Course Guide
Version 4.0

Introduction to Object-oriented Programming

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Printed in the USA

Version 4.0

Progress Software Corporation (PSC) is concerned about the environment. To reduce waste and complete the recycling circle, we printed this manual and cover on stock that is recyclable. PSC has made every effort to look at environmental implications when deciding on this package.

Author: Annie Khan
Contributors: Elaine Rosenberg

Contents

About This Course

Course audience

This course is designed for Progress® OpenEdge® developers.

Course prerequisites

Before you begin this course, you should have:

- Experience with ABL procedural programming
- Created OpenEdge projects in Progress® Developer Studio for OpenEdge®

Student goals

Please take a few minutes to document your own goals for this course.

- What will you have to know and produce when you return to work?
- What are the things that you most want to know about ABL object-oriented programming?

Introduce yourself

Please introduce yourself to the group by sharing the following:

- Your name and your job.
- The name of your company and its type of business.
- Your technical background.
- Any prior experience with OpenEdge?
- What you would like to learn from this course.

Course goals

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

- Describe the key features of object-oriented ABL programming.
- Define the parts of an ABL class
- Access data members and call methods within a class.
- Work with other classes
- Test a class
- Define and use an inheritance hierarchy
- Define and use interface classes
- Create singletons (static instances)
- Create instances dynamically
- Define and use class events

Module overview

This course contains the following modules:

Lesson	What it covers
About This Course	Introduction and overview; identify goals and objectives.
Introduction to Object-oriented Programming	In this lesson, you will be introduced to object-oriented programming and to key features of Progress Software's object-oriented Advanced Business Language (ABL). You will also set up your development environment for the exercises in this course.
Getting Started with ABL Classes	In this lesson, first you will learn how to define data members, constructors, methods, and a destructor for a class. Then you will learn how to access data members and call methods from within a class. Next, you will learn how to work with other classes, including how to create class instances, access data members and methods, access class instances dynamically, and delete class instances. Finally, you will learn how to test an ABL class by writing a test procedure.
Using ABL Classes in an Application	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.

Notes

Lesson 1: Introduction to Object-oriented programming

Lesson introduction

In this lesson, you will be introduced to object-oriented programming and to key features of Progress® Software's object-oriented Advanced Business Language (ABL). You will also set up your development environment for the exercises in this course.

Learning objectives

When you complete this lesson, you should be able to describe the key features of object-oriented ABL programming.

Prerequisites

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

Prerequisite	Resource
Experience with ABL procedural programming	The course *4GL Essentials*
Create OpenEdge projects in Progress® Developer Studio for OpenEdge®	The course *Introduction to Progress Developer Studio for OpenEdge*

Object-oriented programming

ABL supports object-oriented programming. Object-oriented programming is important for developing OpenEdge applications because it enables you to explicitly model the users, systems, and objects that make up the use cases of the application.

ABL classes are used to represents the users, systems, or objects. Each class contains definitions for data and the code that implements the behaviors. An object-oriented application is typically a set of classes that relate to each other to implement use cases. Classes, by definition, support a variety of object-oriented features that help to organize state and behavior in an application.

When you develop an ABL class (its file that uses the **.cls** extension), you define data members (which represent the data) and methods (which 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.

ABL object-oriented programming supports the following features:

- Inheritance
- Encapsulation
- Interfaces
- Polymorphism

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.

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 each data member and method will have at runtime.

Access	Description
private	A method or data member is only accessible from methods within the class. The methods of a derived class cannot access a private method or private data member of a super class.
protected	A method or data member is accessible from the methods within the class or by the methods of its derived class.
public	A method or data member is accessible from the class methods or derived class methods. It is also accessible by any ABL code that creates or uses an instance of the class.

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.

The best practice is to hide as much data as possible, that is, to make all data members private or protected. Then, class data is only accessible by methods within the class.

Interfaces

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*() that, which would send an invoice to a customer at home. The *RetailCustomerInvoice* class would implement its version of *SendInvoice*(), which would send an invoice to a business using a purchase order (PO) number.

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.

Guided Exercise 1.1: Setting up your application development environment

In this Guided Exercise, you will prepare your development environment for writing, testing, and debugging an ABL application.

Important: You must complete this Guided Exercise to perform subsequent Try It Exercises in this course.

The exercise steps take approximately 30 minutes to complete.

Please refer to the *Exercise Guide* for the instructions for this Guided Exercise.

Lesson summary

You should now be able to describe the key features of object-oriented ABL programming.

Notes

Lesson 2: Getting started with ABL classes

Lesson introduction

In the previous lesson, you were introduced to the object-oriented programming paradigm.

In this lesson, first you will learn how to define data members, constructors, methods, and a destructor for a class. Then you will learn how to access data members and call methods from within a class. Next, you will learn how to work with other classes, including how to create class instances, access data members and methods, access class instances dynamically, and delete class instances. Finally, you will learn how to test an ABL class by writing a test procedure.

Learning objectives

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

- Define the parts of an ABL class, including:
  - Data members
  - Constructors
  - Methods
  - A destructor
- Access data members and call methods within a class.
- Work with other classes, including:
  - Creating instances
  - Accessing data members and methods
  - Accessing a class instance dynamically
  - Deleting instances
- Test a class

Prerequisites

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

Prerequisite	Resources
Create OpenEdge projects in Progress® Developer Studio for OpenEdge®	The course *Introduction to Progress Developer Studio for OpenEdge*
Experience with ABL procedural programming	The course *4GL Essentials*

Defining ABL classes

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.

These are the tasks you perform to define an ABL class:

1. Determine the package name.
2. Determine the class name.
3. Create the class file using the *New ABL Class* wizard.
4. Define the following parts of the class:
   a. Data members
   b. Constructors
   c. Methods
   d. Destructor

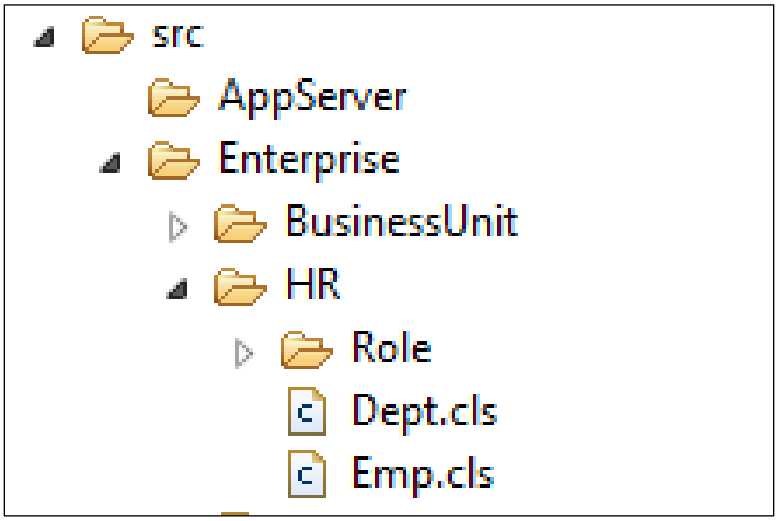
Determining the package name

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

Server
    src
        BusinessComponent
            BusinessEntity
            BusinessTask
            BusinessWorkflow
        DataAccess
        DataSource

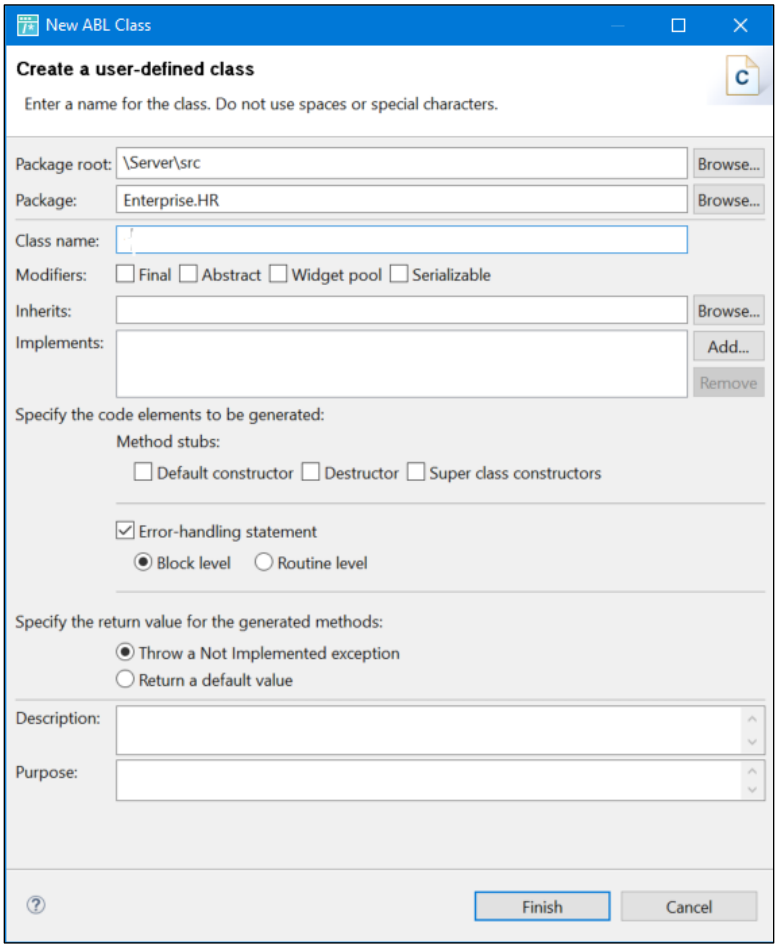
Determining the class name



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.

Using the New ABL Class wizard



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: Newly defined class Emp

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 11 14:45:07 EDT 2015
    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.
```

Parts of an ABL class definition

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.

Next, you will learn how to define these four parts of a class.

Data members of a class

The data members of a class are used to 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.

Defining a data member as a variable

Defining a variable for a class is similar to defining a variable for a procedure. The main difference is in the visibility you specify for the variable.

Here is the simplified ABL syntax for defining a variable data member:

```
define <visibility> variable <variable name> as
 <type-name> [no-undo].
```

Syntax Element	Description
visibility	Specifies whether the data member will be *private*, *protected*, or *public*. As a best practice, you should define a variable to be *private* or *protected*.
variable name	Must begin with a letter and it can contain letters, numbers, underscores, or hyphens. It must not contain periods or spaces.
type-name	Can be one of the ABL data types such as integer or character or it can be a user-defined type.
no-undo	*No-undo* is recommended for all variable definitions, regardless of where they are used. If you use *no-undo*, the value of the variable is not restored to its original value in the event of a transaction rollback. You rarely need this capability so it is recommended that you use *no-undo* as it is more efficient at runtime.

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.
    */

    define public property NumEmployee as integer no-undo.

/* rest of class definition */

end class.
```

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.

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. Most of the data members you add to an ABL class are defined as properties.

Defining 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.

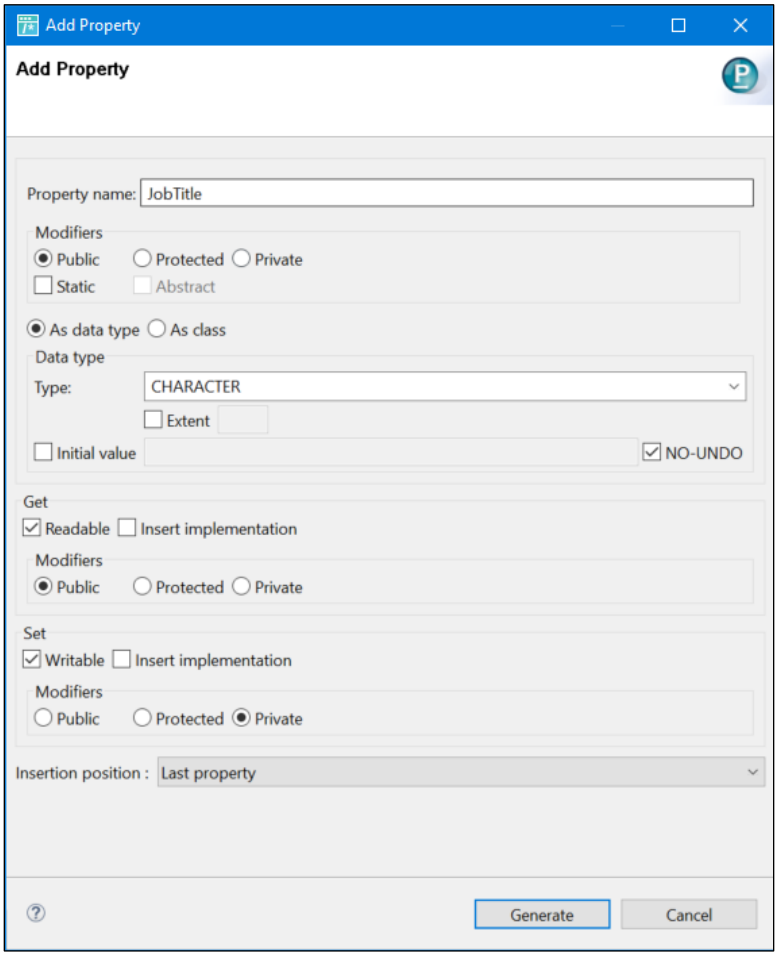
When you define a property, you specify visibility for the property, and for the *get*() and *set*() accessor methods. The visibility for *get()* and *set()* must be the same or more restrictive than the visibility for the property.

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*, *private*, or *protected*.
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.

Procedure: Using 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.

Example: Defining properties

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 pubic, 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 best practice is to define the constructor as public so that other parts of the application can create instances 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.

Defining a class constructor

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

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

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

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.

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. You will learn about overloaded methods in the next lesson.

Defining 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. 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.

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.

Defining 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

Check your understanding – Question 1

What is a package in the context of developing classes?

Choose the best answer.

A. It represents a directory path in which your class file is located.
B. It is the compressed file (.zip) created when your export your source code in Developer Studio.
C. It is the set of r-code files that are used at runtime to test your class.
D. It is a name for the compiled class file.

Answers are at the end of the lesson.

Check your understanding – Question 2

Which of the following can be part of a class definition?

Choose all that apply.

A. Variable
B. Procedure
C. Constructor
D. Method
E. Object
F. Property

Answers are at the end of the lesson.

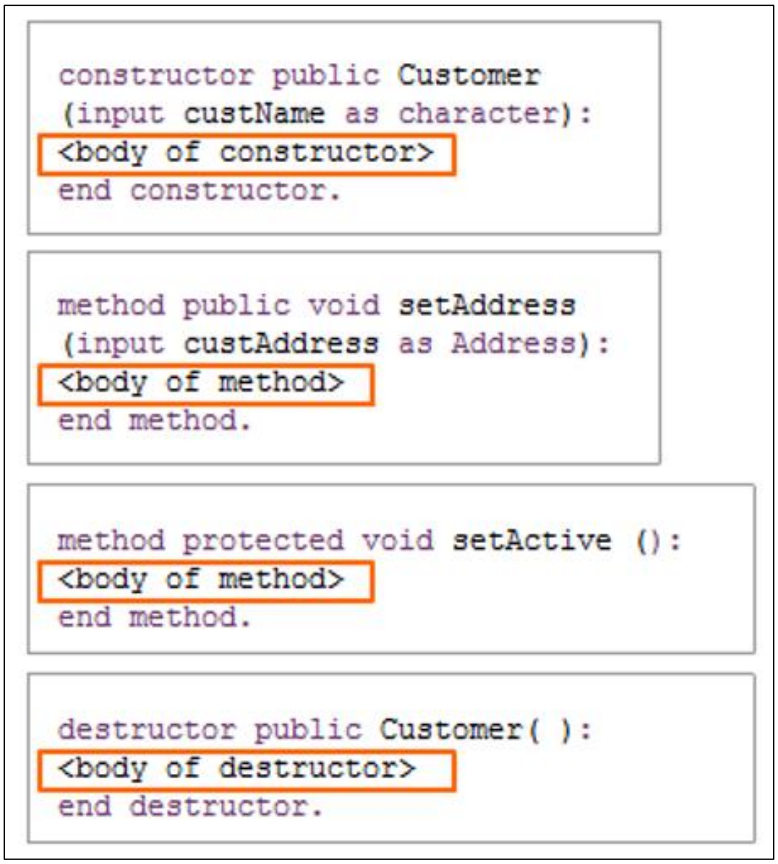
Try It 2.1: Defining classes

In this Try It, you will define data members, constructors, and methods for the *Emp* and *Dept* classes.

The exercise steps take approximately 75 minutes to complete.

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

Accessing data members and calling methods within 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 this topic, you will learn how to access data members and call methods within a class. In the next topic, you will learn how to access other classes.

Accessing 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

Example: Initializing 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.
```

Accessing 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 Element	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.

Examples: Accessing class methods from within the class

Recall that 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.

Accessing data members and calling methods in 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.

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.

To create and access other class instances in your ABL code, you must:

1. Write the using statement for the class.
2. Define the data member that will hold the reference to the instance of the class.
3. Create an instance of the class.
4. Use the instance to access the public data members and methods of the class.
5. Delete the instance of the class when you have finished using it.

Writing ABL using statements

The *using* statements in an ABL source code file are defined at the beginning of the file. The purpose of *using* statements is to specify which other classes your ABL class (or procedure) requires. *Using* statements specify the packages that contain the required classes.

Specifying *using* statements can reduce the code you have to write to accesses a class because you can refer to the class by name and you do not have to specify the package.

Here is the syntax for the *using* statement:

```
using  {<package>.<class name> | <package>.*}.
```

Syntax Element	Description
package	The name of the package in which the class resides in the workspace
class name	The name of the specific class that will be accessed in your code.
*	Enables your code to access all classes in a particular package

Example: Writing ABL using statements

Here is the *using* statement we must define at the beginning of the **Dept.cls** file because we want to access an *Emp* instance. Notice that the *using* statement for *Progress.Lang.**, which is used for all ABL class definitions, is also specified.

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

class Enterprise.HR.Dept:
. . .
/* rest of Dept class definition */
```

Defining 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 visibility 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 *private* or *protected*.

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

```
define [<visibility>] variable <name> as <class-name> [no-undo].
define [<visibility>] property <name> as <class-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	Specifies whether the variable or property will be *private*, *protected*, or *public*
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.
```

Creating 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 will hold the reference to the *Emp* instance. The next statement 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
                              ):
   define variable Empl as Emp no-undo.

   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.
```

Accessing 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
                               ):
   define variable Empl as Emp no-undo.

   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.
```

Calling 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 runtime. The types of the parameters must match the parameters defined for the method in the class definition.

Here is the *AddEmployee*() method of the *Dept* class. We call the public *Initialize*() method in the *Emp* class using the reference to the *Emp* instance. When we call *Initialize*(), we 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
                            ):

   define variable Empl as Emp no-undo.

   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.
```

Accessing 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

Example: Accessing 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.
```

Deleting an instance of a class

When you finish working with an instance of another class, you should delete it. When a class instance is deleted, it calls the destructor, if the class has one defined.

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

Check your understanding – Question 3

What ABL statement must you add at the beginning of a procedure or class file if the code is to access an instance of another class?

Choose the best answer.

A. *include*
B. *block-level*
C. *define class*
D. *using*

Answers are at the end of the lesson.

Check your understanding – Question 4

Suppose you have defined a variable of class type *Dept* as follows:

```
define variable DeptInstance as Dept no-undo.
```

How do you create an instance of the *Dept* class that will be referenced by the *DeptInstance* variable?

Choose the best answer.

A. `create object Dept set DeptInstance.`
B. `create new DeptInstance as Dept.`
C. `DeptInstance = new Dept().`
D. `new Dept(DeptInstance).`

Answers are at the end of the lesson.

Try It 2.2: Working with classes

Now that you have learned some basics about developing code for the constructors, methods, and destructor for a class, you will implement the *Emp* and *Dept* classes.

The exercise steps take approximately 1 hour to complete.

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

Testing 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. As a best practice you should create a separate project for your test procedures that has the same folder structure as your application.

In your test procedure, you should write code to:

- Set up the class test:
  - Specify a using statement for the class you will test.
  - Define a variable of the class type.
  - Open the output file for writing test results.
- Test the class:
  - Create multiple instances of the class, assigning its reference to the variable.
    - Make sure every constructor is called.
    - Pass a range of values to fully test the constructors.
  - Call each method of the class using the reference.
    - Pass a range of values to fully test the methods.
  - Write relevant data to the output file.
- End the class test:
  - Delete each instance when you no longer require it.
  - Close the output file.

Setting up the class test

In your 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.

Example

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 */

define variable EmpInstance as Emp no-undo.

/* set up the file for writing data*/
output to "/progress_education/openedge/IntroOOP/Log/testEmp.out".

/* more of test procedure */
```

Testing 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.

Example

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 the variable for holding the Emp instance */
define variable EmpInstance as Emp no-undo.
/* define the variable for holding the phone numbers extent */

define variable Phones as character extent 3 no-undo.

/* set up the file for writing data*/
output to "/progress_education/openedge/IntroOOP/Log/testEmp.out".

/* create an initialize an Emp instance */
EmpInsatnce = new Emp ().
assign
Phones[1] = "617-284-5937"
Phones[2] = "508-394-3928"
Phones[3] = "508-294-3927"
.
Emp: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 */
```

Ending 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>.
```

Example

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 variable for holding the Emp instance */
define variable EmpInstance as Emp no-undo.
/* define the variable for holding the phone numbers extent */
define variable Phones as character extent 3 no-undo.
/* set up the file for writing data*/
output to "/progress_education/openedge/IntroOOP/Log/test-Dept.out".
/* create an initialize an Emp instance */
EmpInsatnce = new Emp ().
assign
Phones[1] = "617-284-5937"
Phones[2] = "508-394-3928"
Phones[3] = "508-294-3927"
.
Emp: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.
```

Try It 2.3: Testing classes

In this Try It, you will develop simple test procedures to test the *Emp* class and the *Dept* class, and run them to ensure that the class code you have written executes correctly.

The exercise steps take approximately 60 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 the parts of an ABL class, including:
  - Data members
  - Constructors
  - Methods
  - A destructor
- Access data members and call methods within a class.
- Work with other classes, including:
  - Creating instances
  - Accessing data members and methods
  - Accessing a class instance dynamically
  - Deleting instances
- Test a class

Answers to *Check your understanding* questions

Question 1

A

Question 2

A, C, D, F

Question 3

D

Question 4

C

Notes

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