DEVELOPING OBJECT ORIENTED APPLICATIONS

By now, everybody should be comfortable using form controls, their properties, along with methods and events of the form class. In this unit, we discuss creating classes and how to use the properties and methods of the objects defined by those classes.

Review of class and object concepts

- An object is a self-contained unit that has properties (data) and methods (operations).
- A class is the code that defines the properties and methods of an object.
- An object is an instance of a class, and the process of creating an object from a class is called instantiation.
- If a class is based on an existing class called a base class, the new class inherits the properties and methods of the base class.

To illustrate the concepts today, we will develop an invoice application that is implemented with two user defined classes:

- An invoice class used to define invoice object
- A form class used to implement the user interface

How to develop the business classes

- Identify the *business objects* that the application requires.
- Design the *business classes* for the business objects by identifying all the properties and methods that any project will require.
- Code and test the business classes so they're ready for use by any projects in the application.

Invoice Object Properties

- CustomerName As String — Read-Write
- OrderTotal As Decimal — Read only (form cannot change)
- DiscountAmount As Decimal — Read only (form cannot change)
- InvoiceTotal As Decimal — Read only (form cannot change)

Invoice Object Methods

- AddItem(UnitPrice as Decimal, Quantity As Integer)
  Adds an item to the invoice and recalculates the Order Total, Discount Amount, and Invoice Total.
- WriteInvoice( )
  Writes current Invoice data to a file or database.

If we instantiate two or more instances of the same class, they will have all of the same properties and methods. The properties in each instance may have different values, though. These two Invoice objects have been instantiated from the Invoice class

Invoice1
CustomerName = Mark Adams OrderTotal = 139.65 DiscountAmount = 13.96 InvoiceTotal = 125.69

Invoice2
CustomerName = Bill Jones OrderTotal = 100.00 DiscountAmount = 10.00 InvoiceTotal = 90.00

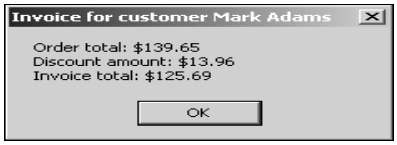
Three Types of Classes for Business Applications

A commercial application often requires three distinct types of classes:

- User interface classes to perform all of the processing related to forms
- Business logic classes to perform the actual business operations
- Database classes to save and retrieve data for the business objects.

Separating the user interface and the data retrieval from the business logic greatly simplifies the design and maintenance of the application.

How the Invoice Application is Implemented



This application is implemented by a form class and the Invoice class.

The form class is a user interface class that defines the form and its controls, accepts the user entries, and responds to any user actions.

The Invoice class is a business class that defines the properties and methods of each Invoice object.

- To begin an invoice, the user enters a customer name and the unit price and quantity for the first item, then clicks the Add Item button.
- The invoice will multiply the unit price and quantity to determine the item total
- It adds the item total to the order total, calculates the discount, and determines the invoice total.
- To add another item to the invoice, the user enters a new unit price and quantity, and then clicks the Add Item button again.
- Finally, the user will click the New Invoice button. A message box with the totals appears, and the form is cleared (we are not going to design a database class for this example).

The Code for the Form Class

Here is the form class that uses the properties and methods that we discussed earlier

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim Invoice As Invoice

    Private Sub Form1_Load(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles MyBase.Load
        Invoice = New Invoice()
        txtCustomerName.Focus()
    End Sub

    Private Sub btnAddItem_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnAddItem.Click
        If ValidData.PositiveNumber(txtUnitPrice, "Unit Price") Then
            If ValidData.PositiveInteger(txtQuantity, "Quantity") Then
                Invoice.AddItem(txtUnitPrice.Text, txtQuantity.Text)
                lblOrderTotal.Text = FormatNumber(Invoice.OrderTotal)
                lblDiscountAmount.Text = FormatNumber(Invoice.DiscountAmount)
                lblInvoiceTotal.Text = FormatNumber(Invoice.InvoiceTotal)
                txtUnitPrice.Text = ""
                txtQuantity.Text = ""
                txtUnitPrice.Focus()
            End If
        End If
    End Sub

      Private Sub btnOK_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnNewInvoice.Click
        Invoice.CustomerName = txtCustomerName.Text
        MessageBox.Show("Invoice number " & Invoice.NextInvoiceNumber & _
            " dated " & Invoice.InvoiceDate & " for " & _
            Invoice.CustomerName & " totaling $" _
            & Invoice.InvoiceTotal & ".")
        Invoice = New Invoice()
        lblOrderTotal.Text = ""
        lblDiscountAmount.Text = ""
        lblInvoiceTotal.Text = ""
        txtCustomerName.Text = ""
        txtUnitPrice.Text = ""
        txtQuantity.Text = ""
        txtCustomerName.Focus()
    End Sub

    Private Sub btnExit_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnExit.Click
        Me.Close()
    End Sub


End Class
```

Object variable declaration

We have declared an object variable of type Invoice. This variable is necessary to refer to the data for the Invoice objects. This is a form level or module level variable

Form1 Load Event Procedure

An instance of the Invoice class is created and assigned to the object variable. That is, we create an Invoice object. The focus is moved to the text box for the customer name and the New Invoice button is disabled to prevent users from creating another invoice object.

AddItem Click Event Procedure

The AddItem method of the Invoice object is invoked and sent the UnitPrice and Quantity entered by the user. This method does all of the required processing. The next three statements get the read-only properties of the Invoice object and update the appropriate labels on the form. Finally, the text boxes are cleared, the focus is moved to UnitPrice, and the New Invoice button is enabled.

NewInvoice Click Event Procedure

The customer name is assigned to the CustomerName property of the Invoice object and a MessageBox is displayed showing the four properties of the Invoice object. Typically, we would invoke a WriteInvoice method at this point. To keep things simple for this example, we just create a new Invoice object, assign it to the Invoice variable, and clear the text boxes and labels.

Note that all the data for the previous Invoice object are lost in this example. But, by looking at this form class, we can see how we access the properties and methods of a business class. It is quite similar to the way that we access the properties and methods of the .NET classes. We don't have to know how they are coded, we just have to know how to use them. This is one of the big advantages of encapsulation.

The Code for The Invoice Class

Now, let's look at the code for this particular business class, the Invoice class

```
Public Class Invoice
    Private sCustomerName As String
    Private dOrderTotal As Decimal
    Private dInvoiceTotal As Decimal
    Private dDiscountAmount As Decimal
    Private Shared iNextInvoiceNumber As Integer = 1000
    Private Shared dtmInvoiceDate As Date = Today()

    Public Shared ReadOnly Property NextInvoiceNumber() As Integer
        Get
            iNextInvoiceNumber += 1
            Return iNextInvoiceNumber
        End Get
    End Property

    Public Shared ReadOnly Property InvoiceDate() As Date
        Get
            Return dtmInvoiceDate
        End Get
    End Property

    Public Property CustomerName() As String
        Get
            Dim i As Integer
            Return sCustomerName
        End Get
        Set(ByVal Value As String)
            sCustomerName = Value
        End Set
    End Property

    Public ReadOnly Property OrderTotal() As Decimal
        Get
            Return dOrderTotal
        End Get
    End Property

    Public ReadOnly Property DiscountAmount() As Decimal
        Get
            Return dDiscountAmount
        End Get
    End Property

    Public ReadOnly Property InvoiceTotal() As Decimal
        Get
            Return dInvoiceTotal
        End Get
    End Property
```

```
    Public Sub AddItem(ByVal UnitPrice As Decimal, ByVal Quantity As Integer)
        dOrderTotal += UnitPrice * Quantity
        dDiscountAmount = Discount(dOrderTotal)
        dInvoiceTotal = dOrderTotal - dDiscountAmount
    End Sub

    Private Function Discount(ByVal OrderTotal As Decimal) As Decimal
        Dim dDiscountPct As Decimal
        Select Case dOrderTotal
            Case Is >= 500
                dDiscountPct = 0.3
            Case Is >= 200
                dDiscountPct = 0.2
            Case Is >= 100
                dDiscountPct = 0.1
            Case Else '<100
                dDiscountPct = 0
        End Select
        Return OrderTotal * dDiscountPct
    End Function
End Class
```

Included in the code for the Invoice class are:

- Private instance variable declarations
- Procedures to define the four properties
- The AddItem Method
- A private procedure to calculate the discount

At the top are declarations for four private variables called *instance variables.* They are created when an objected is instantiated from this class and are used to store the data for the properties of each object. Instance variables are normally declared as private variables so that they may not be referred to from outside the class.

Next are the four property procedures. They provide access to the four instance variables from outside the class. They each have a Get procedure that is used to retrieve the value of the property from the corresponding instance variable. The CustomerName property procedure also has a Set procedure that is used for setting the value of the property.

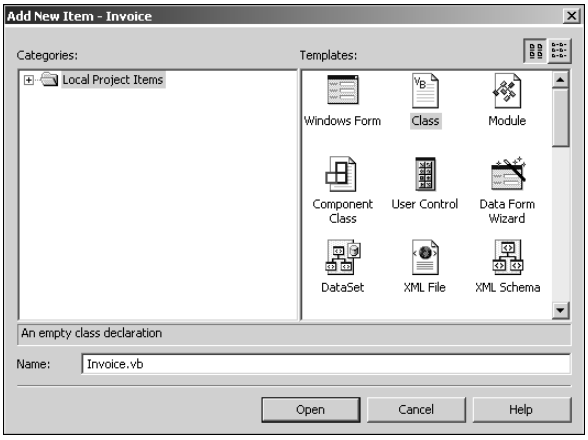
There is a public subprocedure that implements the AddItem method. AddItem accepts two arguments, UnitPrice and Quantity, then multiplies the values of the two arguments and adds the result to the order total. AddItem then calls Discount, a private function, which returns the discount amount, which is subtracted from the order total to get the invoice total.

Note that AddItem may be referred to from other objects in the application, Discount may not.

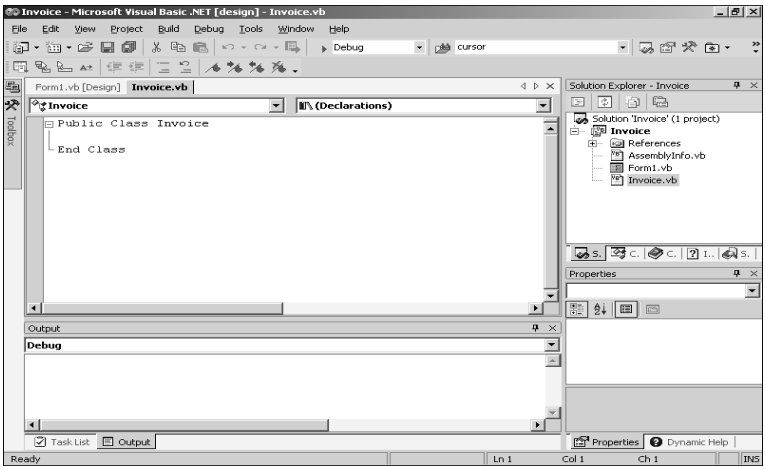
The Public keyword is used to identify the properties and methods available from outside the class. The Private keyword is used to identify instance variables, functions, and subprocedures that may only be accessed internally. This illustrates the concept of encapsulation.

The programmer can hide, or encapsulate, some data operations of a class while exposing others. Private variables and procedures are hidden. Public variables, public function and sub procedures provide the exposed interface to the data and operations of the class. Encapsulation lets the user of a class think of it as a black box that provides useful properties and methods.

Adding a class



To create a user defined class, we start by adding a class file to the application. Click on Project…Add Class and we get a Add New Item dialog box. We type in the name we want to use for the class, then click Open. This adds a class file to the project, which appears in the solution explorer with the .vb extension.



The Class and End Class statements are automatically added to the class. We may then ender the code for the class between those two statements.

The syntax for the Property statement

```
Public [ReadOnly|WriteOnly] Property name As type
   [Get
       [statements]
       {name = propertyvalue} | {Return propertyvalue}
    End Get]
   [Set(ByVal varname [As datatype])
       [statements]
        propertyvalue = newvalue
    End Set]
End Property
```

We use the Property statement to code a property procedure for each property that the class defines. We can create a property that is read-only, write-only, or read/write. One or both of the two accessors, Get and Set will appear, which provide access to the property values. The Get statement is used to service a request to retrieve the property value. The Set statement is used when a request is made to set the property value.

A procedure that defines a read/write property

```
Public Property CustomerName() As String
    Get
        Return sCustomerName
    End Get
    Set(ByVal Value As String)
        sCustomerName = Value
    End Set
End Property
```

When you start the property statement without the ReadOnly or WriteOnly keywords, Get/End Get and Set/End Set pairs are automatically generated. You need to add the code in the middle.

Since a Get returns a value, it is coded similarly to a function by using the Return keyword, or setting the name of the property to the return value.

This example will respond to requests for the value of the CustomerName property by returning the value of the instance variable sCustomerName. Or, it can be used to set the property value by assigning the value of the Value argument to sCustomerName.

A procedure that defines a read-only property

```
Public ReadOnly Property OrderTotal() As Decimal
    Get
        OrderTotal = dOrderTotal
    End Get
End Property
```

This example will retrieve the value of the OrderTotal property by using an assignment statement to assign the value of the instance variable to the property identifier.

When you start a Property statement and use the ReadOnly keyword, only a Get/End Get pair is generated.

A procedure that defines a write-only property

```
Public WriteOnly Property InterestRate() As Decimal
    Set(ByVal Value As Decimal)
        dInterestRate = Value
    End Set
End Property
```

This example shows the code used to set the InterestRate property. It may not be used to retrieve the value of this property. The instance variable sInterestRate is set to the value of the passed argument Value.

When you start a Property statement and use the WriteOnly keyword, only a Set/End Set pair is generated.

A method that does not return a value

```
Public Sub AddItem(ByVal UnitPrice As Decimal, _
            ByVal Quantity As Integer)
        dOrderTotal += UnitPrice * Quantity
        dDiscountAmount = Discount(dOrderTotal)
        dInvoiceTotal = dOrderTotal - dDiscountAmount
    End Sub
```

Methods are nothing more than Public sub procedures and functions. This example shows a method the accepts two arguments, but does not return a value. It is used to set the instance variables. That is, it will set the values for the class properties.

A method that returns a value

```
Public Function InvoiceTotal( _
            ByVal UnitPrice As Decimal, _
            ByVal Quantity As Integer) As Decimal
        dOrderTotal += UnitPrice * Quantity
        dDiscountAmount = Discount(dOrderTotal)
        dInvoiceTotal = dOrderTotal - dDiscountAmount
        Return dInvoiceTotal
    End Sub
```

This example will accept two arguments and returns the value of dInvoiceTotal. A function declaration is used.

Note that in these methods, the Public keyword is used. This is often not the case in ordinary sub and function procedures.

Defining an object variable and instantiating an object

In one statement:

```
Dim Invoice As New Invoice()
```

In two statements:

```
Dim Invoice As Invoice()
Invoice = New Invoice()
```

Before we can instantiate a class, we have to create an object variable to hold the object. The Dim statement is used to define the object variable and names the class from which the object will be created. If we include the New keyword, then a new object will be created. If we do not include the New keyword, then an object will not be created right then. We can also create an object from the class by including the New keyword in an assignment statement.

Referring to object properties and methods

```
Invoice.CustomerName = txtCustomerName.Text
lblOrderTotal.Text = Invoice.OrderTotal
Invoice.AddItem(dUnitPrice, iQuantity)
```

We can refer to the properties and methods of our user-defined classes in exactly the same way as we refer to properties and methods in the .NET classes. We type the object name, followed by the dot operator, followed by the name of the property or method.

The first example assigns the text entered by a user to the CustomerName property of the Invoice object. The second example writes the value of the OrderTotal property of the Invoice object to a label control. The last example invokes the AddItem method of the Invoice object and passes two parameters.

Dereferencing an object

```
Invoice = Nothing
```

When we finish with an object, we do not normally need to do anything with it. The memory will be deallocated when the application completes. If we want to deallocate memory sooner, there is a way to dereference an object, by setting its object variable to nothing. This will release the memory and the runtime environment's garbage collection routine will take it from there.