InfoMsg (INPUT ttCust.Name +
                            " is in good standing." +
                            "  Credit Limit has been increased to " +
                            STRING(rCreditObj:CustCreditLimit)).    /* 12 */
                END.
            END. /* FOR EACH */
        END.
        ELSE rMsg:Alert (INPUT "Unable to check credit").
    END METHOD.
```

CustObj.cls *(3 of 3)*

```
    /* 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 timestamp.                                            /* 5 */
        FOR EACH ttCust:
            DISPLAY ttCust.
        END.
        OUTPUT CLOSE.
    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 */
        DEFINE VARIABLE iCnt AS INTEGER.
        OUTPUT TO PRINTER.
        IF piCopies <> 0 THEN DO iCnt = 1 TO ABS(piCopies):
            DISPLAY timestamp.                                        /* 5 */
            FOR EACH ttCust:
                DISPLAY ttCust.
            END.
        END.
        OUTPUT CLOSE.
    END METHOD.

    METHOD PUBLIC VOID logObj (INPUT fileName AS CHARACTER):
        OUTPUT TO VALUE(fileName).
        DISPLAY timestamp.                                            /* 5 */
        FOR EACH ttCust:
            DISPLAY ttCust.
        END.
        OUTPUT CLOSE.
    END METHOD.

    DESTRUCTOR PUBLIC CustObj ( ):                                    /* 6 */
        EMPTY TEMP-TABLE ttCust.
        DELETE OBJECT rMsg.
        rMsg = ?.
        DELETE OBJECT rCreditObj.
        rCreditObj = ?.
    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 ttCust WHERE   ttCust.State = "MA" OR                 /* 14 */
                                ttCust.State = "VT" OR
                                ttCust.State = "NH" OR
                                ttCust.State = "CT" OR
                                ttCust.State = "RI" OR
                                ttCust.State = "ME" :
            CREATE ttEmail.
            ASSIGN
                ttEmail.RecNum = ttCust.RecNum
                ttEmail.Name = ttCust.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 */
        DEFINE VARIABLE EmailName AS CHARACTER NO-UNDO.

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

    /* Override method to log customer information with email */
    METHOD PUBLIC OVERRIDE VOID logObj (INPUT fileName AS CHARACTER):
        OUTPUT TO VALUE (fileName).
        DISPLAY timestamp.                                             /* 5 */
        FOR EACH ttEmail:
            DISPLAY ttEmail.
        END.
        OUTPUT CLOSE.
    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:

    DEFINE PRIVATE VARIABLE rCustObj AS CLASS CustObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rMsg AS CLASS MsgObj NO-UNDO.
    DEFINE PRIVATE TEMP-TABLE ttNames NO-UNDO
        FIELD CustName AS CHARACTER.

    CONSTRUCTOR PUBLIC HelperClass ( ):
        rMsg = NEW MsgObj (INPUT "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 (INPUT "Not a valid object").
    END METHOD.

    METHOD PUBLIC VOID ListNames (INPUT-OUTPUT prCustObj AS CLASS CustObj):
        DEFINE VARIABLE idx AS INTEGER NO-UNDO.

        DO idx = 1 to prCustObj:iNumCusts:
            CREATE ttNames.
            ttNames.CustName = prCustObj:GetCustomerName (INPUT 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 (INPUT "Not a valid object").
    END METHOD.

    DESTRUCTOR PUBLIC HelperClass ( ):
        DELETE OBJECT rMsg.
        rMsg = ?.
    END DESTRUCTOR.

END CLASS.
```

This class provides a method for notification of customers who have exceeded their credit limit:

CreditObj.cls

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

CLASS acme.myObjs.CreditObj INHERITS CommonObj:

    DEFINE PUBLIC PROPERTY CustCreditLimit AS DECIMAL INITIAL ?      /* 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.
                RETURN ERROR "Over Limit".                           /* 13 */
            END.
            ELSE DO:
                Customer.Creditlimit = piCL + 1000.
                CustCreditLimit = Customer.Creditlimit.
            END.
        END SET.

    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 property SET */
        IF AVAILABLE (Customer) THEN DO:
            CustCreditLimit = Customer.Creditlimit NO-ERROR.         /* 12 */
            IF ERROR-STATUS:ERROR THEN DO:
                /* Indicates Customer balance is greater than credit limit */
                /* and returns error string from property SET              */
                RETURN ERROR RETURN-VALUE.                           /* 13 */
            END.
        END.
        ELSE
            RETURN ERROR "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:

    DEFINE PRIVATE VARIABLE ObjType AS CHARACTER NO-UNDO.

    CONSTRUCTOR PUBLIC MsgObj (INPUT rObjType AS CHARACTER):
        ObjType = rObjType.
    END CONSTRUCTOR.

    METHOD PUBLIC VOID Alert (INPUT ErrorString AS CHARACTER):
        MESSAGE "Error in " ObjType "!" 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 *(1 of 2)*

```
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 rCustObj2 AS CLASS CustObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rCommonObj AS CLASS CommonObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rIBusObj AS CLASS IBusObj NO-UNDO.
    DEFINE PRIVATE VARIABLE rHelperClass AS CLASS HelperClass NO-UNDO.
    DEFINE PRIVATE VARIABLE outFile AS CHARACTER.

    /* first constructor instantiates a Customer object */
    CONSTRUCTOR PUBLIC Main ( ):                                    /* 10 */
        /* Create an instance of the HelperClass class  */
        rHelperClass = NEW HelperClass ( ).

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

        /* Create an instance of the NECustObj class   */
        rCustObj = NEW NECustObj (INPUT EmailFile).
        outFile = "NECustomers.out".
    END CONSTRUCTOR.
```

Main.cls *(2 of 2)*

```
    /* ObjectInfo processes information about the Customer object */
    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 (INPUT rCustObj).                /* 4 */
        rCommonObj = rCustObj.

        /* 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).
        rIBusObj:logObj (INPUT outFile).
        rIBusObj = rCustObj.
    END METHOD.

    DESTRUCTOR PUBLIC Main ( ):
        DELETE OBJECT rCustObj.
        rCustObj = ?.
        DELETE OBJECT rHelperClass.
        rHelperClass = ?.
    END DESTRUCTOR.

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 **/

DEFINE VARIABLE ClassExample AS CLASS Main.

/* run the example for all Customers         */
ClassExample = NEW Main ( ).                                    /* 10 */
ClassExample:ObjectInfo ( ).
DELETE OBJECT ClassExample.
ClassExample = ?.

/* run the example for New England Customers */
classExample = NEW Main (INPUT "email.txt").                    /* 10 */
ClassExample:ObjectInfo ( ).
DELETE OBJECT ClassExample.
ClassExample = ?.
```

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. 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 timestamp as SHARED to illustrate the counterpart */
/* to inherited data members in classes (#5 in Table 5.1)   */

DEFINE SHARED VARIABLE timestamp AS DATETIME NO-UNDO.                 /* 5 */
DEFINE VARIABLE hMsg AS HANDLE NO-UNDO.

PROCEDURE updateTimestamp:
    timestamp = NOW.
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.
    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 *(1 of 2)*

```
/* Main Block */                                                    /* 2 */

DEFINE SHARED VARIABLE iNumCusts AS INTEGER NO-UNDO.
DEFINE TEMP-TABLE ttCust NO-UNDO                                    /* 14 */
    FIELD RecNum AS INTEGER
    FIELD CustNum LIKE Customer.CustNum
    FIELD Name LIKE Customer.Name
    FIELD State AS CHARACTER.
DEFINE NEW SHARED VARIABLE timestamp AS DATETIME NO-UNDO.            /* 5 */
DEFINE VARIABLE hMsg AS HANDLE NO-UNDO.
DEFINE VARIABLE hCommon AS HANDLE NO-UNDO.
DEFINE VARIABLE hCreditProc AS HANDLE NO-UNDO.

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.

iNumCusts = 0.
/* Fill temp table and get row count */
FOR EACH Customer WHERE CreditLimit > 50000:
    CREATE ttCust.
    ASSIGN
        iNumCusts = iNumCusts + 1
        ttCust.RecNum = iNumCusts
        ttCust.CustNum = Customer.CustNum
        ttCust.Name = Customer.Name
        ttCust.State = Customer.State.
END.
hMsg = MessageHandler(INPUT "CustProc").                             /* 3 */

PROCEDURE GetCustomerName:
    DEFINE INPUT PARAMETER piRecNum AS INTEGER NO-UNDO.
    DEFINE OUTPUT PARAMETER poName AS CHARACTER NO-UNDO.

    FIND ttCust WHERE ttCust.RecNum = piRecNum NO-ERROR.
    IF AVAILABLE ttCust THEN DO:
        poName = ttCust.Name.
    END.
    ELSE DO:
        RUN Alert IN hMsg(INPUT "Customer number" + STRING(ttCust.RecNum)
                          + "does not exist").
        poName = ?.
    END.
END PROCEDURE.
```

CustProc.p *(2 of 2)*

```
PROCEDURE CheckCredit:
    IF VALID-HANDLE (hCreditProc) THEN DO:
        FOR EACH ttCust:
            RUN SetCurrentCustomer IN hCreditProc (INPUT ttCust.CustNum).
            RUN CheckCustCredit IN hCreditProc NO-ERROR.
            IF ERROR-STATUS:ERROR THEN                                  /* 13 */
                RUN Alert IN hMsg (INPUT "Customer not found").
            ELSE DO:
                IF RETURN-VALUE = "Over Limit" THEN DO:                 /* 13 */
                    RUN Alert IN hMsg (INPUT ttCust.Name +
                                       " is on Credit Hold." +
                                       "  Balance exceeds Credit Limit of " +
                                       STRING (GetCreditLimit( )) ).    /* 12 */
                END.
                ELSE DO:
                    RUN InfoMsg IN hMsg (INPUT ttCust.Name +
                                       " is in good standing." +
                                       "  Credit Limit has been increased to " +
                                       STRING (GetCreditLimit( )) ).    /* 12 */
                END.
            END.
        END. /* FOR EACH */
    END.
    ELSE RUN Alert in hMsg (INPUT "Unable to check credit").
END PROCEDURE.

/* timestamp is a SHARED variable defined in CommonProc.p as well */
PROCEDURE printProc:                                                    /* 11 */
    OUTPUT TO PRINTER.
    DISPLAY timestamp.                                                  /* 5 */
    FOR EACH ttCust:
        DISPLAY ttCust.
    END.
    OUTPUT CLOSE.
END PROCEDURE.

PROCEDURE logProc:
    DEFINE INPUT PARAMETER fileName AS CHARACTER NO-UNDO.
    OUTPUT TO VALUE(fileName).
    DISPLAY timestamp.                                                  /* 5 */
    FOR EACH ttCust:
        DISPLAY ttCust.
    END.
    OUTPUT CLOSE.
END PROCEDURE.

PROCEDURE GetTT:                                                        /* 14 */
    DEFINE OUTPUT PARAMETER TABLE for ttCust BIND.
END PROCEDURE.

PROCEDURE CleanUp:
    EMPTY TEMP-TABLE ttCust.
    DELETE OBJECT hMsg.
    hMsg = ?.
    DELETE OBJECT hCreditProc.
    hCreditProc = ?.
    RUN CleanUp IN hCommon.
    DELETE OBJECT hCommon.
    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 VARIABLE hCustProc AS HANDLE NO-UNDO.
DEFINE VARIABLE hCommonProc AS HANDLE NO-UNDO.
DEFINE NEW SHARED VARIABLE timestamp AS DATETIME NO-UNDO.
DEFINE TEMP-TABLE ttCust 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 ttCust BIND).

INPUT FROM VALUE(EmailFile).
FOR EACH ttCust WHERE   ttCust.State = "MA" OR                     /* 14 */
                        ttCust.State = "VT" OR
                        ttCust.State = "NH" OR
                        ttCust.State = "CT" OR
                        ttCust.State = "RI" OR
                        ttCust.State = "ME" :
    CREATE ttEmail.
    ASSIGN
        ttEmail.RecNum = ttCust.RecNum
        ttEmail.Name = ttCust.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.
    IF AVAILABLE(ttEmail) THEN
        RETURN poName + ";" + ttEmail.Email.
END PROCEDURE.
```

This procedure provides a notification of customers who have exceeded their credit limit:

CreditProc.p

```
DEFINE VARIABLE TempCustNum AS INTEGER.
DEFINE VARIABLE CustCreditLimit AS DECIMAL INITIAL ?.                /* 12 */

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:
        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.
END PROCEDURE.

PROCEDURE CheckCustCredit:
    IF AVAILABLE (Customer) THEN DO:
        IF SetCreditLimit (Customer.Creditlimit) THEN                /* 12 */
            RETURN "Credit Good".
        ELSE
            RETURN "Over Limit".                                     /* 13 */
    END.
    ELSE
        RETURN ERROR.                                                /* 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 VARIABLE hCustProc AS HANDLE NO-UNDO.
DEFINE VARIABLE oCustNum  AS INTEGER NO-UNDO.
DEFINE VARIABLE oCustName AS CHARACTER NO-UNDO.
DEFINE VARIABLE idx       AS INTEGER NO-UNDO.

/* define iNumCusts as SHARED to illustrate the counterpart */
/* to PUBLIC data members in classes.                       */
DEFINE NEW SHARED VARIABLE iNumCusts AS INTEGER NO-UNDO.
DEFINE TEMP-TABLE ttCustNames NO-UNDO
    FIELD CustName AS CHARACTER.
DEFINE VARIABLE i AS INTEGER NO-UNDO.

RUN CustProc.p PERSISTENT SET hCustProc.

RUN updateTimestamp IN hCustProc.                                   /* 4 */

DO idx = 1 to iNumCusts:
    CREATE ttCustNames.
    RUN GetCustomerName IN hCustProc(INPUT idx, OUTPUT oCustName).
    ttCustNames.CustName = oCustName.
END.

RUN CheckCredit IN hCustProc.

RUN logProc IN hCustProc (INPUT "CustomerProc.out").

RUN CleanUp IN hCustProc.                                           /* 6 */

DELETE OBJECT hCustProc.
hCustProc = ?.
```

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 VARIABLE hCustProc AS HANDLE NO-UNDO.
DEFINE VARIABLE oCustNum  AS INTEGER NO-UNDO.
DEFINE VARIABLE oCustName AS CHARACTER NO-UNDO.
DEFINE VARIABLE idx       AS INTEGER NO-UNDO.

/* define iNumCusts as SHARED to illustrate the counterpart */
/* to PUBLIC data members in classes.                       */
DEFINE NEW SHARED VARIABLE iNumCusts AS INTEGER NO-UNDO.
DEFINE TEMP-TABLE ttCustNames NO-UNDO
    FIELD CustName AS CHARACTER.
DEFINE VARIABLE i AS INTEGER NO-UNDO.

RUN NECustProc.p PERSISTENT SET hCustProc (INPUT "email.txt").

RUN updateTimestamp IN hCustProc.                                   /* 4 */

DO idx = 1 to iNumCusts:
    CREATE ttCustNames.
    RUN GetCustomerName IN hCustProc(INPUT idx, OUTPUT oCustName).
    ttCustNames.CustName = oCustName.
END.

RUN CheckCredit IN hCustProc.

RUN logProc IN hCustProc (INPUT "CustomerProc.out").

RUN CleanUp IN hCustProc.                                           /* 6 */

DELETE OBJECT hCustProc.
hCustProc = ?.
```

Summary comparison of classes and procedures

Table 5–1 compares the application implemented by the sample classes (see the "Sample classes" section on page 5–10) with the equivalent application implemented by comparative procedures (see the "Comparative procedures" section on page 5–19). The code numbers in the table match the commented numbers in the code for both classes and procedures.

Table 5–1: Comparing sample classes to comparative procedures *(1 of 4)*

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.
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 invoked inside the `acme.myObjs.CustObj` class. The method is found 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 `acme.myObjs.CustObj`, and is found and executed in the `CustObj` super class, `acme.myObjs.Common.CommonObj`. (This method is called 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`.

Table 5–1: Comparing sample classes to comparative procedures *(2 of 4)*

Code No.	Sample classes	Comparative procedures
5	The `updateTimestamp( )` method initializes the value of the `PROTECTED timestamp` 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, `timestamp`. In order for both `CustProc.p` and `NECustProc.p` to use `timestamp`, 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`.
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`.

Table 5–1: Comparing sample classes to comparative procedures *(3 of 4)*

Code No.	Sample classes	Comparative procedures
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` object and one that takes as a parameter the name of a text file containing E-mail addresses used to initialize an `acme.myObjs.NECustObj` object. The driver procedure, `Driver.p`, instantiates the `Main` class once using each constructor.	`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.
11	The `acme.myObjs.CustObj` class defines two overloads of the `printObj( )` method, one that prints a single copy of a report to the default printer and another that prints the number of copies specified by a parameter. `CustObj` defines these methods because it implements the interface, `acme.myObjs.Interfaces.IBusObj`, which specifies them.	The `CustProc.p` defines a single `PrintProc` internal procedure that prints one copy of a report to the default printer. To print a specified number of copies, it can either define an additional procedure with a different name that takes a parameter to specify the number of copies, or add the parameter to `PrintProc` itself to handle both the single copy and multiple copy scenarios.

Table 5–1: Comparing sample classes to comparative procedures *(4 of 4)*

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 publically 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 raises ERROR and returns an appropriate error string when it is set for a customer whose balance is over their current credit limit. The CheckCustCredit( ) method checks the property error and passes it up, if it is raised, and also raises its own ERROR and returns 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 error handling capability for its corresponding user-defined functions and internal procedure, although not quite as efficiently because of the limitations of using the RETURN statement for user-defined functions and procedures. The CheckCredit internal procedure of CustProc.p responds to these error conditions in a similar fashion, as well.
14	The acme.myObjs.CustObj class defines a PROTECTED temp-table (ttCust) that is accessed by its subclass, acme.myObjs.NECustObj. Because of data inheritance, both CustObj and NECustObj directly access the same instance of ttCust.	CustProc.p defines a private temp-table (ttCust) 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 ttCust in NECustProc.p. This allows both CustProc.p and NECustProc.p to access the same instance of ttCust.

6

Developing and Deploying Classes

OpenEdge supports the development and deployment of classes with several features of the ABL development environment, as described in these sections:

- Accessing class definition files using the Procedure Editor
- Accessing class definition files using OpenEdge Architect
- Compiling class definition files
- Using procedure libraries
- Using the XCODE utility

Accessing class definition files using 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.

Saving and opening 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.

Checking and running 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 the "Using the CLASS construct" section on page 2–11 or the "Using the INTERFACE construct" section on page 2–38, 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.

Accessing class definition files using OpenEdge Architect

OpenEdge Architect provides support for creating new class and interface definition files (`.cls`) through wizards. To follow this description, see Figure 6–1. 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 figure.

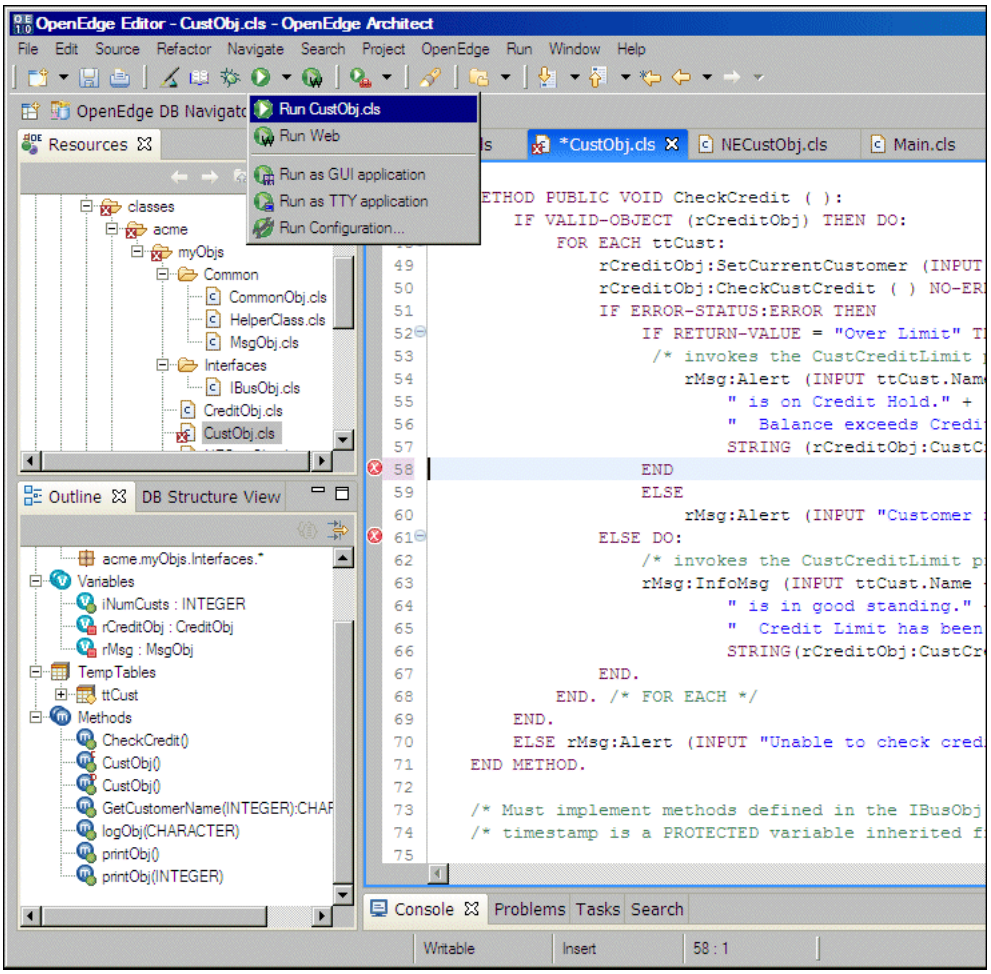

Figure 6–1: Class definition file open in OpenEdge Architect

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 checking, compiling, and running 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, OpenEdge Architect online help.

▶ To locate these class topics, choose **Help→OpenEdge Architecture Guide→ABL Tools→ Writing and Testing ABL Code**.

Compiling class definition files

A class is a user-defined data type and 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 referenced class definition files, that is, for other classes that are instantiated within a class using the `NEW` phrase. The compiler instead inspects referenced class files to determine their external interfaces (their public data members, properties, and methods). This information is used to validate the use of these classes in the class being compiled. The compiler traces up through the referenced class's hierarchy for super classes and interfaces. It does not check any additional class references specified in this referenced class hierarchy.

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 state or behavior. 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 Figure 6–2, **ClassC** inherits from both **ClassB** and **ClassA**. **ClassC** also implements **InterfaceI1**, and **ClassB** implements **InterfaceI2**.

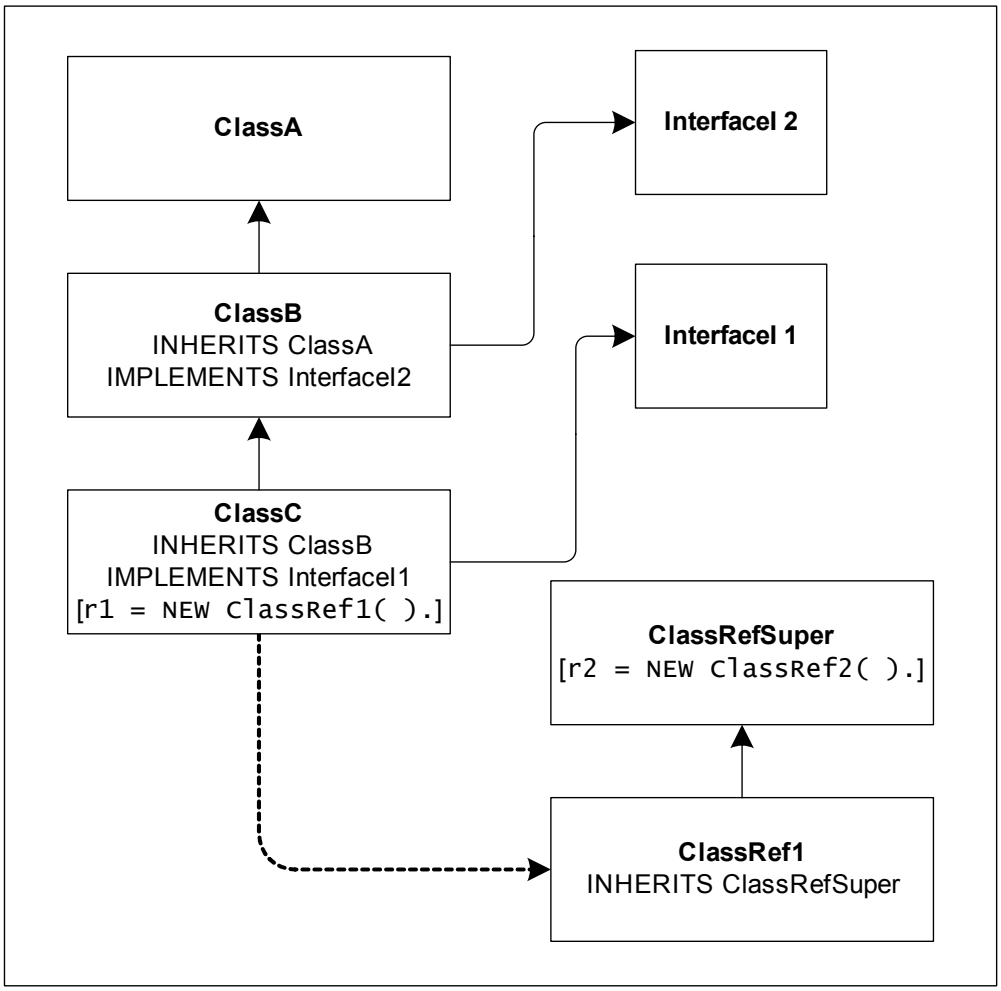

Figure 6–2: Compiling class definition files

During the compilation of **ClassC**, the compiler generates separate r-code files for **ClassC**, **ClassB** and **ClassA**, as well as for **InterfaceI1** and **InterfaceI2**. 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**.

Again, using Figure 6–2, 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**.

Data type mapping

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 *OpenEdge Development: ABL Reference*.

Using 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 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
- CLASS-TYPE attribute
- NUM-MESSAGES attribute
- GET-MESSAGE(n) method
- GET-NUMBER(n) method
- GET-FILE-NAME(n) method
- GET-ERROR-ROW(n) method
- GET-ERROR-COLUMN(n) method
- GET-FILE-OFFSET(n) method

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 or 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.

Using procedure libraries

You can store and retrieve the r-code for class files using procedure libraries.

The DEFINE statements (such as DEFINE VARIABLE) and NEW phrase 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:

Syntax

```
RUN procedure-library-path<<member-name>>
```

There is **no** support of this capability for classes. When you define an object reference or create an object instance, the r-code for that object must be found on the PROPATH.

Using 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 *OpenEdge Deployment: Managing 4GL Applications*.

A

Overloaded Method and Constructor Calling 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 class 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. The following sections describe some of these scenarios and how ABL handles them:

- Parameters differing only by mode
- Parameters matching widened data types
- Matching dynamic and static temp-table or ProDataset parameters
- Object reference parameters matching a class hierarchy or interface
- Matching the Unknown value (?) to parameters
- Matching values of unknown data types to parameters

Parameters differing 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: Progress Software Corporation recommends that you always specify the mode for all parameters of a class-based method call.

Parameter data types differing only by extent

Parameter data types differing by extent (arrays) typically match strictly according to scalar type and extent. However, if the method is called using a parameter with a fixed extent, and no method exists that matches a corresponding parameter with that fixed extent, the call can be resolved by a method definition whose corresponding parameter has an indeterminate extent. In any case, failure to find a match raises a compile-error.

The following examples show how different calls to overloaded methods (or constructors) with parameters of the same data type, but with different extents, 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 bolded number referenced in the comments:

```
DEFINE VARIABLE rExt AS CLASS Extents.

DEFINE VARIABLE i00 AS INTEGER.
DEFINE VARIABLE i10 AS INTEGER EXTENT 10.
DEFINE VARIABLE i08 AS INTEGER EXTENT 8.
DEFINE VARIABLE ixx AS INTEGER EXTENT.

rExt = NEW Extents( ).

rExt:setVal(INPUT 42 ). /* Calls 1 - constant integer                    */
rExt:setVal(INPUT i00). /* Calls 1 - integer variable                    */
rExt:setVal(INPUT i10). /* Calls 2 - fixed array variable of extent 10   */
rExt:setVal(INPUT i08). /* Calls 3 - no fixed array variable of extent 8:
                                      matches an indeterminate extent     */
rExt:setVal(INPUT ixx). /* Calls 3 - indeterminate extent                */
```

Parameters matching 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 *OpenEdge Development: ABL Reference*.

The following examples 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 bolded number referenced in the comments:

```
DEFINE VARIABLE rWid AS CLASS Widening.

DEFINE VARIABLE iVal   AS INTEGER INITIAL 42.
DEFINE VARIABLE i64Val AS INT64 INITIAL 42.
DEFINE VARIABLE dVal   AS DECIMAL INITIAL 4.2.

rWid = NEW Widening( ).

rWid:setVal (INPUT iVal).   /* Calls 1 - matches exactly                */
rWid:setVal (INPUT i64Val). /* Calls 2 - INT64 matches the wider DECIMAL */
rWid:setVal (INPUT dVal).   /* Calls 2 - matches exactly                */
```

Matching 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 following example 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 CustTT  LIKE Customer.
    DEFINE TEMP-TABLE OrderTT LIKE Order.

    METHOD PUBLIC VOID setTable(INPUT TABLE CustTT):   /* first */
    END METHOD.

    METHOD PUBLIC VOID setTable(INPUT TABLE OrderTT):  /* second */
    END METHOD.

    METHOD PUBLIC VOID Scenario1( ):
        CREATE CustTT.
        setTable(INPUT TABLE CustTT).
    END METHOD.

    METHOD PUBLIC VOID Scenario2( ):
        DEFINE VARIABLE hTT AS HANDLE.
        CREATE TEMP-TABLE hTT.
        hTT:CREATE-LIKE("Customer").
        hTT:TEMP-TABLE-PREPARE("Customer").
        setTable(INPUT TABLE-HANDLE hTT).
    END METHOD.

    METHOD PUBLIC VOID Scenario3( ):
        DEFINE VARIABLE hTT AS HANDLE.
        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 CustTT LIKE Customer.
    DEFINE TEMP-TABLE OrderTT LIKE Order.

    METHOD PUBLIC VOID setTable(INPUT TABLE CustTT):        /* first */
    END METHOD.

    METHOD PUBLIC VOID setTable(OUTPUT TABLE OrderTT):      /* second */
    END METHOD.

    METHOD PUBLIC VOID setTable(INPUT TABLE-HANDLE ttHndl): /* third */
    END METHOD.

    METHOD PUBLIC VOID Scenario1( ):
        CREATE OrderTT.
        setTable(OUTPUT TABLE OrderTT).
    END METHOD.

    METHOD PUBLIC VOID Scenario2( ):
        DEFINE VARIABLE hTT AS HANDLE.
        CREATE TEMP-TABLE hTT.
        hTT:CREATE-LIKE ("Customer").
        hTT:TEMP-TABLE-PREPARE("Customer").
        setTable(INPUT TABLE-HANDLE hTT).
    END METHOD.

    METHOD PUBLIC VOID Scenario3( ):
        DEFINE VARIABLE hTT AS HANDLE.
        CREATE TEMP-TABLE hTT.
        hTT:CREATE-LIKE ("Order").
        hTT:TEMP-TABLE-PREPARE("Order").
        setTable(INPUT TABLE-HANDLE hTT).
    END METHOD.

    METHOD PUBLIC VOID Scenario4( ):
        DEFINE VARIABLE hTT AS HANDLE.
        CREATE TEMP-TABLE hTT.
        setTable(INPUT TABLE-HANDLE hTT).
    END METHOD.

    METHOD PUBLIC VOID Scenario5( ):
        DEFINE VARIABLE hTT AS HANDLE.
        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 generic 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 matching a class hierarchy or interface

ABL searches first for an exact class 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 class 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 class type 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 passed class type 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 class type parameters, because more than one combination of parameter definitions can satisfy the combination of passed class type parameters equally well. In general, if the corresponding parameters of overloaded methods differ by an interface, ABL only accepts an exact class type match to the passed class type parameter.

For more information on defining and passing object reference parameters, see the "Defining an object reference as a parameter" section on page 4–19 and the "Passing object reference parameters" section on page 4–20.

The following examples define method (or constructor) overloads that take different combinations of class type and interface type parameters and how they respond to invocations that pass object references to a variety of related class 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 overloads 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:

```
DEFINE VARIABLE rClassA AS CLASS ClassA.
DEFINE VARIABLE rClassB AS CLASS ClassB.
DEFINE VARIABLE rClassC AS CLASS ClassC.
DEFINE VARIABLE rClassD AS CLASS ClassD.

rClassA = NEW ClassA( ).
rClassB = NEW ClassB( ).
rClassC = NEW ClassC( ).
rClassD = NEW ClassD( ).

DEFINE VARIABLE rOne AS CLASS MonoClass().
rOne = NEW MonoClass().

rOne:setClass( INPUT rClassA ). /* 1 */
rOne:setClass( INPUT rClassB ). /* 2 */
rOne:setClass( INPUT rClassC ). /* 3 */
rOne:setClass( INPUT rClassD ). /* 4 */
```

Execution of the procedure results in the following messages displayed with numbers corresponding to the bolded 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 following examples 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 overloads 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:

```
DEFINE VARIABLE rObjW AS CLASS ClassW.
DEFINE VARIABLE rObjX AS CLASS ClassX.

rObjW = NEW ClassW( ).
rObjX = NEW ClassX( ).

DEFINE VARIABLE rTwo AS CLASS BiClass( ).
rTwo = NEW BiClass( ).

rTwo:setClasses( INPUT rObjW, INPUT rObjW ). /* 1 */
rTwo:setClasses( INPUT rObjX, INPUT rObjX ). /* 2 */
rTwo:setClasses( INPUT rObjW, INPUT rObjX ). /* 3 */
rTwo:setClasses( INPUT rObjX, INPUT rObjW ). /* 4 */
```

The two calls to these setClasses( ) methods resolve as follows, according to the bolded 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 overloads 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:

```
DEFINE VARIABLE rObjX AS CLASS ClassX.

rObjX = NEW ClassX( ).

DEFINE VARIABLE rThree AS CLASS TriClass( ).
rThree = NEW TriClass( ).

rThree:setClasses( INPUT rObjX, INPUT rObjX, INPUT 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 INT test ( ):
        DEFINE VARIABLE rObjB AS CLASS ClassB.
        DEFINE VARIABLE rObjC AS CLASS InterfaceC.

        rObjB = NEW ClassB( ).
        rObjC = NEW ClassF( ).

        setObj( INPUT rObjB ). /* 1 */
        setObj( INPUT rObjC ). /* 2 */
    END METHOD.
END CLASS.
```

The two calls to these `setObj( )` methods resolve as follows, according to the bolded 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 INT test ( ):
        DEFINE VARIABLE rObjB AS CLASS ClassB.
        DEFINE VARIABLE rObjF AS CLASS ClassF.

        rObjB = NEW ClassB( ).
        rObjF = NEW ClassF( ).

        setObj( INPUT rObjB ). /* 1 */
        setObj( INPUT rObjF ). /* 2 */
    END METHOD.
END CLASS.
```

The two calls to these `setObj( )` methods resolve as follows, according to the bolded 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 INT test ( ):
        DEFINE VARIABLE rObjB AS CLASS ClassB.
        DEFINE VARIABLE rObjF AS CLASS ClassF.

        rObjB = NEW ClassB( ).
        rObjF = NEW ClassF( ).

        setObj( INPUT rObjB ). /* 1 */
        setObj( INPUT rObjF ). /* 2 */
    END METHOD.
END CLASS.
```

The two calls to these `setObj( )` methods resolve as follows, according to the bolded 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.

Matching 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 ():
        DEFINE VARIABLE iReturn AS INTEGER.

        iReturn = setUnknowns( INPUT ?, INPUT 12 ).     /* 1 */
        iReturn = setUnknowns( INPUT "Frank", INPUT ?). /* 2 */
    END METHOD.
END CLASS.
```

The two calls to these methods resolve as follows, according to the bolded 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:

```
DEFINE VARIABLE rUObj AS CLASS Unknown.
DEFINE VARIABLE iUnknown AS INTEGER.

rUObj = NEW Unknown( ).

rUObj:setUnknowns( INPUT "Bill", INPUT STRING(?) ). /* 1 */

iUnknown = ?.
rUObj:setUnknowns( INPUT "Steve", INPUT iUnknown ). /* 2 */
```

The two calls to these methods resolve as follows, according to the bolded 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.

Matching 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

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X