In this lecture we will briefly examine a few new controls and then introduce you to programming with Objects. The textbook introduces these topics in chapter 9 and 11.

**ListBoxes**

We have already seen how to add items to listboxes. Another common use of the listbox is to retrieve the item that is selected.

We can do this with the listbox.Items property. It turns out that this is actually an array that we can access:

- `listBox.Items.Count` - Number of items in the list box
- `listBox.SelectedIndex` - Index in array of selected item (-1 if none)
- `listBox.Items(n)` - Value of item at index n in the listbox
- `listBox.Items.RemoveAt(n)` - Removes item at index n in the listbox

(Short demo in class)

**Combo Box**

The Combo Box control is like a textbox with a pull-down menu of choices. We can access the user’s selection with:

```
comboBox.Text
```

The list of choices that the user can pick is an array of Items. We can add to the items just like with a listbox:

```
comboBox.Items.Add(newItem)
```

(Short demo in class)

**OpenFileDialog Control**

The OpenFileDialog is a control that allows you to browse for a file on the disk. To use it, drag the control on the form. It will not be visible.

Here is an example:

```
OpenFileDialog1.ShowDialog()
s = OpenFileDialog1.FileName
MsgBox("You selected " & s)
```

Once the path of the file is set it could be used in a IO.File.OpenText call, for example.
**Group Box Control**

The group box is used to group related sets of controls for visual effect. To use it, drag the group box onto the form. Then drag any new controls into the group box. The new controls will now be “part” of the group box.

The group box can be used to create different sets of radio buttons (upcoming).

(Short demo in class)

**Check Box Control**

The checkbox is a small box that can be checked or unchecked by the user. To see if something is checked or not you can inspected the Checked Property:

```
checkbox.Checked - True if checked, False if not
```

(Short demo in class)

**Radio Button Control**

The radio button operates like an old car radio. When one button is pushed, any other buttons “pop out”. For all radio buttons that are on a form, only one can be active at a time. If you would like to have multiple subgroups of radio buttons then they should be added to a GroupBox.

To see the value of a radio button, you can inspect the .Checked property just as with a checkbox.

(Short demo in class)

**Main Menu Control**

This control allows you to add a menu to the application. To use it, drag a Main Menu control to your form. Then double-click it in the form area. A menu designer will appear at the top of your form saying “Type Here”.

You can now type the name of the top-level of your menu. Click and type to fill in sub-areas. To attach code to the sub-areas, double-click on the menu item. The VB Code window will appear with an event for you to fill in code.

Try adding a menu for F)ile, O)pen, and C)lose.
ArrayList

The book does not describe the ArrayList, but it is a useful little class. The ArrayList behaves like a dynamic array. We can add and remove from it at will, and also delete items at specific locations. This can be useful when you don’t know how many items will be processed in your array and you can then use an ArrayList instead.

The following code sample illustrates the usage:

```vbnet
Dim aryList As New Collections.ArrayList
Dim i As Integer
aryList.Add("foo")
aryList.Add("bar")
aryList.Add("zot")
aryList.Add("bah")

Console.WriteLine("Size of aryList: " & aryList.Count)
Console.WriteLine("Contents:")
For i = 0 To aryList.Count - 1
    Console.WriteLine("At index " & i & " Value=" & CStr(aryList(i)))
Next
aryList.Remove("bar")
Console.WriteLine("Contents after remove bar:")
For i = 0 To aryList.Count - 1
    Console.WriteLine("At index " & i & " Value=" & CStr(aryList(i)))
Next
aryList.RemoveAt(0)
Console.WriteLine("Contents after remove:")
For i = 0 To aryList.Count - 1
    Console.WriteLine("At index " & i & " Value=" & CStr(aryList(i)))
Next
```

Output:

```
Size of aryList: 4
Contents:
At index 0 Value=foo
At index 1 Value=bar
At index 2 Value=zot
At index 3 Value=bah
Contents after remove bar:
At index 0 Value=foo
At index 1 Value=zot
Contents after remove:
At index 0 Value=zot
At index 1 Value=bah
```
As you can see, the ArrayList is convenient because we don’t need to declare the size up front like an array, and we are able to delete (or insert) items at specific places in the arraylist. Both of these operations are not allowed with normal arrays.

**Multiple Forms**

It turns out that the form that you use to create the layout of your program is itself an object. A VB.NET program can contain multiple forms. To add one, select “Project” from the main menu, and then select “Add New Windows Form”.

You can give a name to the new form which will be added to your project. In the Solution Explorer, you can switch back and forth between the form you wish to design.

To display the new form, use the following code:

```vbnet
Dim secondForm As New Form2
secondForm.Show()
```

Assuming the second form is named “Form2” this will display it. Using the “Show” method, one can switch back and forth between the new form and the original form. If you would like to show the second form as a dialog (i.e. the user can’t switch back to the first form until the new one is closed) then use the ShowDialog method:

```vbnet
secondForm.ShowDialog()
```

You can even have your form make a duplicate of itself!

```vbnet
Dim secondForm As New Form1
secondForm.Show()
```

A common operation is to want to access variables set in a second form back in the first form. To do this, you can make class variables (or sub/functions) **Public** and these are then accessible from outside the form. For example, here is code for Form2:

```vbnet
Public Class Form2
    Inherits System.Windows.Forms.Form
    Public y As Integer

    Private Sub Form2_Load(...) Handles MyBase.Load
        y = 5 ' Normally more complex logic to set the value of y!
    End Sub
End Class
```

We can do the following now from Form1:

```vbnet
secondForm.ShowDialog()
MsgBox(secondForm.y)
```

The value set into variable `y` is now accessible from our original Form.
Introduction to Classes and Objects

The preceding material regarding multiple forms is a nice transition into describing classes and objects because the Form is just a specific type of class.

A **class** is short for classification, and in object-oriented-speak, it corresponds to a user-defined specification for an object. The class is the definition; you can think of it like the blueprint for a complex device. It says how the device works, but if all you have is the blueprint then you can’t actually use the device yet.

We will define classes with two major subcomponents:

- **Member Data:** these are variables, properties, or adjectives of interest regarding the class. For example, if a class is “Auto” then the member data might specify variables to hold the make, VIN, owner, etc.

- **Methods:** these are functions or subs that perform some action regarding the class. For the “Auto” class we might have methods to brake, accelerate, shift, etc.

A class only specifies how some device works, like a blueprint. To use the device, you need some **instantiation** of the blueprint; i.e. the device must be constructed. The instantiation is called an **instance** and is sometimes referred to as the object. To create an instance of a class we use the keyword **new**. We’ve seen some examples of this already, e.g. with Forms and the Random class.

To make a new class, from the “Project” menu select “Add New Class”. Here is the format to define a class:

```
Class ClassName
    PublicOrPrivate ClassVar1 As DataType
    PublicOrPrivate ClassVar2 As DataType
    ...
    PublicOrPrivate Function FunctionName..
    PublicOrPrivate Sub SubName...
    ...
    PublicOrPrivate Property ...
End Class
```

The **PublicOrPrivate** is either the word **Public** or **Private**. If set to public then the variable, function, subroutine, or property is accessible from outside the class (using the dot notation, e.g. varname.ClassVar1). If set to private then the variable, function, subroutine, or property is accessible only inside the class. This principle supports the notion of data-hiding: data and variables that the user doesn’t need to see should be hidden to prevent them from being mucked up accidentally (e.g. the innards of the auto are hidden from a driver).
All of these options, variables, subroutines, functions, and properties are optional. We’ll talk about what the properties do a bit later.

A common notation is to preface the letters `m_` in front of any class variables. This is to denote that they are **member** variables, as opposed to local variables.

For example, instead of:

```plaintext
Public value As Integer
```

One might use:

```plaintext
Public m_value As Integer
```

Here is an example for a simple Money class. For now it only contains variables; later we will add some subroutines and functions and properties.

```plaintext
Public Class Money
  Public m_dollars As Integer
  Private m_cents As Integer
End Class
```

From another place in our program, such as in a button click event on a form:

```plaintext
Dim m1 As New Money
m1.m_dollars = 10 ' Legal, Public
m1.m_cents = 20  ' ILLEGAL, Private
```

This program will generate a compiler error since we are trying to access a private variable from outside the class.

For now let’s set both member variables in the class to public:

```plaintext
Public Class Money
  Public m_dollars As Integer
  Public m_cents As Integer
End Class
```

Here is some code that uses this class:

```plaintext
Dim m1 As New Money
Dim m2 As New Money

m1.m_dollars = 3
m1.m_cents = 40
m2.m_dollars = 10
m2.m_cents = 50
Console.WriteLine(m1.m_dollars & " " & m1.m_cents)
Console.WriteLine(m2.m_dollars & " " & m2.m_cents)
```
The output of this program is:

3  40
10 50

When the program reaches the print statement, we have created two separate instances of the Money object, each with different values stored in their member variables:

<table>
<thead>
<tr>
<th>m_dollars: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_cents: 40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>m_dollars: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_cents: 50</td>
</tr>
</tbody>
</table>

This can be quite convenient, because we can now associate multiple variables together in a single object. While both of these variables were of type integer in this example, the types could be anything. For example, a class to represent an Employee might contain variables like the following:

```vbnet
Class Employee
    Public m_Name As String
    Public m_Age As Integer
    Public m_HourlyWage As Double
    Public m_IDNumber As Long
End Class
```

In this example, we are associated different variable types with the Employee object. This is a powerful construct to help organize our data efficiently and logically.

### Class Constructors

Because we use classes to encapsulate data types, it is essential that class objects be initialized properly. When we defined the Money class, we were relying upon the user to set the value for dollars and cents outside the class. What if the client forgets to initialize the values? This can be such a serious problem that VB.NET provides a mechanism to guarantee that all class instances are properly initialized, called the class constructor.

A class constructor is a Sub with the special name of `New`. We can even make multiple constructors with multiple parameters, to differentiate different ways a class may be initialized. Below are two constructors for the Money class along with a subroutine to print the currency value:
Public Class Money
    Private m_dollars As Integer
    Private m_cents As Integer

    ' This is the default constructor, invoked if we create
    ' the object with no parameters
    Public Sub New()
        m_dollars = 1
        m_cents = 0
    End Sub

    ' This constructor invoked if we create the object with
    ' a dollar and cents value
    Public Sub New(ByVal initialDollars As Integer, ByVal initialCents As Integer)
        m_dollars = initialDollars
        m_cents = initialCents
    End Sub

    ' This subroutine prints the value out
    Public Sub PrintValue()
        Console.WriteLine(m_dollars & "." & m_cents)
    End Sub
End Class

Code in a button click event to use the money class:

    Dim m1 As New Money
    Dim m2 As New Money(5, 50)

    m1.PrintValue()
    m2.PrintValue()

When this program runs, the output is:

    1.0 ← From m1
    5.50 ← From m2

When we create m1, we give no parameters in Money(). This invokes the default constructor, which is given as:

    Public Sub New()

This code initializes dollars to 1 and cents to 0.

When we create m2, we give two parameters in Money(5,50). VB.NET will then search for a constructor that has two parameters that match the ones provided. The constructor that is found is then invoked:

    Public Sub New(ByVal initialDollars As Integer, ByVal initialCents As Integer)
This code then sets the member variables to the input parameters, resulting in the output previously specified.

Let’s add some more code to the Money class to make it a bit more useful. In particular, since the class variables are private, let’s make a way for the user of the class to get and set the dollars and cents. One way to provide access is through a property block. The Get procedure of the property block is used to retrieve the value of a variable. The Set procedure of the property block is used to set the value of a variable.

Add the following to the Money Class:

```vba
' Provide access to get and set the dollars variable
Public Property Dollars() As Integer
    Get
        Return m_dollars    ' This behaves like a Function
    End Get
    Set (ByVal Value As Integer)
        m_dollars = Value   ' This behaves like a Sub
    End Set
End Property

' Provide access to get and set the cents variable
Public Property Cents() As Integer
    Get
        Return m_cents
    End Get
    Set (ByVal Value As Integer)
        m_cents = Value
    End Set
End Property
```

We can now access the property “Dollars” and “Cents” from the caller:

```vba
Dim m1 As New Money
Dim m2 As New Money(5, 50)

m1.Dollars = 25     ' Use the SET part of the Properties
m1.Cents = 75
m1.PrintValue()    ' Use the GET part of the properties
Console.WriteLine(m2.Dollars & "." & m2.Cents)
```

The output is:

```
25.75
5.50
```

One of the nice things about the Property block is that we can add more code to perform validation and control regarding what values we want to get and set. For example, let’s say that we did the following:
Dim m1 As New Money

m1.Dollars = 25 ' Use the SET part of the Properties
m1.Cents = 375
m1.PrintValue()

The output is: 25.375

This is not really desirable, because it really means we have 375 cents, but this would probably be interpreted as 37.5 cents. A better solution would be to turn groups of 100 cents into dollars. We can add the proper logic to do this in the Property block:

' Provide access to get and set the cents variable
Public Property Cents() As Integer
Get
    Return m_cents
End Get
Set(ByVal Value As Integer)
    ' Increment dollars if we have more than 99 cents
    If Value > 99 Then
        m_dollars += Math.Floor(Value / 100)
        m_cents = Value Mod 100
    End If
End Set
End Property

The output now becomes: 28.75

Notice the abstraction we have implemented in the Cents property. If we ever add together something more than 100 cents, then we automatically update the cents into dollars. This logic is hidden for us in the Money class simply by executing the line of code:

moneyVar.Cents = 375

If we had just made the dollars and cents variables be public variables, then it would allow the outside user to set the cents to values over 100, possibly causing errors in how the program is interpreted.

By also making the dollars and cents private, we have the option to change the internal details and have these changes hidden from a user of the class. For example, we might decide to use a single variable of type Double to store the dollars and the cents. Then we could provide the proper logic in the Dollars and Cents properties to set and extract the dollar and cent amounts out of the Double and return them as an Integer. The user of the class won’t see any difference.

Next we’ll see some more examples of using classes, and finally get a peek at the concepts of inheritance and polymorphism.