Progress®



Object-oriented ABL Developer Guided Journey

OpenEdge®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

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

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Object-oriented ABL concepts....9

Overview of Object-oriented ABL Programming....10
Support for classes in ABL....10
Programming models in ABL....12
What is inheritance?....14
What is encapsulation?....14
What is an interface?....15
What is polymorphism?....15
Glossary of terms....15

Benefits of classes....21

Define a class....21
Determine the package name....22
Determine the class name....22
Define a class in Developer Studio....22
Anatomy of a class....23
Access modes....24
Data members of a class....25
Class variables....25
Class properties....26
Class constructors....29
Class methods....30
Class events....31
Class destructor....32
Use a class....33
Access a data member within a class....33
Access a class method within a class....34
Access class methods from within the class....34
Use other classes....35
Define a variable or property of a class type....35
Create an instance of another class....36
Access a public data member of a class instance....37
Call a public method of a class instance....38
Access a class instance dynamically....39
Delete an instance of a class....39
Test classes....40
Set up the class test 40
Test the class....41

End the test....41

Benefits of inheritance....43

Define a hierarchy....43
Use the New ABL Class wizard to create a derived class....44
Use inheritance....46
Method overloading....48
Delegation....50
Use delegation with classes....50

Benefits of interfaces....53

Define an interface class....54
Use the New ABL Class wizard to create a class that implements an interface class....54
Define a class that uses an interface class....55
Interface hierarchies and inheritance....56
Multiple interfaces in a hierarchy....56
Duplicate members in a hierarchy....58
Use interface hierarchies....59
What is strong typing?....60

Benefits of singletons....61

Define a static data member....62
Define a static constructor....62
Singleton classes and procedures as Data Object resources....63

Benefits of dynamic instances....65

Create a class instance dynamically....66

Benefits of events....67

Define events....67
Define events within a class....68
Use class events....69
Publish class events....70
Subscribe and unsubscribe event handlers....70

Preface

Purpose

The purpose of this content is to introduce the basic object-oriented programming concepts and the benefits of those concepts when building ABL applications. Object-oriented ABL need not replace existing procedural code but rather expand programming options available to ABL developers.

Audience

The audience is those new to ABL object-oriented programming concepts. The expectations is that ABL developers will already understand the procedural ABL developer content presented in the basic and advanced ABL developer guided journeys.

Organization

The organization of this content is based on object-orient concepts of classes, inheritance, and interfaces. With that foundation explained use of singletons, dynamic instance and events are presented. This content is intended to expand the skill set of or procedural ABL developers to include object-oriented solutions to business problems.

Documentation Conventions

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

1

Object-oriented ABL concepts

OpenEdge supports object-oriented programming concepts common in other object-oriented languages such as Java. Using the same underlying ABL, procedural and object-oriented programing models can seamlessly coexist to leverage the best of both approaches.

When you develop an ABL class, you define data members to represent the data and methods to represent the behaviors. An ABL class is a definition of how a user, system, or object will behave at runtime. In your application, you write code to create instances of a class. At runtime, the instance is populated with data and is capable of executing the desired methods, depending on the use case.

For example, you can create a `Customers` class that represents customers and their associated set of orders. An instance of `Customers` is created to represent a customer or set of customers.

For details, see the following topics:

- Overview of Object-oriented ABL Programming
- What is inheritance?
- What is encapsulation?
- What is an interface?
- What is polymorphism?
- Glossary of terms

Overview of Object-oriented ABL Programming

In this video you will learn about the object-oriented capabilities of ABL and how to use Progress Developer Studio to define a simple class.

Support for classes in ABL

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

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

ABL persistent procedures allow you to create and manage objects in which most of the relationships between them are created and managed at run time. ABL classes, on the other hand, contain relationships that you create at compile time, which allows the ABL Virtual Machine (AVM) to automate the management of these relationships at run time. Thus, unlike a persistent procedure, a *class* defines a well-structured *type* (or *object type*) that ABL recognizes at compile time and that you, at run time, can realize as an object that behaves according to the definition of that type.

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

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

Advantages of ABL classes

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

- Classes support a powerful programming model by encapsulating related functionality into objects. The benefit of organized code is especially important for maintenance, where changes or enhancements can be limited to the objects that are affected by the change.
- Classes enhance code reuse. Common code can be put into one class and shared among other related classes. The related classes can provide specialized behavior whenever necessary to extend the initial class functionality, as well as the functionality of other related classes.
- Classes provide strong typing that enables ABL to do compile-time checking in order to catch errors before the code is deployed or executed. This compile-time checking verifies all interfaces for correctness, not just the ones that are accessed during run-time testing, as with procedures.

- The object-oriented support using classes maps much more closely to the constructs used in a service-oriented architecture (SOA) and to those used by modeling tools and other design tools that are increasingly used to provide a basis for application design, and even code generation.
- ABL support for classes includes access to .NET objects, which allows you to work with .NET object types on Windows together with ABL class-based object types. Using the OpenEdge GUI for .NET, you can use .NET form and control objects to create a .NET graphical user interface for all Windows-based GUI ABL clients. Using non-GUI .NET objects, you can also access non-GUI .NET features in all ABL sessions that run on Windows, including non-GUI sessions such as character-mode clients and application server agent sessions. For more information, see *Use GUI for .NET Objects in ABL Applications*.

Foundations of ABL classes

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

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

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

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

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

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

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

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

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

Programming models in ABL

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

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

Procedure-based model

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

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

Class-based model

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

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

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

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

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

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

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

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

Compare procedure-based with class-based models

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

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

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

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

What is inheritance?

In object-oriented programming, classes can inherit data and methods from other classes. So, some or all the data or methods of the super class are inherited by the derived class.

Note that the opposite is not true. A super class cannot inherit the data or methods of its derived class. Derived classes can also define their own data members and methods. Inheritance promotes code reuse.

A derived class can inherit all the non-private data members, properties, methods, and events of a super class.

A super class itself might be a derived class that extends its own super class. This forms a class hierarchy with super class and subclass relationships.

A special type of super class is an abstract class, which is used as a template for a set of derived classes. You cannot create an instance of an abstract class.

What is encapsulation?

Encapsulation is a way of restricting data and methods that are exposed to users of a class.

When you define a class, you specify the type of Access modes on page 24 each data member and method will have at runtime. The benefits of encapsulation are:

- Only what is needed to interact with a class instance is exposed. Other parts of the application are shielded from (and do not need to know about) implementation details or changes.
- The implementation of the behavior is localized.
- Code is easy to maintain.

Note: The best practice is to hide as much data as possible until a valid case is made for making the access mode less restrictive.

What is an interface?

An interface is an ABL class used to define PUBLIC data members and methods that must be implemented by other classes. The interface class does not provide any code, but simply the names and parameters for methods that must be implemented. An interface enforces coding standards for classes that implement the interface. Classes that implement an interface must define and provide code for all data members and methods defined in the interface class, with identical names, parameters, and return types. Multiple classes can implement the same interface, which ensures that they all behave in the same way.

For example, an interface class, `ICustomerInvoice`, could define the required methods related to customer billing functionality such as a method named `SendInvoice()`. Two classes would be defined to implement the `ICustomerInvoice` interface. A `RetailCustomerInvoice` class would implement its version of `SendInvoice()`, which would send an invoice to a customer at home. The `EnterpriseCustomerInvoice` class would implement its version of `SendInvoice()`, which would send an invoice to a business using a purchase order (PO) number.

What is polymorphism?

Polymorphism is a powerful object-oriented feature that reduces the amount of ABL code you need to write.

To use polymorphism, you must write your classes to use either inheritance or interfaces. If you use inheritance, the derived classes must all define the same method defined in the super class. Interfaces already provide this capability as all implementations of an interface class must implement the same method.

Polymorphism allows you to write ABL code to access an instance of a super class or interface class (without worrying about particulars of the derived or implemented classes or methods), but at runtime, ABL dynamically calls the method for the derived or implemented class.

Here is an example with interfaces. Suppose you have defined the `ICustomerInvoice` interface class that specifies that a method named `SendInvoice()` must be implemented. You define the `RetailCustomerInvoice` and `EnterpriseCustomerInvoice` classes to implement ICustomerInvoice. The `SendInvoice()` method in `RetailCustomerInvoice` uses a home address format for sending the invoice. The `SendInvoice()` method in `EnterpriseCustomerInvoice` uses a business address format with a PO number for sending the invoice. Polymorphism enables you to write code to process all invoices, regardless of whether they are for retail or enterprise customers. The code you write calls the `SendInvoice()` method for the instance of the `ICustomerInvoice` interface class. It does not have to determine whether an instance is for a retail customer or enterprise customer. ABL dynamically chooses the correct method to call at runtime.

Glossary of terms

A glossary of terms introduces the object-oriented concepts.

Table 1: Glossary of terms

Term	Meaning
Access mode	Controls where and how a class member can be accessed.

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

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

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

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

2

Benefits of classes

Classes represents a user-defined data type whose characteristics are defined by class members, including a set of class `data members` and `properties` that define class data, and a set of class `methods` and `events` that define class behavior. In addition to class members, a class definition can include special optional methods (one or more `constructors` and one `destructor`). It can also include class-scoped handle-based objects that provide PRIVATE resources to the class, `function` prototypes for user-defined functions referenced by the class, and `triggers` to handle `events` for widgets and other `handle-based` objects defined by the class.

For details, see the following topics:

- Define a class
- Anatomy of a class
- Use a class
- Use other classes
- Test classes

Define a class

ABL classes are used to represent users, objects, and systems, which are business entities in an enterprise application. A part of your job as a developer is to come up with a set of classes to model your application. Most of the code that you will write to implement a modern application are implemented as classes.

A class contains data members and methods that are used to provide the behavior for the class and to access the data members of the class. An instance of a class is an in-memory object that contains values for the data members. There can be several instances of a class at runtime, each with its own data.

You will most likely develop a class iteratively. That is, you will start by defining the class and its initial behavior. Then you will modify the class as you further develop the layers of the application or as the application requirements change.

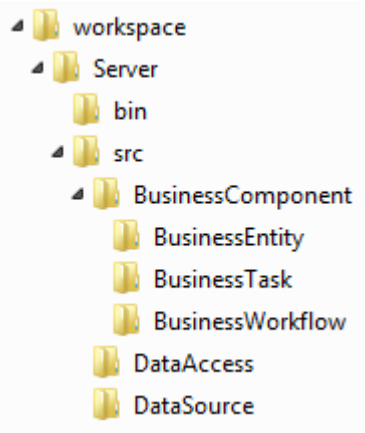
See also:

- 'Define classes' in *Develop Object-oriented ABL Applications*
- 'CLASS statement' in *ABL Reference*

Determine the package name

A package is a directory path in which a class file is located. It maps to the organization of your code within your projects. ABL uses the PROPATH and the package to locate your classes.

For example, in this directory structure where `Server/src` is in the PROPATH, `BusinessComponent.BusinessEntity` is a valid package name:



Determine the class name

The name of a class should represent its purpose. It must begin with a letter and it can contain letters, numbers, underscores, or hyphens. It must not contain spaces or periods. The name of the class must be the same as the name of the file with the `.cls` extension that contains the class definition.

For example, here are two class files—`Emp.cls`, which contains the definition of the `Emp` class that models an employee, and `Dept.cls`, which contains the definition of the `Dept` class that models a department.

Define a class in Developer Studio

The **New ABL Class wizard** helps you define an ABL class. You start the wizard in the workspace directory where you want to create the ABL class file. This is typically in a directory underneath the `src` directory for a project.

In the wizard, the package is automatically set by the location in the directory structure where you start the wizard. You must specify the class name. As a best practice, you should select the method stubs for the default constructor and destructor. In addition, you should retain the default error-handling generated by the wizard.

It is always a good practice to provide information that describes the class.

When the New ABL Class wizard ends, the class file is created in the package location in the workspace and opens in the editor.

Example

Here is an example of the code that the **New ABL Class wizard** generates for the *Emp* class. A `using` statement is the first statement in this file. It specifies that this class will use all the built-in classes defined in the `Progress.Lang` package. The *block level* error-handling is then specified. The name of the class is `Emp` and the name of the file is `Emp.cls`. This file resides in the `Enterprise.HR` package, which maps to the `Server/src/Enterprise/HR` directory in the workspace.

The class definition begins with the `class` keyword, followed by the fully qualified class name, which includes the package. The class definition ends with the `end class` statement. This class currently does not have any data defined. It has the default `public constructor` and `destructor` defined, but not implemented.

```
//----------------------------------------
  File : Emp
  Purpose : Used by HR component
  Syntax :
  Description : Employee of the company
  Author(s) : Created : Fri Sep 12 14:45:07
  Notes :
//----------------------------------------
using Progress.Lang.*.
block-level on error undo, throw.
class Enterprise.HR.Emp:

  constructor public Emp ( ):
    super ().
  end constructor.

  destructor public Emp ():
  end destructor.

end class.
```

Anatomy of a class

An ABL class consists of definitions of `data members`, definitions and implementations of `constructors`, `methods`, and a `destructor`. `Constructors` are used to create class instances at runtime. The `destructor` cleans up resources when a class instance is deleted. Learn more about these topics.

- Data members of a class on page 25
- Class variables on page 25
- Class properties on page 26
- Class constructors on page 29
- Class methods on page 30
- Define events on page 31
- Class destructor on page 32

For more information, see 'Class definition file structure' in *Develop Object-oriented ABL Applications*.

Access modes

Access modes provide the ability to hide or expose class members to other parts of an application based on how the class member is defined. Class members include variables, properties, methods and events which are common to object-oriented programming languages and ABL resources like buffers, temp-tables, datasets and queries. The available access modes from most to least restrictive are:

- `PRIVATE` - Limits access to the class.
- `PACKAGE-PRIVATE` - Limits access to the class and any class within its package.
- `PROTECTED` - Limits access to the class and subclasses of the class.
- `PACKAGE-PROTECTED` - Limits access to the class, any class within its package, and to any subclass that inherits from the class.
- `PUBLIC` - Accessible to all.

In ABL, a package is the directory structure, relative to the `PROPATH` leading up to the class itself, as specified in the `CLASS` statement. This directory path identifies the location of the file that defines a class, interface, or enumeration. `PACKAGE-PRIVATE` and `PACKAGE-PROTECTED` restrict access at the package level.

The following table identifies whether a specific access mode exposes a class member to other parts of the application.

Access mode	Class	Subclass	Package	All
PRIVATE	Y	N	N	N
PACKAGE-PRIVATE	Y	N	Y	N
PROTECTED	Y	Y	N	N
PACKAGE-PROTECTED	Y	Y	Y	N
PUBLIC	Y	Y	Y	Y

See the 'ABL Syntax Reference' in *ABL Reference* for available access modes and default values for each class member.

See the 'ABL package-level access modes' video in *OpenEdge Videos* for more information on the `PACKAGE-PRIVATE` and `PACKAGE-PROTECTED` access modes.

Best practices

A commonly held object-oriented best practice is making code open to extension and closed to modification. As a developer, picking the right access mode is critical. The compiler enforces access modes so if access is denied to other parts of the application then a conversation between developers is required to relax the restriction by changing the access mode. The general guideline is to begin with the most restrictive access mode and build valid use cases for relaxing the access restrictions.

1. Always be explicit in defining an access mode.
2. Use `PRIVATE` to ensure class members can only be accessed by the class.
3. If a class member is required by other classes in the class hierarchy, use `PROTECTED`.

4. Organize class, interfaces and enumerations in named packages for components, frameworks or libraries. Use PACKAGE-PRIVATE or PACKAGE-PROTECTED access modes to limit access based on the named package.
5. Avoid PUBLIC, unless the intent is for anyone from anywhere to access the class member.

Data members of a class

Data members of a class hold runtime values for a class instance. Whenever an instance of a class is created, each instance has its own values for the data members of the class. The values for the data members of a class instance are assigned when the instance is created or are assigned by the methods of the class.

You can define four types of data members for an ABL class:

Data member type	Description
variable	Holds a value of a built-in type or user-defined type. A best practice is to define variables as PRIVATE or PROTECTED so that they can only be accessed by methods of the class.
property	Holds a value of a built-in type or user-defined type. A property is like a variable, but it allows you to customize how its values will be read or set. A best practice is to define properties as PUBLIC for reading and PRIVATE or PROTECTED for setting. This enables you to control how data is accessed by code from other parts of your application.
temp-table	Holds aggregate data such as a database record in a relational database table. A temp-table typically holds data of multiple types. A best practice is to define a temp-table as PRIVATE.
dataset	Holds data that represent a set of temp-tables.

See also:

- 'Define data members' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming' in *Develop Object-oriented ABL Applications*

Class variables

You define a variable for a class is similar to how you define a variable for a procedure. The main difference is in the access mode you specify for the variable.

Here is the simplified ABL syntax to define a variable data member:

```
VAR access-mode type-name variable-name.
```

Syntax Element	Description

access-mode	Specifies whether the data member will be available to other parts of an application. As a best practice, you should define as `PRIVATE` or `PROTECTED`. For more information, see Access modes on page 24.
type-name	Can be one of the ABL data types such as integer or character or it can be a user-defined type.
variable-name	Must begin with a letter and it can contain letters, numbers, underscores, or hyphens. It must not contain periods or spaces.

Here is a variable data member defined for the Dept class. Every instance of a Dept class will hold its own values for this data member. Since this data member is `PRIVATE`, it will be accessible only by the methods of this class.

```
class Enterprise.HR.Dept:
//used for keeping track of how many employees
//are in the set of employees for the department
//used internally within the class.

  var private integer NumEmployee.

  // rest of class definition

end class.
```

See also:

- 'VAR statement' and 'DEFINE VARIABLE statement' in *ABL Reference*
- 'Local variables and other data elements of methods' in *Develop Object-oriented ABL Applications*

Class properties

Like a variable, a property holds a value of a built-in type or user-defined type, but it also has built-in accessor methods `get()` and `set()` that are used to access the data in the property. This is useful because you can define a property to be read by another part of the application, but only to be set by methods of the class. In addition, you can add code to the `get()` and `set()` accessor methods to customize how the property value will be read and set. For example, you can implement the `set()` accessor method to transform a `PostalCode` value into a standard format.

Note: A best practice is to define properties as `PUBLIC` for reading and `PRIVATE` or `PROTECTED` for setting. `PACKAGE-PRIVATE` and `PACKAGE-PROTECTED` allow package-level protection. This enables you to control how data is accessed by code from other parts of your application. Most of the data members you add to an ABL class are defined as properties.

See also:

- Define a data member as a property on page 27 in *Object-oriented ABL Developer Guided Journey*
- 'DEFINE PROPERTY statement' in *Develop Object-oriented ABL Applications*

Define a data member as a property

You define a property in the same way that you define a variable, except that you use the keyword property. In addition, you specify the get and set accessors for the property. Optionally, you can customize the behavior of the `get()` and `set()` accessor methods.

When you define a property, you specify visibility for the property, and for the `get()` and `set()` accessor methods. The visibility for `get()` and `set()` must be the same or more restrictive than the visibility for the property.

Note: You must define `get()` before `set()`, otherwise your code will not compile.

Here is the simplified syntax for defining a property:

```
define[<visibility>] property <property-name> as
  <type-name> [no-undo]
<visibility> get
  [(): <body of get that returns property> end get].
<visibility>] set
  [(<parameter>): <body of set that sets property> end set].
```

Syntax Element	Description
visibility	`Public`, `package-protected`, `protected`, `package-private`, or `private`.
property-name	The name of the data member.
type-name	A built-in ABL type or a user-defined type.
parameter	The value that is used to set the property. It must match the type of the property.

See also:

- 'DEFINE PROPERTY statement' in *Develop Object-oriented ABL Applications*

Use the Add Property wizard

In your class file, you can use Developer Studio to help you generate code for a property definition. The **Add Property** wizard generates most of the code that you need to define a property for your class.

Follow these steps to generate code for a property definition in Developer Studio:

Step	Action
1.	Anywhere in your source file, right-click and then select **Source > Add Property….**The *Add Property* wizard opens.

2.	Enter the name of the property.
3.	Specify whether the property will be *public*, *protected*, or *private*. The best practice is to make your properties *public*.
4.	Select the data type for the property using the drop-down list.
5.	If the property will be an array, select the **Extent** box and optionally specify the number of elements that the array can hold.
6.	If the property will have an initial value, specify it. It must match the data type.
7.	For the **Get accessor**, leave **Readable** checked, select **Insert implementation** if you plan to add custom code, and specify the visibility for reading this property.
8.	For the **Set accessor**, leave **Writable** checked, select **Insert implementation** if you plan to add custom code and specify the visibility for setting this property.
9.	Select an insertion position for the property. A best practice is to select **Last Property**.
10.	Click **Generate**. This will create the code to define the property. If you specified that you will insert custom code, you will need to eventually add ABL code to these blocks in the property definition.

Define properties example

Here is an example of some properties that are part of the *Emp* class definition. The *Address* property is defined as public, as are its accessors. So other parts of the application will be able to read and set the *Address*.

The *PostalCode* property is also defined as public, as are its accessors. The set accessor, however, has a placeholder for a custom implementation. You can add code here to transform a value for this property.

The *JobTitle* property is also defined as public, but only methods of this class can set its value. Other parts of the application can read this property, but cannot set its value.

```
class Enterprise.HR.Emp:
  define public property Address as character no-undo
    get.
 set.
  define public property PostalCode as character no-undo
    get.
 set(input arg as character):
   // code to standardize the PostalCode
 end set.
  define public property JobTitle as character no-undo
    get.
 private set.

//rest of class definition....
```

Class constructors

A class `constructor` is a special method that creates an instance of a class. The constructor returns an instance of the class.

A class can have more than one constructor, each with a different parameter list. This is useful when you need to create instances in different ways for different parts of the application. For example, you could have one Customer constructor that takes no input parameters that is called by one part of the application. In this case an “empty” *Customer* instance is created and another part of the application initializes the data members of the `Customer` instance. You can also have a constructor that takes values for the `Customer` when it is created. When the constructor executes, it will populate the data members for this instance with these values.

Note: A best practice is to define the constructor as `PUBLIC` so that other parts of the application can create instances of the class.

See also:

- 'Comparison with procedure-based programming main blocks of persistent procedures' in *Develop Object-oriented ABL Applications*

Define a class constructor

A class definition must contain at least one constructor. The syntax for defining a constructor for a class is:

```
constructor <access-mode> <class-name>
               ( [input <parameter> as <type-name>][,…]):
               <body of constructor>
               end constructor.
```

Syntax Element	Description
access-mode	Specifies whether the constructor will be available to other parts of an application. A best practice is to define the constructor as `PUBLIC` so that other parts of the application can create instances of the class. For more information, see Access modes on page 24.
class-name	Must match the name of the class in the class definition
type-name	Can be one of the built-in or user-defined types.
parameter	The value that is used to set the property. It must match the type of the property.

You can use the *New Constructor* wizard to generate the code for a constructor (right-click, then select **Source > New Constructor**). If the constructor takes parameters, you must manually add them to the definition.

There are two common design patterns for constructors:

1. Use a no-argument constructor and then use a PUBLIC method to initialize the data members of the instance.
2. Use a constructor with parameters and use the parameters to initialize the data members of the instance in the body of the constructor.

Note: If you use the *New Constructor* wizard to generate the constructor, the first statement that is generated is the call to *super()*. This means the constructor for the super class will be called first. It is safe to keep this statement in your constructor, even if you have not defined a super class.

See also:

- 'CONSTRUCTOR statement' in *ABL Reference*

Class methods

Class methods are used to perform the behavior and functionality of the class. When a class instance is created, memory is allocated for the data members of the instance. The methods for the class operate on the data members of the class instance. Some methods are PRIVATE or PROTECTED and can only be called by other methods in the class. Other methods in a class may be PUBLIC, in which case they can be called by other parts of the application.

PACKAGE-PRIVATE and PACKAGE-PROTECTED provide similar restrictions at the package level. For more information, see Access modes on page 24.

You may want to have multiple methods in a class that have the same name but have a different parameter list or return type. Creating methods with the same name is called overloading methods. Overloaded methods make your classes more flexible for users of the class.

See also:

- 'Comparison with procedure-based programming internal procedures and user-defined functions' in *Develop Object-oriented ABL Applications*

Define a class method

Like data members, a method has visibility defined so that the method can be called within a class or from outside the class. A method can have zero or more parameters. Regardless of whether a method has parameters, it can return a value.

Notice that a method begins with a *method* statement and ends with an *end method* statement.

Note: A best practice is to also use a return statement for a method, even if it returns void. Returning void means the method returns no value. This is different from an output parameter that is used to return a value in the parameter list for a method.

Here is the simplified syntax for defining a class method:

```
method [visibility] {<return-type> | void } <method-name>
  ( [<parameter-use> <parameter> as <type-name> ][,…]):
  <body of method>
  return [return-value].
end method.
```

Syntax Element	Description

visibility	*Public*, *private*, or *protected*.
return-type	Can be a built-in or user-defined type. If void is specified, the method returns no value.
parameter-use	Can be one of *input*, *output*, or *input-output*.
parameter	The name for the parameter.
type-name	An ABL built-in type or a user-defined type.
body of method	The ABL code necessary to implement the functionality of the method. If the method specifies a return type, the body of the method must be written to return a value of the return type specified in the definition of the method.
return-value	If the method returns a value, the value must match the type specified for return-type.

See also:

- 'METHOD statement' in *ABL Reference*

Class events

An *event* is a condition—typically, a behavior and/or data change—that you can identify in your application. You then write code to *publish* the event when the event condition occurs. The class where you identify the condition and publish the event is referred to as the *publisher* class.

Subscribers of the event can specify an action, called an *event handler*, that executes when the event is published. The event handler is a *class method* in the *subscriber* class.

Define events

A class event is a class member of the publisher class that you define. Here is the short syntax for defining a class event in the publisher class.

```
define [access-mode] event Event-Name
  signature void (parameter, [parameter…]).
```

Syntax Element	**Description**

access-mode	Specifies whether the method will be available to other parts of an application. For more information, see Access modes on page 24. For a class event, the access mode specifies where you can subscribe a handler for the event using the subscribe( ) method.
Event-Name	The name of the event. This name must be unique among all other events, properties, and variable data members in the application.
signature void (parameter, [parameter..])	The signature of the class event. This indicates the required signature for any event handler that subscribes to the event.

See also:

- 'Define class events' in *ABL Reference*
- 'DEFINE EVENT statement' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming named events' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming PUBLIC and PRIVATE events' in *Develop Object-oriented ABL Applications*

Class destructor

A destructor is an optional method that runs when an instance of a class is deleted or removed from memory. You define a destructor if your class needs to perform some special processing when the instance is deleted. For example, let's assume that the Dept class contains a temp-table of its employees. If an employee leaves the department, the Emp instance is deleted, but you would also want to remove the reference to the Emp instance in the Dept instance.

See also:

- 'Comparison with procedure-based programming cleanup' in *Develop Object-oriented ABL Applications*

Define a class destructor

Here is the simplified syntax for defining a destructor for a class:

```
destructor public <class-name> ():
  <body of destructor>
end destructor.
```

Syntax Element	Description
class-name	Must match the name of the class in the class definition
body of destructor	The ABL code necessary to implement the functionality of the destructor

For more information, see 'DESTRUCTOR statement' in *ABL Syntax Reference*.

Use a class

Until now, you have learned how to write ABL code to define the parts of a class including data members, constructors, methods, and a destructor. When you develop an ABL class, you must provide ABL code for the bodies of your constructors, methods, and the destructor.

In these topic, you will learn how to access data members and call methods within a class:

- Access a data member within a class on page 33
- Access a class method within a class on page 34

You can access `PUBLIC` data members and methods of a class instance from any part of your application. To do this, you must create or obtain an instance of the class you want to access. Once you have the reference to the instance, you can access any of its `PUBLIC` data members and methods.

For example, a `Dept` class may have to access instances of the `Emp` class because the `Emp` class maintains information about employees.

Note: You can create instances of other classes and access them from procedures as well as from constructors and methods.

Access a data member within a class

In the constructors, methods, or destructor of a class, you can write code to directly access the data members of the class. You access the data members by name, but you can also type the keyword this-object: in the editor. When you do so, the editor provides a drop-down list of all available data members in the class for you to choose from. This makes it easier to find the data member name that you want to use when you are working with a large class.

Here is the syntax for using a data member from within a class:

```
[this-object:]<data-member>
```

Syntax Element	Description
data-member	The name of a data member in the class

See also:

- 'THIS-OBJECT system reference' in *ABL Syntax Reference*

Initialize data members in the constructor

Here is part of the constructor for the `Dept` class. It takes values from the parameters to the constructor and uses them to set the values of the data members of the class. Once this instance has been created, it is available for adding employees and performing other behaviors of the class.

Notice that we used the `this-object:` prefix to allow the editor to help us select the `ExpenseCode` data member name.

```
constructor public Dept (  input pDeptName as character, input pMaxNumEmployees as
integer, input pExpenseCode as character ):
  super ().
  assign DeptName = pDeptName /* add code here to size the array using  pMaxNumEmployees
 */
  this-object:ExpenseCode = pExpenseCode.
end constructor.
```

Access a class method within a class

In the constructors, methods or destructor of a class, you can access any other method defined in the class. If a method returns a value, you can use that value anywhere in your code.

Here is the syntax for calling a class method from within a class:

```
:<method>( [ <parameter>][,…])
```

Syntax	Description
`method`	The name of the method in this instance of the class.
`parameter`	Zero or more values that are passed to the method at runtime. You must specify the same number of parameters, in the same order, and with the same types as in the definition of the method.

Access class methods from within the class

When you define a method, it can be defined either to return a value or not to return a value (using `void`).

If a method returns a value, you can use it like a function call in your code. That is, it can be used in any valid ABL statement that expects that type of value. Here is an example of using the `GetName()` method, which returns a `character` value:

```
EmpInfo = GetName() +
  " " +  Address +
  " " + PostalCode +
  " " + "Job Title: " + JobTitle +
  " " + "Vacation Hours: " + string(VacationHours).
```

If a method does not return a value, you must call it in a separate ABL statement. Then you typically use any returned output parameters in your code.

```
GetValue(3,result).message result.
```

In either case, you specify the name of the method followed by the parameter list.

Use other classes

You can access PUBLIC data members and methods of a class instance from any part of your application. To do this, you must create or obtain an instance of the class you want to access. Once you have the reference to the instance, you can access any of its PUBLIC data members and methods.

If you are using packages and package-level access modes, then the instance reference allows you to access PACKAGE-PRIVATE and PACKAGE-PROTECTED members from other classes within the same package.

For example, a Dept class may have to access instances of the Emp class because the Emp class maintains information about employees.

Note: You can create instances of other classes and access them from procedures as well as from constructors and methods.

Define a variable or property of a class type

To access the PUBLIC data members and methods of a class instance, you must first define a variable or property that will hold the reference to the instance. The type you specify is a user-defined type, which is the name of the class. When you define this data member, you specify its access mode just as you do for other data members. If you want other parts of the application to access this data member, you define it as PUBLIC. Otherwise, you define it as according to your access mode best practices.

Here is the simplified syntax for creating a variable or property that holds the reference to a class instance:

```
define [<access mode>] variable <name> as <class-name> [no-undo].
define [<access mode>] property <name> as <class-name> [no-undo]
  <access mode> get  [():
    <body of get that returns property>
  end get].
  <access mode> set [(<parameter>):
    <body of set that sets property>
  end set].
```

Syntax Element	Description
access mode	Specifies whether the variable or property will be available to other parts of the application. See Access modes on page 24.
name	The name of the variable or property that you will use to reference the class instance
class-name	The name of the class previously specified in a using statement
parameter	The name for the parameter

Here is the definition of the DeptRef variable in the Company class. It will hold a reference to an instance of a Dept class.

```
define private variable DeptRef as Dept no-undo.
```

See also:

- 'Comparison with procedure-based programming variables' in *Develop Object-oriented ABL Applications*

Create an instance of another class

When you create an instance of another class, you call the constructor for the class using the ABL new keyword. The constructor returns a value, which is the reference to the newly created class instance. You must assign the reference to the class instance to the variable or property whose type is the name of the class.

Note: The class you call may have multiple constructors, each with its own set of parameters. You must make sure that you call the correct constructor by using the expected parameter list for that constructor.

Here is the simplified syntax for calling the default constructor for a class with no parameters:

```
<defined-name> = new <class-name>().
```

Syntax Element	Description
defined-name	The name of a previously defined variable or property that will hold the reference to the class instance
class-name	The name of the class

Suppose the application has a Dept class that is used to hold data for employees of a department. A user of the Dept class can call the AddEmployee() method to add an employee to the department. Here is the AddEmployee() method of the Dept class that takes as input all of the data necessary to initialize an Emp instance. It defines a variable, Empl, that holds the reference to the Emp instance and creates an instance of the Emp class using the new keyword to call the constructor. In this same statement, the reference to the Emp class instance returned by the constructor is assigned to Emp.

After it is created, the Emp instance can be accessed and initialized.

```
method public void AddEmployee (input pEmpNum as integer,
  input pFirstName as character,
  input pLastName as character,
  input pAddress as character,
  input pPostalCode as character,
  input pPhones as character extent 3,
  input pVacationHours as integer,
  input pJobTitle as character):

    var Emp Empl = new Emp ().
    // rest of method to initialize the values for the Employee
    // instance using the parameters passed into this method
    // then add reference to Employee instance to array of Employees
  return.
end method.
```

Access a public data member of a class instance

After you have created an instance of another class, you can access any PUBLIC data member in the class instance. Here is the simplified syntax for accessing a PUBLIC data member of a class:

```
<ref>:<data-member>
```

Syntax Element	Description
ref	The variable or property that holds the reference to the instance of the class.
data-member	The name of the data member in the class. This data member must be PUBLIC.

Suppose you have defined some of the data members of the Emp class to be PUBLIC. For example, the phone numbers, address, and postal code data members for an Emp are defined as PUBLIC because you want to be able access them from the Dept class and from other parts of the application. Here, in the AddEmployee() method of the Dept class, we have created the instance of the Emp class and assigned it to Emp. The PRIVATE data members of the Emp class must be set by calling the Initialize() method of the Emp class. The PUBLIC data members of the Emp class can be set from the Dept class. In this code, we assign values to the Address, Phones, and PostalCode data members of the Emp class instance using the reference to the Emp instance.

```
method public void AddEmployee
  (input pEmpNum as integer, input pFirstName as character,
  input pLastName as character, input pAddress as character,
  input pPostalCode as character,
  input pPhones as character extent 3,
  input pVacationHours as integer, input pJobTitle as character ):

  var Emp Empl = new Emp ().
  // call Initialize() method to initialize the private data
  // members for the Emp Instance
  assign ttEmployee.FirstName = pFirstName
    ttEmployee.LastName = pLastName
    ttEmployee.EmpRef = pEmpl
    NumEmployees = NumEmployees + 1.
  return.
end method.
```

See Also:

- 'Comparison with procedure-based programming variables and data elements' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming user-defined functions' in *Develop Object-oriented ABL Applications*

Call a public method of a class instance

After you have created an instance of another class, you can call any PUBLIC constructor, method, or destructor defined in the class. If a method returns a value, you can use that value anywhere in your code. Here is the simplified syntax for calling a method of a class:

```
<ref>:<method>( [ <parameter>][,…]).
```

Syntax Element	Description
ref	The variable or property that holds the reference to the instance of the class.
method	The name of a method defined for the class.
parameter	Zero or more values that are passed to the method at run time. The types of the parameters must match the parameters defined for the method in the class definition.

Note: You can call any PACKAGE-PRIVATE or PACKAGE-PROTECTED constructor or method defined in the class if they are in the same package.

Here is the AddEmployee() method of the Dept class. You call the PUBLIC Initialize() method in the Emp class using the reference to the Emp instance. When we call Initialize(), you pass values that match the number and type of parameters defined in the Initialize() method of the Emp class. Note that the values passed into the Initialize() method were passed in to the AddEmployee() method of the Dept class.

```
method public void AddEmployee (input pEmpNum as integer,
  input pFirstName as character,
  input pLastName as character,
  input pAddress as character,
  input pPostalCode as character,
  input pPhones as character extent 3,
  input pVacationHours as integer,
  input pJobTitle as character ):

    var Emp Empl = new Emp ().
    Emp:Initialize(pEmpNum,pFirstName,pLastName,pAddress,
   pPostalCode, pPhones, pVacationHours, pJobTitle).
    create ttEmployee.
    assign ttEmployee.FirstName = pFirstName
   ttEmployee.LastName = pLastName
   ttEmployee.EmpRef = pEmpl
   NumEmployees = NumEmployees + 1.
 return.
end method.
```

See also: 'Comparison with procedure-based programming invoking internal procedures' in *Develop Object-oriented ABL Applications*

Access a class instance dynamically

In ABL, you can write generic application code that accesses a class instance where the type of the instance is not known until runtime. This can reduce the amount of code you write. You must, however, anticipate the possible types your code can expect at runtime.

You use the `cast()` function to transform a `Progress.Lang.Object` into a particular type at runtime. Once the type has been transformed, you can access any of its public methods or data members. You can call the `cast()` function on class instances that are created dynamically with dynamic-new or statically with new.

Here is the syntax for the `cast()` function.

```
<typed-object-reference> = cast (<object-reference>,<class-name>).
```

Syntax Element	Description
`typed-object-reference`	A property, variable, or field in a temp-table of type class-name
`object-reference`	A property, variable, or field in a temp-table of type `Progress.Lang.Object`
`class-name`	The fully qualified class name where the definition of the class is known only at runtime

Use the cast function to access a class instance dynamically

Here is a simple example of using the ABL `cast()` function. A temp-table, `ttEmployee`, has been defined. It contains `employee` records.

A class instance is created dynamically in the `GetEmployee()` method of the `Dept` to find employees based on the first and last name in the `ttEmployee` temp-table. If an employee is found, it should `cast` the `EmpRef` field and `return` the `Emp` instance. If an employee is not found, it should return `unknown`. Once the object reference is cast to the correct type, you can execute any method of that instance.

```
method public Emp GetEmployee(input pFirstName as character, input pLastName as character):

  find first ttEmployee where ttEmployee.FirstName = pFirstName and
    ttEmployee.LastName = pLastName.
  if available (ttEmployee)
    then
      return cast(ttEmployee.EmpRef,Emp).
    else
      return ?.
end method.
```

Delete an instance of a class

When you finish working with an instance of another class, you should delete it to free up the resources that were used by the class. When a class instance is deleted, it calls the destructor, if the class has one defined.

```
delete object <ref>[no-error].
```

For more information, see 'DELETE OBJECT statement' in *ABL Syntax Reference*.

Test classes

Whenever you develop or revise a class, you should test it. A common way to test a class is to write a procedure that creates instances of the class and executes every constructor and method, and the destructor. It is useful to keep a record of your test results, typically in a log file. This enables you to compare test results when you modify a class. '

Note: A best practice is to create a separate project for your test procedures that has the same folder structure as your application.

---

1. Write code to set up the class test.
   a. Specify a `using` statement for the class for the test.
   b. Define a variable of the class type.
   c. Open the output file for writing test results.
2. Test the class.
   a. Create multiple instances of the class, assigning its reference to the variable.
   b. Make sure every constructor is called.
   c. Pass a range of values to fully test the constructors.
   d. Call each method of the class using the reference.
   e. Pass a range of values to fully test the methods.
   f. Write relevant data to the output file.
3. End the class test
4. Delete each instance when you no longer require it.
5. Close the output file.

Set up the class test

In a test procedure, you must define a variable to hold the reference to the class instance. Since you are defining a variable of a class type, you must specify a using statement for that class at the top of your procedure.

Here is the setup code for testing the `Emp` class. It first specifies the `Enterprise.HR.Emp` class in a using statement. Then, it defines the variable, `EmpInstance` that will hold a reference to the `Emp` class instance. Next, it specifies the file where test output will be written.

```
block-level on error undo, throw.
using Enterprise.HR.Emp.
// define the variable for holding the Emp instance
var Emp EmpInstance.
// set up the file for writing data
output to "/testResults/log/testEmp.out".
// more of test procedures
```

Test the class

If your class contains more than one version of the constructor, then you should write code to test all of them. If a class constructor takes parameters, you must add code to create instances that test the range of values for that constructor. For example, you must write code to pass in values that are valid and values that are not valid.

Next, in your test procedure, you write code that tests the instance created by a constructor. Depending on the values of the data members of the instance, you must call the relevant methods to confirm that it executes correctly. Along the way, you can use message statements to write data to the output file.

In this example, we call the default constructor to create an instance of the `Emp` class. Next, we assign values to the `Phones` elements and then call the `Initialize()` method of the `Emp` class, passing values to initialize the instance. The message statement writes information about the `PUBLIC` data members of the instance after the `Initialize()` method has been called.

After this, if there is more to test, you can use the same instance to test other `PUBLIC` data members and methods of the class.

```
block-level on error undo, throw.
using Enterprise.HR.Emp.

// define and initialize the array for holding the phone numbers
var char[3] Phones = ["617-284-5937","508-394-3928","508-294-3927"].

// set up the file for writing data
output to "/testResults/log/testEmp.out".

// Create and initialize the variable for holding the Emp instance
var Emp EmpInstance = new Emp().
EmpInstance:Initialize(99,
  "John",
  "Doe",
  "123 Main Street",
  "01730",
  Phones,
  50,
  "Senior Developer").

Message "*********testInitialize()************" skip
  EmpInstance:FirstName " "
  EmpInstance:LastName " , "
  EmpInstance:JobTitle skip
  "Emp # " EmpInstance:EmpNum "- Vacation hours: "
  EmpInstance:VacationHours skip
  EmpInstance:Address " " EmpInstance:PostalCode skip
  "Phones: " EmpInstance:Phones[1] " " EmpInstance:Phones[2]
  "" EmpInstance:Phones[3] " ".
//  more of test procedure
```

End the test

When you finish testing or if you want to test a different constructor, you should first delete the instance using the following syntax:

```
delete object <instance name>.
```

Here is all the code for our example test procedure for the Emp class. Notice at the bottom of the file that we delete EmpInstance and close the output file.

```
block-level on error undo, throw.
using Enterprise.HR.Emp.

// define the array for holding the phone numbers
var char[3] Phones = ["617-284-5937","508-394-3928","508-294-3927"].

// set up the file for writing data
output to "/testResults/log/test-Dept.out".

// create and initialize an Emp instance
var Emp EmpInstance = new Emp().
EmpInstance:Initialize(99,
  "John",
  "Doe",
  "123 Main Street",
  "01730",
  Phones,
  50,
  "Senior Developer").
Message "**********testInitialize()************" skip
  EmpInstance:FirstName " "
  EmpInstance:LastName " , "
  EmpInstance:JobTitle skip
  "Emp # " EmpInstance:EmpNum "- Vacation hours: "
  EmpInstance:VacationHours skip
  EmpInstance:Address " "
  EmpInstance:PostalCode skip
  "Phones: " EmpInstance:Phones[1] " "
  EmpInstance:Phones[2] "" EmpInstance:Phones[3] " ".

delete object EmpInstance.
output close.
```

3

Benefits of inheritance

Inheritance allows you to create new classes based on existing classes. This supports code reuse because subclasses can leverage data member definitions and methods available in the superclass. Additionally, you can add new functionality and/or overriding methods to a change the behavior for a specific subclass. This is meant to model inheritance in the physical world.

For details, see the following topics:

- Define a hierarchy
- Use inheritance
- Method overloading
- Delegation

Define a hierarchy

A derived class can inherit ***non-private*** data members, properties, and methods from a super class. The definition of non-private means an access mode of `PROTECTED` or `PUBLIC`. If you are using packages the definition of non-private is extended to include `PACKAGE-PROTECTED` or `PACKAGE-PRIVATE` if the derived class and super class are in the same package. A super class itself can have a super class. In this way, a derived class can have a chain of super classes forming a class hierarchy, in which each super class inherits from the super class above it. Each derived class inherits the non-private class members from all super classes in its class hierarchy.

To define a top-level super class, you simply define it as a class. To define a class that inherits from another class, you use the `INHERITS` keyword.

Here is the syntax for defining a class that inherits from another class:

```
class <derived-class-name> inherits <super-class-name>:
```

In a derived class that inherits from another class, you can define a method to override a method in its class hierarchy. For example, you may want to define specialized behavior for a method in a derived class. The derived method must have the same or less restrictive visibility as the overridden super class method. For example, you can override a PROTECTED super class method with a PUBLIC or a PROTECTED derived class method. The derived class method should also have the same method name, return type, and number of parameters. Each corresponding parameter must be of the same mode (input, output, or input-output) and of the same data type as the method it overrides.

Note: To a derived class you can also add additional methods that are not defined in the super class.

To indicate that a method overrides the super class method, you use the override key word. Here is the syntax for overriding a super class method:

```
method [acces-mode] override <return-type> <method-name>( ):
```

Note: If you use PACKAGE-PRIVATE or PACKAGE-PROTECTED access modes those restrictions apply. For more information see, Access modes on page 24.

Use the New ABL Class wizard to create a derived class

In Developer Studio, you can create a derived class using the New ABL Class wizard. The New ABL Class wizard helps you locate the super class you want to use. After you select the super class, the New ABL Class wizard generates the class definition statement with the inherits key word and the name of the super class.

Follow these steps in Developer Studio to create a derived class:

Step	Action
1	Right-click on the folder you want to add a derived class and then select New > ABL Class. The New ABL Class wizard opens.
2	Enter the name of the derived class.
3	Click Browse in the Inherits field to select the super class. The Super Class Selection window opens.
4	Type the filter text to find the super class.
5	Click OK.

6	If you want the class to have a default constructor and/or destructor, select the corresponding check boxes.
7	Click Finish. This creates the derived class with a statement that defines the derived class and the super class it inherits from.

TeamMember class example

The `TeamMember.cls` that you will import in this course inherits the data members, methods, and properties of the `Emp.cls` that you have already worked with. Its data represents the team members. Displayed here is the code for the `TeamMember class`. Notice that the `class` statement includes the `inherits` keyword for inheriting from the `Emp` class. Notice that we add `using` statements for both the `Emp` and `Manager` classes so we can access the definitions of these classes. The `GetInfo()` method is typically inherited by default, but in this example the `GetInfo()` method is overridden to include the `TeamMember` details. In addition, this class defines an additional method, `GetManager()`, which is specific to this class and does not exist in the `Emp` class. Since this class inherits from `Emp`, it has access to the `public` and `protected` data members of the `Emp` class.

```
using Progress.Lang.*.
using Enterprise.HR.Emp.
using Enterprise.HR.Role.TeamMember.
using Enterprise.HR.Role.Manager.
block-level on error undo, throw.
class Enterprise.HR.Role.TeamMember inherits Emp:
  define private property Mgr as Manager no-undo
    get.
    set.

  constructor public TeamMember ( input pMgr as Manager ):
    super ().
    assign
      EmpType = "TeamMember"
  Mgr = pMgr .
  end constructor.

  method public Manager GetManager():
    return Mgr.
  end method.

  method public override character GetInfo( ):
    var char result.
    result = super:GetName() + ", Team Member " +
      Address + " " + PostalCode + " " +
   "Job Title: " + JobTitle + " " +
   "Vacation Hours: " + string(VacationHours).
   return result.
  end method.
end class.
```

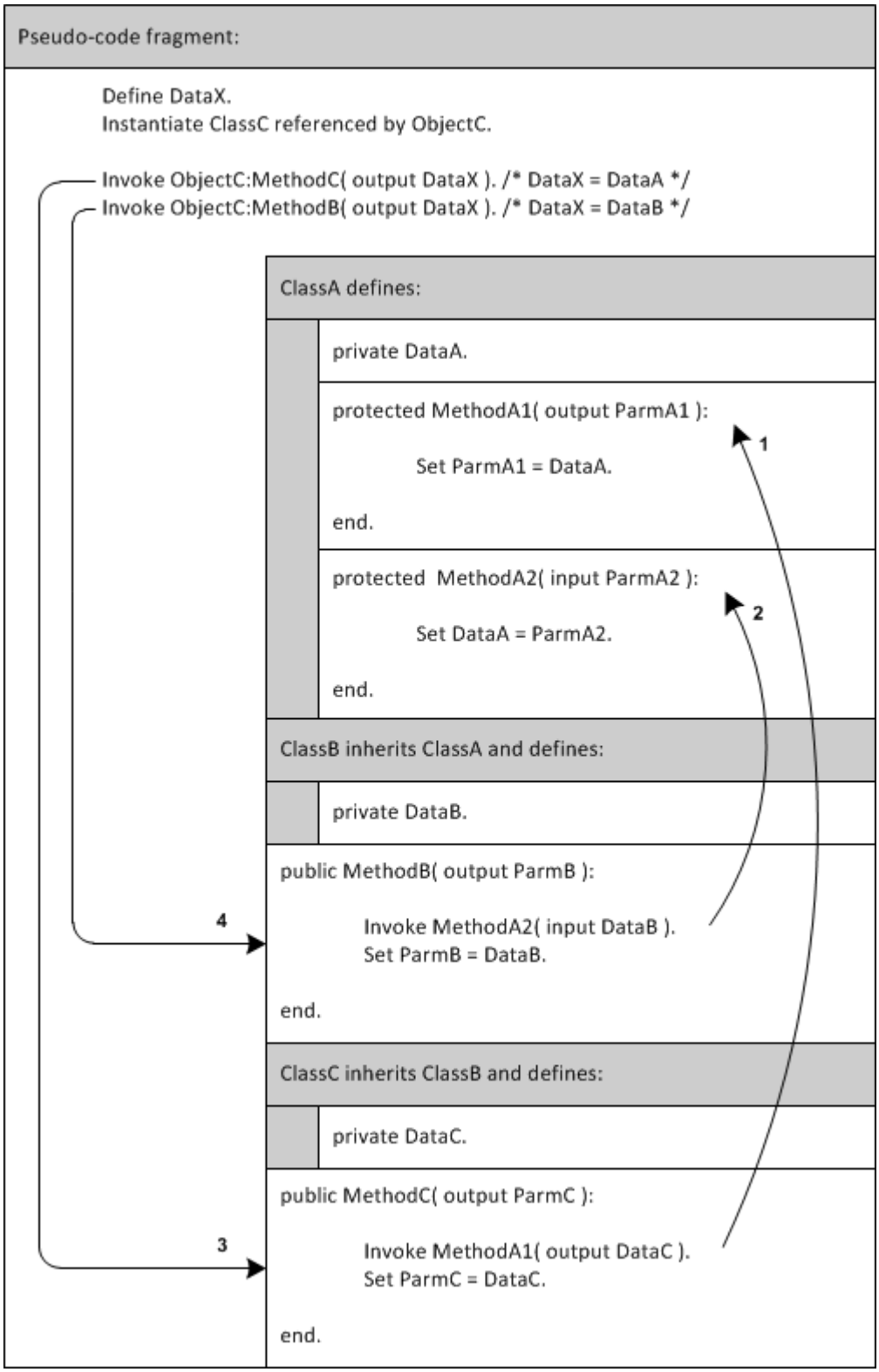
Use inheritance

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

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

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

Figure 1: Inheritance example



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

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

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

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

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

Note: Method overriding is fundamental to implementing polymorphism. For more information, see 'Use polymorphism with classes' in *Develop Object-oriented ABL Applications*.

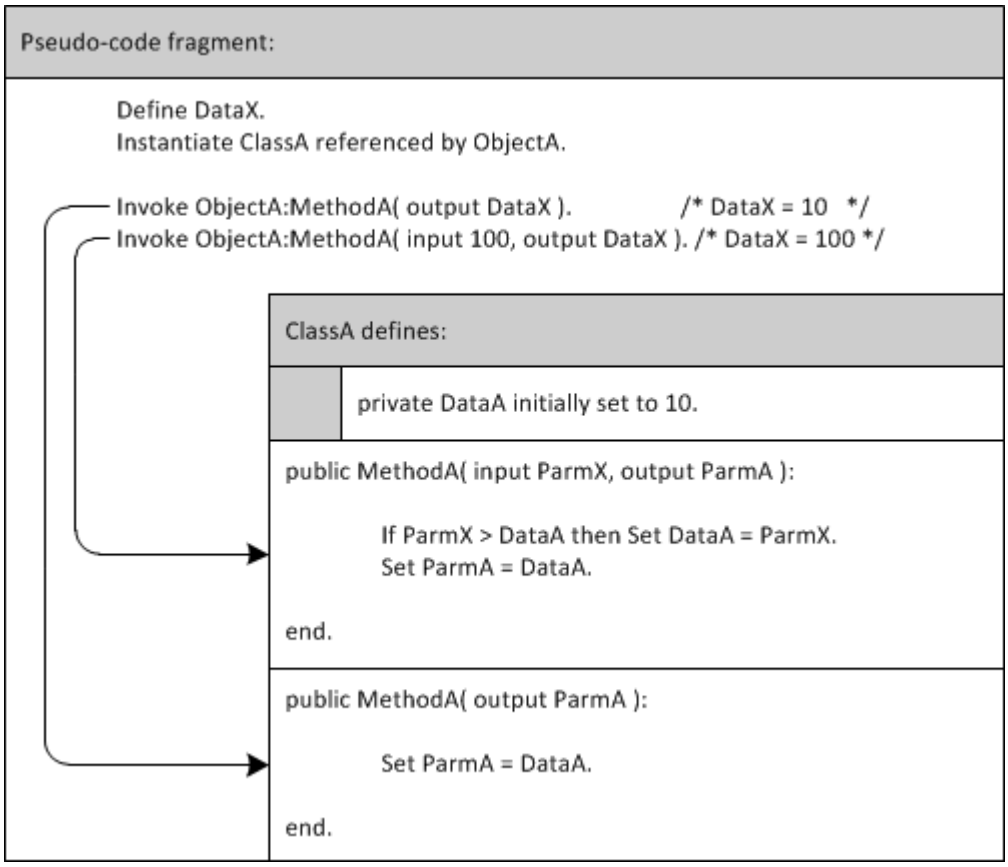
ABL, like Java and some other object-oriented languages, uses a *single-inheritance model*. This model allows a subclass to explicitly inherit from only a single super class. This single super class can then inherit from a single super class, and so on. Thus, a subclass explicitly inherits from its single,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

Method overloading

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

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

Figure 2: Method overloading example



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

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

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

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

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

Delegation

Delegation is the simple yet powerful concept of letting contained objects do their own work. It is used to describe the situation where one object assigns a task to another object, known as the delegate.

In ABL, Delegation uses composition to build a class from one or more other classes without relating the classes in a hierarchy. 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.

See also:

- 'Delegation' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming delegation' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming compile time checking' in *Develop Object-oriented ABL Applications*

Use delegation with classes

Object-oriented design provides multiple models for facilitating code reuse. One model is inheritance, which is the model shown in most of the examples thus far. The other code reuse model is the delegation model. This topic demonstrates how you can use the delegation model in ABL. In the delegation model one class acts as a principal object for a set of related behavior that is implemented by other classes. This principal object is called a *container class*. The container class can create instances of other classes with the `NEW` function. These other classes that provide services to the container are called *delegate classes*. The container class executes delegate class behavior by forwarding method invocations (object messages) on to the delegate for processing. In this way, the principal object acts as a container for aggregated behavior in the sense that it is responsible for starting, managing, and using the other classes that provide services to the container, which in turn provides the services of all its delegates to users or clients of the container. In addition, each delegate class can be used by multiple containers to provide the same services for different purposes.

The association between a container class and a delegate class is somewhat looser than the strict compile-time definition of a class hierarchy, in which each class in the hierarchy explicitly states its position within the hierarchy as its first statement. There is nothing inherent in a class, no statement in a class, that specifically defines it as a container or a delegate. However, note that because of strong typing, the relationships and dependencies between a container and its delegates are verified and enforced at compile-time; the compiler references the code for each delegate of a container in order to validate all method calls to the delegate from the container. Any variable that holds an object reference must be defined explicitly for the object's class type. Therefore, the compiler knows exactly what data members, properties, methods, and events can be accessed through the object reference to a delegate.

To support the running of behavior by procedures and other classes through a container, the container must define a `PUBLIC` stub method for each method that is implemented by a delegate. In other words, a container class simply using behavior in one of its delegates does not make that delegate behavior available to a container's client unless the container exposes the behavior through a method of its own.

The following example demonstrates delegation. In the example, only the container is shown. The behavior of the delegates is not represented. The container class and both delegate classes, called `logToDB` and `logToFile`, must provide an implementation of the methods `openLog( )`, `writeLog( )`, and `closeLog( )`. You can use two natural mechanisms to enforce this design—an interface or an abstract class.

The following code shows an interface definition for the example:

```
INTERFACE Ilog:

  METHOD PUBLIC VOID openLog (pcName AS CHARACTER).
  METHOD PUBLIC VOID writeLog (pcTxt AS CHARACTER).
  METHOD PUBLIC VOID closeLog ( ).

END INTERFACE.
```

The container class itself implements the method prototypes in the `Ilog` interface. As you can see from the following sample code, the container's version of these methods invokes the same method in one or the other of its two delegates, which are created by the constructor. The container also implements a separate `setMode( )` method to specify the delegate behavior to use, which is initially `logToFile`. For example:

```
CLASS Container IMPLEMENTS Ilog:

  DEFINE PRIVATE VARIABLE rlogToDB   AS CLASS logToDB   NO-UNDO.
  DEFINE PRIVATE VARIABLE rlogToFile AS CLASS logToFile NO-UNDO.
  /* logMode = 1 (File) is the default */
  DEFINE PRIVATE VARIABLE logMode    AS INTEGER INITIAL 1 NO-UNDO.

  CONSTRUCTOR PUBLIC Container ( ):
    rlogToDB = NEW logToDB ( ).
    rlogToFile = NEW logToFile ( ).
  END CONSTRUCTOR.

  METHOD PUBLIC VOID setMode (INPUT piLogMode AS INTEGER)
    logMode = piLogMode.
  END METHOD.

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

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

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

END CLASS.
```

The following code shows an abstract class definition for the example:

```
CLASS LogAbs ABSTRACT:

  METHOD PUBLIC ABSTRACT VOID openLog (pcName AS CHARACTER).
  METHOD PUBLIC ABSTRACT VOID writeLog (pcTxt AS CHARACTER).
  METHOD PUBLIC ABSTRACT VOID closeLog ( ).

END CLASS.
```

The container class itself derives from the `LogAbs` abstract class. As you can see from the following sample code, the container class is almost identical to the version that implements the `ILog` interface, except that the method implementations specify the `OVERRIDE` option. For example:

```
CLASS Container INHERITS LogAbs:

  DEFINE PRIVATE VARIABLE rlogToDB   AS CLASS logToDB   NO-UNDO.
  DEFINE PRIVATE VARIABLE rlogToFile AS CLASS logToFile NO-UNDO.
  /* logMode = 1 (File) is the default */
  DEFINE PRIVATE VARIABLE logMode    AS INTEGER INITIAL 1 NO-UNDO.

  CONSTRUCTOR PUBLIC Container ( ):
    rlogToDB = NEW logToDB ( ).
    rlogToFile = NEW logToFile ( ).
  END CONSTRUCTOR.

  METHOD PUBLIC VOID setMode (INPUT piLogMode AS INTEGER)
    logMode = piLogMode.
  END METHOD.

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

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

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

END CLASS.
```

4

Benefits of interfaces

An interface class is like a template that developers can use to help standardize how a set of classes are defined. An interface class defines the PUBLIC data members and methods required for a set of classes. PUBLIC data members can include temp-tables, datasets, or properties, but not variables.

Defining and developing a class that uses an interface class includes a number of tasks:

- Define the interface class that will be used by the class.
- Define the class using the interface class.
- Implement the constructors, methods, and destructor of the class.

Note: An interface class is not a superclass in an inheritance hierarchy. It is simply a way to provide a standardized template for how a set classes are to be implemented. These classes do not need to be part of a specific hierarchy. Because classes that are based on an interface class, do not inherit from the interface class, all members of the interface class must be PUBLIC.

For details, see the following topics:

- Define an interface class
- Define a class that uses an interface class
- Interface hierarchies and inheritance
- What is strong typing?

Define an interface class

You create an interface class in Developer Studio by opening the **New ABL Interface** wizard from the directory where you want to create the interface class file and then naming the interface class. Then, in the editor, you add the PUBLIC data members and methods for the class.

PUBLIC data members can be temp-tables, datasets, variables, or properties. You define them just as you would in a class definition.

You also need to define all the PUBLIC methods required by the interface. You define the method's return type, name, and parameter list just as you would in a class definition, except that statement ends with a period after the parameter list, rather than with a colon. There is no body or end method statement.

Here is an example. Suppose we used the **New ABL Interface** wizard to create an interface class called IProduct in the Inventory folder. We will use the IProduct interface class as a template for each type of product in the inventory.

This is what the **New ABL Interface** wizard will generate and open in the editor.

```
using Progress.Lang.*.
using Inventory.IProduct.
block-level on error undo, throw.
interface Inventory.IProduct:
end interface.
```

Within the interface block, we would then need to define the PUBLIC data members and methods required by the interface.

Here is the interface block for the IProduct interface class with a data member and three methods defined.

```
using Progress.Lang.*.
using Inventory.IProduct.
using Inventory.Item.
block-level on error undo, throw.
interface Inventory.IProduct:
  {include/Items.i}
    method public void AddItem( input pItem as Item).
    method public Item GetItem( input pItemCode as character ).
    method public integer NumberItems().
end interface.
```

Note: The methods are defined with no body or end method statements and end with a period instead of a colon.

Use the New ABL Class wizard to create a class that implements an interface class

After you define an interface class, you use the **New ABL Class** wizard to create a class that uses an interface class. When you create a class using the **New ABL Class** wizard, the starter code for the class is automatically generated for you.

Follow these steps in **Developer Studio** to create a class that uses an interface class:

Step	Action

1	Select the directory in which you want to create the class.
2	Right-click and select **New > ABL Class**.
3	In the **Class name** field, specify the name of the class.
4	In the **Implements** area, click the **Add…** button.
5	In the I**nterface Selection** window, find and select the interface that you want to use as the template for your class.
6	Click **OK**.
7	Specify any other information that you want for the new class. If it is a class, you must define the default constructor.
8	Click **Finish**. The newly created class is open in the editor.

Define a class that uses an interface class

When you define a class that uses an interface class, it will have the same data member names and method names with parameters as the interface class.

Here is an example. Suppose you created a class named `MobileApp` based on the `IProduct` interface class using the **New ABL Class** wizard. The wizard generates the following starter code for you and opens it in the editor.

```
using Progress.Lang.*.
using Inventory.IProduct.
using Inventory.Item.

block-level on error undo, throw.

class Inventory.MobileApp implements IProduct:

  {include/Items.i}

  method public void AddItem( input pItem as Item ):
    undo, throw new Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
  end method.

  method public Item GetItem( input pItemCode as character ):
    undo, throw new Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
  end method.

  method public integer NumberItems( ):
    undo, throw new Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
  end method.
end class.
```

When you define a class that uses an interface class, it must have the same data member names and method names with parameters as the interface class. You cannot change the definitions of any of the methods from the interface class. All you can change is the body of the code within the methods.

You do not have to write code for the bodies of all the methods defined in the interface class, but they must be in the class, at least with an empty method body. In addition, you can add any data members or methods to the class that will help implement the behaviors of the class.

Here is an example of a `MobileApp` class based on the `IProduct` interface class. Notice that we added a data member `NativeOS` to represent the operating system for a class instance. We provided implementations for two methods `AddItem()` and `GetItem()`. We did not provide an implementation for the `NumberItems()` method but its definition remains in the class.

```
using Progress.Lang.*.
using Inventory.IProduct.
using Inventory.Item.
block-level on error undo, throw.

class Inventory.MobileApp implements IProduct:
  {include/Items.i}
  define public property NativeOS as character no-undoget.
    set.

  method public void AddItem( input pItem as Item ):
    create ttItem.
 // copy data from pItem to ttItem
  end method.

  method public Item GetItem( input pItemCode as character ):
    find ttItem where ItemCode = pItemCode no-error.
    //return the reference to the item object in the ttItem record
  end method.

  method public integer NumberItems( ):
    new Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
  end method.

end class.
```

See also

- 'Comparison with procedure-based programming persistent procedures' in *Develop Object-oriented ABL Applications*

Interface hierarchies and inheritance

You can define an interface by inheriting the member prototypes from other interfaces as well as by defining member prototypes directly. Any class instance that implements the interface must implement the members of all super interfaces in the interface hierarchy as well as its own members. In addition, the class instance can be referenced as any of the super interface types as well as the interface type that inherits them. This provides an additional dimension to enable access to any object that implements the interface. The following topics detail how an interface hierarchy can be built, including limitations and guidelines for using it.

Multiple interfaces in a hierarchy

Each class implementing an interface is required to implement all members of the interface. However, a class inheriting from a class already implementing the same interface also inherits all of the implementations. Thus you need not explicitly implement the members.

Interface inheritance allows the same interface to appear multiple times within an interface hierarchy. That is, it is valid for an interface to inherit member prototypes from an interface that it already inherits from.

For example:

```
INTERFACE IMyX :
```

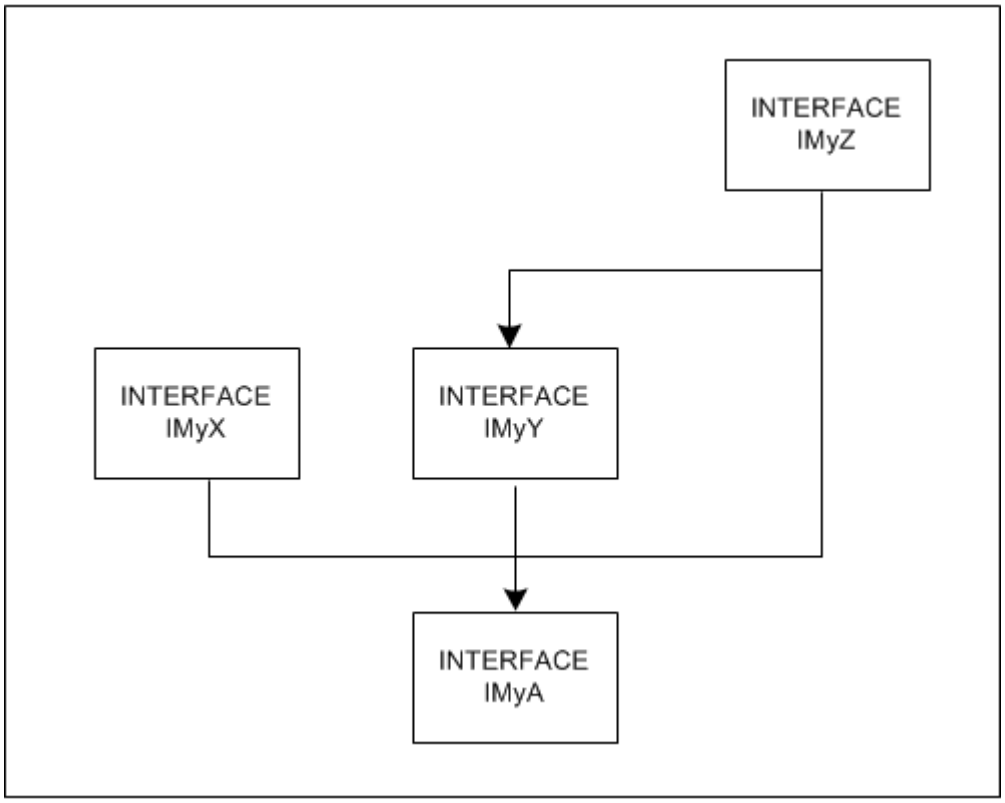
```
INTERFACE IMyY INHERITS IMyZ :
```

```
INTERFACE IMyZ :
```

```
INTERFACE IMyA INHERITS IMyX, IMyY, IMyZ :
```

The following figure depicts the interface hierarchy where interface `IMyZ` is inherited by both interface `IMyY` and `IMyA`.

Figure 3: Multiple Interfaces in hierarchy



It is not an error having interface `IMyZ` appear twice in the hierarchy. It is however, a compile time error to have an interface appear within its own hierarchy.

For example:

```
INTERFACE IMyJ INHERITS IMyJ :
```

The above example results in a compile time error.

Duplicate members in a hierarchy

Without interface inheritance, a class can implement more than one interface with multiple interfaces that include identical member signatures. In this case, the class is required to implement only one member that satisfies the identical signature in multiple interfaces.

With interface inheritance, the same rules apply. Multiple identical members appearing in different interfaces within the hierarchy are allowed and require only one implementation in the implementing class. Member types in the hierarchy that differ only by return type will generate compile time errors. Overloaded members may appear within the same interface, or in different interfaces within the hierarchy. This is comparable to existing rules with respect to overloading.

```
INTERFACE IMyTop:
  DEFINE PUBLIC PROPERTY myInteger AS INTEGER
    GET.
    SET.
END INTERFACE.
```

```
INTERFACE IMyBottom INHERITS IMyTop:
  DEFINE PUBLIC PROPERTY myInteger AS INTEGER
    GET.
    SET.
END INTERFACE.
```

```
CLASS MyImplementation IMPLEMENTS IMyBottom :
  DEFINE PUBLIC PROPERTY myInteger AS INTEGER
    GET():
      RETURN myInteger.
    END GET.
    SET(INPUT val AS INTEGER):
      myInteger = val.
    END SET.
END CLASS.
```

In the above example, it is allowed for interface `IMyBottom` to have its own definition of the existing property, `myInteger` that appears in `IMyTop` interface. Also, the class `MyImplementation` needs to implement only the single property that satisfies them both.

Use interface hierarchies

A class implementing an interface hierarchy does so in the same manner as a regular single interface. To the implementing class, the extended interface acts as a meta-interface that encapsulates all of the members of the interface hierarchy. This enhances the ability to implement more advanced interface polymorphism. You can implement any of the interfaces in the hierarchy.

For example:

```
INTERFACE ISedan  INHERITS IVehicle, ICar   :
```

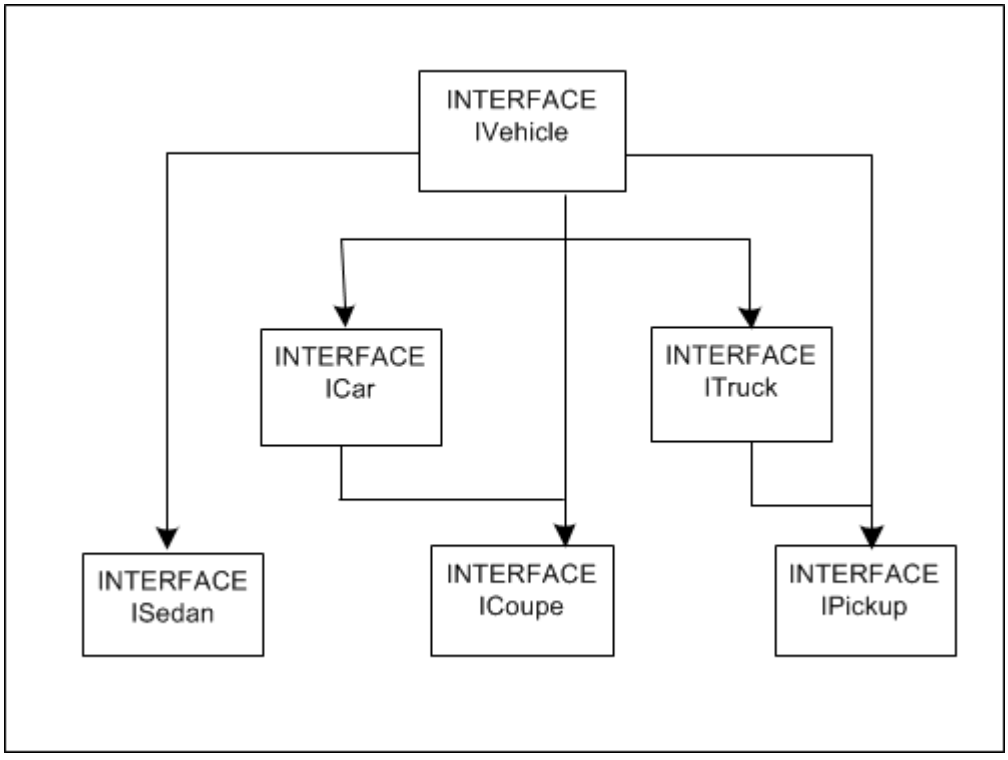
```
INTERFACE ICoupe  INHERITS IVehicle, ICar   :
```

```
INTERFACE IPickup INHERITS IVehicle, ITruck :
```

In the above example, three interfaces, `ISedan`, `ICoupe`, and `IPickup` inherit from the existing interfaces.

The following figure depicts the interface hierarchy where interface `ISedan`, `ICoupe`, and `IPickup` inherits member prototypes from the existing interfaces `IVehicle`, `ICar`, and `ITruck`.

Figure 4: Using interface hierarchy



The three classes that implement each extended interface are:

```
CLASS Altima  INHERITS Nissan IMPLEMENTS ISedan :
```

```
CLASS Mustang INHERITS Ford   IMPLEMENTS ICoupe  :
```

```
CLASS Tundra  INHERITS Toyota IMPLEMENTS IPickup :
```

An application making use of these types can choose to program to the interfaces via polymorphism.

For all three classes, both the function `TYPE-OF` and the `Progress.Lang.Class` method `IsA( )` will return true for `IVehicle`. The first two, `Altima` and `Mustang`, will both return true for `ICar`. Class `Tundra` will return true for `IPickup`.

```
PUBLIC METHOD LOGICAL handleVehicle(INPUT refVehicle AS CLASS IVehicle):

  IF TYPE-OF(refVehicle, "ICar") EQ TRUE
    THEN THIS-OBJECT:handleCar(refVehicle).

  ELSE IF TYPE-OF(refVehicle, "ITruck") EQ TRUE
    THEN THIS-OBJECT:handleTruck(refVehicle).
    ELSE THIS-OBJECT:handleGeneric(refVehicle).
```

The above example is of a client programming to the interfaces. All the three classes are valid as parameters to this method. The first two is passed into `handleCar( ) method`, while the third is passed to `handleTruck( )` method.

What is strong typing?

Strong Typing is the enforcement of type conformance. The object types defined by classes and interfaces are analogous to the built-in primitive types of the language. Object types are identified using an object type name that includes the name of the class or interface. For a user-defined class or interface, this name matches the file name of the file that stores the definition of the class or interface type.

Strong types defined by a programmer are enforced by the compiler for conformance to a specific Class definition. Among other benefits, this allows checking many possible errors at compile time instead of discovering them at run time.

See also:

- 'Strong typing' in *Develop Object-oriented ABL Applications*
- 'Classes and strong typing' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming compile time validation' in *Develop Object-oriented ABL Applications*

5

Benefits of singletons

When an instance is created, you have a reference to the instance. You are responsible for the life cycle of the instance. In some cases, you may want to have a global instance that is available to all code running in the ABL Virtual Machine (AVM).

For example, you may want to have a single class instance that provides security for your application that all application code can access. You can implement this by defining a class that has a static constructor and static data members. One of the static data members is used to hold an instance of the class in the AVM. This data member must be public. This single class instance is called a singleton.

A static constructor behaves differently from a regular constructor. You cannot call a static constructor in your code. When an application starts, the static constructors for all singletons are called automatically. You must add code to these static constructors to create an instance of the class and assign it to the static data member. Any code that runs in the AVM has access to this static data member. You do not define parameters for a static constructor.

Note: Singleton classes are used extensively in applications that utilize a state-free operating mode in the application server, such as Progress OpenEdge Data Object Services that can be accessed via a REST protocol by web and mobile clients.

See also:

- 'Comparison with procedure-based programming main block' in *Develop Object-oriented ABL Applications*

For details, see the following topics:

- Define a static data member
- Define a static constructor
- Singleton classes and procedures as Data Object resources

Define a static data member

In your class, you define a PUBLIC static data member to represent a single instance of the class that runs in the AVM.

Here is the syntax for defining a static property as a data member:

```
define public static property <property-name> as {<type-name>} [no-undo]
  public get [():
    <body of get that returns property>
  end get].
  [<access-mode>] set[(<param>):
    <body of set that sets property>
  end set].
```

Syntax Element	Description
access-mode	Specifies whether the data member will be available to other parts of an application. As a best practice, you should define a set to be PRIVATE or PROTECTED. For more information, see Access modes on page 24.
property-name	The name of the data member.
type-name	A built-in ABL type or a user-defined type.
param	The value that is used to set the property. It must match the type of the property.

Here is the definition of the public static data member, Instance, for the Corporation class. This data member will be used to hold the single and only instance of the Corporation class in the AVM. Notice that you must include the full name of the class for the type that includes the package, Enterprise.Corporation.

```
define public static property Instance as  Enterprise.Corporation no-undo
  get.
  private set.
```

Define a static constructor

A static constructor is used to initialize the static data member for a class.

Here is the syntax for defining a static constructor:

```
constructor static <class-name> ():
  <body of constructor>
end constructor.
```

Syntax Element	Description
class-name	Must match the name of the class in the class definition.
body of constructor	The ABL code necessary to implement the functionality of the constructor.

Here is the code for the static constructor of the `Corporation` class. Notice that it creates an instance of `Enterprise.Corporation` by calling the default constructor for Enterprise.Corporation. In this static constructor, you must fully specify the name of the class. The default constructor returns an instance of `Corporation`. The static constructor then assigns this value to the static data member named `Instance`. After this constructor runs, there is a single instance of the Corporation class running in the AVM.

```
constructor static Corporation ( ):
  Enterprise.Corporation:Instance = new  Enterprise.Corporation().
end constructor.
```

Singleton classes and procedures as Data Object resources

The ABL to implement the interface to a single Data Object resource must be coded in a single class or external procedure that can be executed as a singleton object. A singleton object is a class or external procedure that once initially instantiated, the same instance is shared by all consumers of the class or procedure no matter how many times it is instantiated in a given application server session.

When, on behalf of a mobile or web app, a Web application executes any Data Object operation in the resource running in an application server session, if the ABL class or external procedure has not yet been instantiated as a singleton object, the session instantiates it and executes the ABL routine that implements the operation. When the operation completes, the object remains instantiated for access by other client requests. When the Web application executes another operation on the same resource running in the same ABL session, the same singleton object is then used to execute the ABL routine for that operation, and so on. If another application server session executes the operation for the same Data Object resource, the same process repeats itself, instantiating the singleton if it does not exist and remaining instantiated for all additional calls to the same resource on that or another session where the same object is instantiated. Once all running application server sessions have executed an operation for that same Data Object resource, they all maintain their singleton objects as long as they continue to run. Again, the process repeats for any additional session that runs on the application server and responds to a Data Object operation request.

ABL classes are inherently coded to be instantiated as singletons. However, external procedures must meet a basic requirement to be instantiated as singletons, and that is, they cannot contain any `DEFINE PARAMETER` statements in the main block. They can contain internal procedures and user-defined functions with their own parameters, each of which can implement a Data Object operation exactly like a method of a class.

Note that the singleton coding requirement for external procedures applies only to an external procedure that implements the resource for a Data Object and its operations. Any additional procedures or classes that a singleton class or procedure accesses can be implemented like any other class or procedure that runs on an OpenEdge application server.

For more information on singleton procedures instantiated in the context of an ABL client, see the `RUN` statement in *OpenEdge Development: ABL Reference*. Although this information is for ABL clients, singleton procedure behavior and coding requirements apply for OpenEdge procedures running on an application server as well.

6

Benefits of dynamic instances

You can create class instances by defining a variable or property of a user-defined type and then use the new option to create an instance of that class type. Here is an example:

```
using Enterprise.BusinessUnit.IBusinessUnit.
  . . .
var Dept D = new Dept().
```

When you create an instance of a class this way, you must know the class name at compile time to create the class. If your application will create many class instances of various types, you will have to write a lot of type-specific code. In some cases, you may not know the types until runtime. For example, the type information is in a configuration file that is used to bootstrap the application.

In these cases, you can write your code to create and access class instances dynamically. You use the `dynamic-new` statement to create an instance of a class type and assign the instance to the variable of type `Object`.

Here is the syntax for creating an instance dynamically.

```
<object-reference> = dynamic-new <class-name>([<parms>]).
```

Syntax Element	Description
object-reference	A property, variable, or field in a temp-table of type Progress.Lang.Object.

class-name	The fully qualified class name from which the definition of the class is known at runtime.
parms	The parameters, if any, for the public constructor for the class.

For details, see the following topics:

- Create a class instance dynamically

Create a class instance dynamically

Suppose a **Corporation** class has been written to create and access class instances dynamically. The **Corporation** class has defined a temp-table, **ttBusinessUnit**, with information about all the possible classes that the application may instantiate at runtime.

```
define private temp-table ttBusinessUnit no-undo
  field MaxNumDepartments as character
  field ClassName as character // .CLS name
  field BusinessUnitRef as Progress.Lang.Object.
```

When the application starts, records are added to the **ttBusinessUnit** temp-table for every class that could be used at runtime for all use cases of the application. At runtime, only a subset of the possible classes is instantiated, based upon the use case. Here is the code to dynamically create the required class instances that are now known at runtime. In this example, the classes that are instantiated have a single argument constructor for the name of the business unit.

```
// add code here to dynamically create the business  unit depending
// on the Type value
if ttBusinessUnit.Type = "Company"
  then
    ttBusinessUnit.BusinessUnitRef = dynamic-new(ttBusinessUnit.Classname)
      (ttBusinessUnit.Name).
 else
   ttBusinessUnit.BusinessUnitRef = dynamic-new(ttBusinessUnit.Classname)
       (ttBusinessUnit.Name,ttBusinessUnit.MaxNumDepartments).
  empty temp-table ttDepartment.
```

7

Benefits of events

Object-oriented applications include the ability for objects to kick off events when the state of an object changes. This is very common in the user interface for business applications. Whenever a button is clicked, an action is completed because of the button being chosen or clicked. For example, clicking on a button labelled send will send an email to al the recipients.

Events can be triggered by a user or triggered programmatically. Events provide the benefit of asynchronously responding to the the changes in the state of the application.

For details, see the following topics:

- Define events
- Use class events
- Publish class events
- Subscribe and unsubscribe event handlers

Define events

A class event is a class member of the publisher class that you define. Here is the short syntax for defining a class event in the publisher class.

```
define [access-mode] event Event-Name
  signature void (parameter, [parameter…]).
```

Syntax Element	Description

access-mode	Specifies whether the method will be available to other parts of an application. For more information, see Access modes on page 24. For a class event, the access mode specifies where you can subscribe a handler for the event using the subscribe( ) method.
Event-Name	The name of the event. This name must be unique among all other events, properties, and variable data members in the application.
signature void (parameter, [parameter..])	The signature of the class event. This indicates the required signature for any event handler that subscribes to the event.

See also:

- 'Define class events' in *ABL Reference*
- 'DEFINE EVENT statement' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming named events' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming PUBLIC and PRIVATE events' in *Develop Object-oriented ABL Applications*

Define events within a class

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

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

When you define a class event, you also define a method signature (which is always `VOID`) for any event handlers that you subscribe to the event. For methods that you subscribe to, ABL type checks this event signature at compile time, like any method signature. However, for internal procedures that you subscribe, the AVM checks signature compatibility only at run time. Note that you pass the parameters for an event when you publish it, and the data flow through these parameters depends on the modes of the parameters (`INPUT`, `OUTPUT`, or `INPUT-OUTPUT`) and how many event handlers you have subscribed to the event. The values of `OUTPUT` and `INPUT-OUTPUT` parameters are determined by the last handler to run. In the case of `INPUT-OUTPUT` parameters, each handler invoked for the published event has access to the value set by the previous handler. For more information on how data flows through the signatures of published class events, see Publish class events on page 70.

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

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

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

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

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

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

Use class events

For any ABL class event, used by itself or to implement an interface or inherited abstract event, you must follow these general steps:

1. Define the event using the 'DEFINE EVENT statement' in *Develop Object-oriented ABL Applications*.
2. Subscribe one or more handlers to the event using the 'Subscribe( ) event method' in *ABL Reference*. You can subscribe handlers to an event anywhere the event is accessible in an application.
3. Publish the event for a given condition using the 'Publish( ) event method' in *ABL Reference*, which runs all subscribed event handlers for the event. You can publish the event only from within the class definition that implements a non-abstract definition for the event.
4. Process any values returned from the Publish( ) method parameters.

At any point after subscribing event handlers, you can optionally unsubscribe any event handlers you no longer need using the 'Unsubscribe( ) event method' in *ABL Reference*. Thus, like named events for procedures, class events allow you to dynamically define and undefine alternate routines that execute in response to the event.

Note that similar to ABL class events, only the class that implements a non-abstract definition for a .NET event can publish it. This includes any ABL class that derives from and implements a .NET abstract class defining abstract events or that implements a .NET interface defining event prototypes.

However by convention, a .NET class that defines an event also typically defines a corresponding protected `On`*`EventName`*`( )` method that allows a derived class to publish the event, where *`EventName`* specifies the name of the .NET event and the signature defines a `System.EventArgs` (or derivation) as input. If the `On`*`EventName`*`( )` method is overrideable (`virtual` in C#), you can also override this method in an ABL derived class. If you do override this method, you must also invoke the super class implementation of `On`*`EventName`*`( )` from within the overriding method (using the 'SUPER system reference' in *Develop Object-oriented ABL Applications*) in order to ensure that all subscribed .NET delegates also respond to the event.

For more information on overriding .NET methods, see the 'METHOD statement' in *ABL Reference*.

To allow .NET events from .NET form and control objects to be published as part of the OpenEdge GUI for .NET, you must block to display the .NET forms and allow your subscribed event handlers to execute in response to any published events. To enable this .NET event handling, ABL provides WAIT-FOR statement syntax that blocks for both GUI for .NET events and ABL handle-based object events.

For more information, see the 'WAIT-FOR statement (.NET and ABL)' in *ABL Reference*.

Otherwise, you can work with .NET events in exactly the same way as ABL class events.

Publish class events

In your publisher class you write code to identify when a condition has occurred. Then, you call the `publish()` method to publish the event.

Here is the syntax to publish a class event:

```
Event-Name:publish (parameter, [parameter…])no-error.
```

Syntax Element	Description
no-error	Specifies how you want to handle errors raised from publishing the event. Note that if there are multiple subscribers to an event and an error is raised, any unexecuted event handlers will not execute.

Note: The parameter list should match the parameters defined in the event definition.

See also

- 'Publish class events' in *Develop Object-oriented ABL Applications*
- 'Comparison with procedure-based programming PUBLISH and SUBSCRIBE' in *Develop Object-oriented ABL Applications*

Subscribe and unsubscribe event handlers

In the subscriber class, you subscribe to an event specifying the event handler that will execute when the event occurs. You also need to write a method to handle the event.

Here is the syntax to subscribe to a class event:

```
publisher:Event Name :subscribe(handler-method).
```

Syntax Element	Description
publisher	The class where the event is defined and published.
handler-method	The name of the method that will execute when the event is published.

You can also unsubscribe to an event. Here is the syntax to unsubscribe to a class event:

```
publisher:Event-Name:unsubscribe.
```

See also:

- Subscribe( ) event method in *ABL Reference*
- Unsubscribe( ) event method in *ABL Reference*