# Lesson 1: Introduction to Object-oriented programming

##### Lesson description

Block line

##### Course title

Introduction to OpenEdge

Block line

##### Product family

OpenEdge

Block line

## Lesson introduction

##### Page name: l1t1p000

Next: l1t1p005

Block line

##### Page type

Lesson introduction

Block line

##### Text

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

##### Page name: l1t1p005

Next: l1t1p010

Block line

##### Page type

Learning objectives

Block line

##### Text

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

### Prerequisites

##### Page name: l1t1p010

Next: l1t2p000

Block line

##### Page type

Prerequisites

Block line

##### Graphic/Slide

##### Text

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

##### Page name: l1t2p000

Block line

##### Page type

Conceptual animation

Block line

##### Graphics/Slide

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

##### Page name: l1t4p015

Next: l1t4p020

Block line

##### Page type

Basic content

##### Keywords

class, inheritance, reuse, derived class, super class

Block line

##### Graphics/Slide

##### Text

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

##### Page name: l1t2p005

Next: l1t2p010

Block line

##### Page type

Basic content

Block line

##### Keywords

Application architecture, architecture, client, business logic, business data, OERA, OpenEdge Reference Architecture

Block line

##### Graphics/Slide

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

##### Page name: l1t2p010

Next: l1t2p015

Block line

##### Page type

Basic content

Block line

##### Keywords

Process architecture, architecture, client, server, client process, server process

Block line

##### Graphics/Slide

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

##### Page name: l1t2p025

Next: l1t2p030

Block line

##### Page type

Basic content

Block line

##### Keywords

OpenEdge client, client, ABL client, Open Client, web services, web services client, WebSpeed, WebSpeed client, SQL, SQL client, Mobile client, Mobile App

Block line

##### Graphics/Slide

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.

##### Implementation notes

Block line

### Guided Exercise 1.1: Setting up your application development environment

##### Page name: l1t5p025

Next: l1t6p000

Block line

##### Page type

Guided exercise structure

Block line

##### Graphics/Slide

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

##### Page name: l1t7p000

Next:

Block line

##### Page type

Lesson summary

Block line

##### Graphic/Slide

##### Text

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

**Notes**