Lesson 3: Using ABL Classes in an Application

Lesson introduction

In the previous lessons, you learned how to develop and test classes. In this lesson, you will learn how to further develop an application using various ABL object-oriented programming features such as inheritance, interface classes, singletons, dynamic instances, and events.

In this lesson, first, you will learn how to build class inheritance hierarchies to share data members, properties, methods, and events between related classes. Next, you will learn how to define a class using an interface class. Then you will learn how to define singletons. Next, you will learn how to create instances dynamically. Finally, you will learn how to define and publish an event and subscribe to it.

Learning objectives

When you complete this lesson, you should be able to:

- Define and use an inheritance hierarchy
- Define and use interface classes
- Create singletons (static instances)
- Create instances dynamically
- Define and use class events

Prerequisites

Before you begin this lesson, you should meet the following prerequisites:

Prerequisite	Resources
Experience with ABL procedural programming	The course *4GL Essentials*
Creating OpenEdge projects in Progress® Developer Studio for OpenEdge®	The course *Introduction to Progress Developer Studio for OpenEdge*
Understanding of the ABL object-oriented programming concepts	Lesson 1 of this course, "Introduction to Object-oriented programming"
Creating classes and methods	Lesson 2 of this course, "Getting started with ABL classes"

Using inheritance

As you previously learned, a derived class can inherit non-private (that is protected or public) data members, properties, and methods from a super class. 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>:
```

Overriding methods within a class hierarchy

In a derived class that inherits from another class, you can define a method to override a *public* or *protected* 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. That is, 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 [visibility]  override <return-type>  <method-name>(  ):
```

Procedure: Using 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.

Example: TeamMember class

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(  ):

        define variable result as character no-undo.
        result = super:GetName() + ", Team Member " +
                  Address + " " + PostalCode + " " +
                  "Job Title: " + JobTitle + " " +
                  "Vacation Hours: " + string(VacationHours).
        return result.

    end method.

end class.
```

Try It 3.1: Using inheritance

In this Try It, you will use the *Emp* class as a super class for the *Manager* and *TeamMember* derived classes. First, you will modify a data member of the *Emp* super class so that it can be used by its derived classes. Then you will create and develop the code for the *Manager* class. Next, you will import existing code for the *TeamMember* and *Dept* classes, along with test procedures for testing your class hierarchy. Finally, you will test the inheritance hierarchy.

The exercise steps take approximately 60 minutes to complete.
Please refer to the *Exercise Guide* for the instructions for this Try It.

Using interface classes

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.

In this topic, you will learn how to define an interface class and use it to define another 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.

Defining 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. Note that the methods are defined with no body or *end method* statements and end with a period instead of a colon.

```
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.
```

Procedure: Using the New ABL Class wizard to create a class that uses 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 Interface 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.

Defining 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-undo
get.
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(  ):
undo, throw new Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
end method.
end class.
```

Check your understanding – Question 1

What items can you define in an interface class?

Choose all that apply.

A. Public variables
B. Private variables
C. Public properties
D. Private properties
E. Public methods
F. Private methods

Answers are at the end of the lesson.

Try It 3.2: Using an interface class

In this Try It, you will create the *IBusinessUnit* interface class that defines the required methods of the *Company* and *Franchise* classes. You will then create the *Company* class and import the *Franchise* class. Next, you will test the classes.

The exercise steps take approximately 45 minutes to complete.
Please refer to the *Exercise Guide* for the instructions for this Try It.

Using singletons

In the previous lesson, you learned how to write code to create an instance of a class using a constructor with or without arguments. 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 constructers 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.

In this topic, you will learn how to define a static data member and a static constructor for a singleton.

Defining 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].
[<visibility>] set[(<param>):
   <body of set that sets property> end set].
```

Syntax Element	Description
visibility	Public, private, or protected.
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.
```

Defining 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.
```

Creating class instances dynamically

In the previous lesson, you learned how to create class instances by defining a variable or property of a user-defined type and then using the *new* statement to create an instance of that class type. Here is an example:

```
using Enterprise.BusinessUnit.IBusinessUnit.
. . .
define variable D as Dept no-undo.
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.

Example: Creating 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.
```

Check your understanding – Question 2

For a class, what must you define so that the class will have "singleton" behavior?

Choose all that apply.

A. Specify a *singleton* keyword in the class definition.
B. A static public data member that will hold a reference to an instance of the class.
C. A private data member that will hold a reference to an instance of the class.
D. A constructor that takes no arguments.
E. A static constructor that takes no arguments.
F. A public Create() method for creating the instance.

Answers are at the end of the lesson.

Try It 3.3: Using a singleton and creating classes dynamically

In this Try It, you will import the *Corporation* class and add a static data member and a static constructor to it so that it can be used as a singleton. Then, you will add code to the *Corporation* class to create the business unit instances (of types *Company* and *Franchise*) of the corporation dynamically. The *Corporation* instance will be created automatically when the AVM detects the first use of *Corporation*.

The exercise steps take approximately 30 minutes to complete.

Please refer to the *Exercise Guide* for the instructions for this Try It.

Using 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.

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

```
define <private | protected | public> event Event-Name signature
void (parameter, [parameter…]).
```

Syntax Element	Description
<private | protected | public>	Specifies an access mode for this event, similar to other class members. 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.

Publishing 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.

Subscribing and unsubscribing 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.
```

Try It 3.4: Using events

In this Try It, you will define and publish an event in the *Manager* class and then subscribe to it in the *Dept* class. You will also write an event handler to handle the event. Finally, you will test your event handling code to ensure it executes correctly.

The exercise steps take approximately 30 minutes to complete.

Please refer to the *Exercise Guide* for the instructions for this Try It.

Lesson summary

You should now be able to:

- Define and use an inheritance hierarchy
- Define and use interface classes
- Create singletons (static instances)
- Create instances dynamically
- Define and use class events

Answers to *Check your understanding* questions

Question 1

C, E

Question 2

B, E